CLOUD
SIGNATURE
CONSORTIUM

Architectures and protocols for remote signature applications

version 2.2.0.0

Contents

Foreword

This document is a work by members of the Cloud Signature Consortium, a nonprofit association founded by industry and academic organizations for building upon existing knowledge of solutions, architectures and protocols for Cloud-based Digital Signatures, also defined as “remote” Electronic Signatures.

The Cloud Signature Consortium has developed the present specification to make these solutions interoperable and suitable for uniform adoption in the global market, in particular – but not exclusively – to meet the requirements of the European Union's Regulation 910/2014 on Electronic Identification and Trust Services (eIDAS) [i.1], which formally took effect on 1 July 2016.

Revision history

Version	Date	Version change details
0.1.7.9-PR	14/02/2017	Public Pre-Release for early implementations
1.0.2.4-PR	24/09/2018	V1 Pre-Release for public comments
1.0.3.0	13/12/2018	V1 Public Release
1.0.4.0	28/06/2019	V1 Updated with new IPR information and errata
2.0.0.0	25/03/2022	V2 Pre-Release for public comments
2.0.0.1	19/08/2022	V2 Pre-Release after solving public comments
2.0.0.2	20/04/2023	Correction of some typos and credentials/list example output
2.1.0.0	03/12/2024	V2.1 Public release with updates for signatures/signDoc and authorization
2.1.0.1	22/01/2025	Minor typographic corrections and update to signatures/signPolling
2.2.0.0	04/11/2025	V2.2 Public release

Acknowledgements

This work is the result of the contributions of several individuals from the Technical Working Group of the Cloud Signature Consortium and some additional contributors. In particular, the following people have provided a significant contribution to the drawing up and revision of the present specification:

Ałła Stoliarowa-Myć, Andrea Röck, Andrea Valle, Andrew Papastefanou, Andreas Vollmert, Arno Fiedler, Bernd Wild, Carlos Ares, Christoph Bimminger, Cornelia Enke, Daniel Fett, David Ruana, Davide Barelli, Enrico Entschew, Francesco Barcellini, Franck Leroy, George Gugulea, Giuliana Marzola, Giuseppe Damiano, Harald Bratko, Håvard Grindheim, Iñigo Barreira, Jon Ølnes, Kapil Khattar, Kåre Fiedler Christiansen, Dr. Kim Nguyen, Klaus-Dieter Wirth, Luca Boldrin, Luigi Rizzo, Mangesh Bhandarkar, Marc Kaufman, Marcin Szulga, Meena Muralidharan, Michael Traut, Patrycja Wiktorczyk, Patryk Sosiński, Peter Lipp, Prof. Reinhard Posch, Sander Dijkhuis, Simon Roth, Sven Prinsloo, Thomas B. Pedersen, Thomas Pielczyk, Torsten Lodderstedt.

Introduction

For a long time, transactional e-services have been designed for typical end-user devices such as desktop computers and laptops. Accordingly, existing digital signature solutions are tailored to the characteristics of these devices as well. This applies to smart card and USB token-based solutions. These traditional signature solutions implicitly assume that the user accesses e-services from a desktop or laptop computer and in addition uses a smart card or token to create any required digital signatures. This assumption is not valid any longer. During the past few years, smartphones, tablets and other mobile end-user devices have started to replace desktop and laptops computers.

This situation raises several challenges for e-services: smart cards and tokens cannot be easily connected to smartphones and other mobile devices, or cannot at all. For instance, smartphones usually do not provide support for USB devices, which is the common technology for smart card based solutions.

In this regard, recent regulations in various regions worldwide – like eIDAS [i.1] in the European Union – have introduced the concept of electronic signatures that are created using a “remote signature creation device”, which means that the signature device is not anymore a personal device under the physical control of the user, but rather it is replaced by cloud-based services offered and managed by a trusted service provider.

This is, in summary, the scope of the Cloud Signature Consortium, also known as CSC, aiming at the definition of a common architecture, building blocks and communication protocols intended for creating a standard API to integrate the essential components of a remote signature solution established among different service providers and consumers.

Where the context of the eIDAS Regulation is applicable, this specification, and the term “remote signature solution” herein developed, aim to cover solutions for remote electronic signatures and remote electronic seals, in the domains of both qualified and advanced electronic signatures / seals.

Intellectual Property Rights

The Intellectual Property Rights Policy (IPR Policy) of the Cloud Signature Consortium is available at https://cloudsignatureconsortium.org/about-us/intellectual-property/.

Trademark notice

The Cloud Signature Consortium logo is a Registered Trademark of the Cloud Signature Consortium:
EU Trademark number 015579048.

Essential Patents

IPRs essential or potentially essential to the present document may have been declared to the Cloud Signature Consortium. The information pertaining to these essential IPRs, if any, is available on request from the Cloud Signature Consortium secretariat at info@cloudsignatureconsortium.org.

No investigation, including IPR searches, has been carried out by the Cloud Signature Consortium. No guarantee can be given as to the existence of other IPRs not referenced in the present document which are, or may be, or may become, essential to the present document.

Legal notices

The Cloud Signature Consortium seeks to promote and encourage broad and open industry adoption of its standard.

1 Scope

When digital signatures are created within a device, the interfaces and functions are standardized, e.g. the API used by the application program to access the signature creation libraries and the interface to the smart card or similar device (if a device is used) holding the signing key. When digital signatures move to the cloud, the functions needed to create a digital signature can be distributed across several service instances, each carrying out one or more steps in the signature creation process. The interfaces between such services are however until now not standardized.

The Cloud Signature Consortium aims to fill this gap in standardization by defining the architectural design, communication protocols, application programming interfaces, data structures, and technical requirements needed to establish interoperable solutions for cloud-based digital signatures. While these specifications are applicable in a wide variety of use cases with different security requirements, the fulfilment of requirements imposed by the eIDAS Regulation of the EU [i.1] is particularly addressed, supporting the creation of "advanced" or "qualified" electronic signatures and electronic seals in the cloud.

This document contains technical specifications that are intended for use by applications for creating digital signatures in the cloud and by a variety of applications consuming these services. By implementing their services according to these specifications, service providers can ensure that services are applicable as parts of complete digital signature systems in the cloud in a plug and play manner.

Existing standards and open specifications are considered by the consortium as far as applicable.

The following are out of scope of this specification:

- Policy requirements for (qualified and other) service providers; this is an area of standardization covered by ETSI.
- Signing key creation and enrollment; although keys MAY be created by the remote service during the signing workflow, these activities are not covered by specific API methods.
- Signature and certificate formats; use of the standards specified by ETSI is RECOMMENDED.
- Signature validation; this will be addressed in future specifications from the Consortium.
- Security evaluation and requirements for hardware components used to hold signing keys (HSM – hardware security module); this is being standardized by CEN in Europe and FIPS in the USA.
- Internal functionality and internal interfaces in service provider systems.

Note that the current specifications mainly cover architectures where the signing key is held "in the cloud", i.e. by a signature creation device managed by a service provider. Architectures where the signing key is in the hand of the signer, stored in the user's device or in an attached smart card or similar, are not covered as a particular case. The consortium will consider the need for further specifications covering situations where a user device holding the signing key interacts with cloud services for digital signature creation, e.g. cloud services MAY be used for document storage, hash computation, and signature formatting.

2 Interpretation of Requirement Levels

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

3 References

3.1 Normative references

The following documents, in whole or in part, are normatively referenced in this specification and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or errata) applies.

[1] IETF RFC 2119: “Key words for use in RFCs to Indicate Requirement Levels”.

[2] IETF RFC 3161: “Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)”.

[3] IETF RFC 3986: “Uniform Resource Identifier (URI): Generic Syntax”.

[4] IETF RFC 4514: “Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names”.

[5] IETF RFC 4627: “The application/json Media Type for JavaScript Object Notation (JSON)”.

[6] IETF RFC 4648: “The Base16, Base32, and Base64 Data Encodings”.

[7] IETF RFC 5246: “The Transport Layer Security (TLS) Protocol Version 1.2”.

[8] IETF RFC 5280: “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”.

[9] IETF RFC 5646: “Tags for Identifying Languages”.

[10] IETF RFC 5816: “ESSCertIDv2 Update for RFC 3161”.

[11] IETF RFC 6749: “The OAuth 2.0 Authorization Framework”.

[12] IETF RFC 6750: “The OAuth 2.0 Authorization Framework: Bearer Token Usage”.

[13] IETF RFC 7009: “OAuth 2.0 Token Revocation”.

[14] IETF RFC 7235: “Hypertext Transfer Protocol (HTTP/1.1): Authentication”.

[15] IETF RFC 7518: “JSON Web Algorithms (JWA)”.

[16] IETF RFC 7519: “JSON Web Token (JWT)”.

[17] void

[18] IETF RFC 8017: “PKCS #1: RSA Cryptography Specifications Version 2.2”.

[19] IETF RFC 8446: “The Transport Layer Security (TLS) Protocol Version 1.3”.

[20] IETF draft-ietf-oauth-security-topics: “OAuth 2.0 Security Best Current Practice”

[21] ETSI TS 119 312: “Electronic Signatures and Infrastructures (ESI); Cryptographic Suites”.

[22] ISO 3166-1: “Codes for the representation of names of countries and their subdivisions — Part 1: Country codes”.

[23] IETF RFC 8414: “OAuth 2.0 Authorization Server Metadata”

[24] IETF RFC 7591: “OAuth 2.0 Dynamic Client Registration Protocol”

[25] IETF RFC 7636: “Proof Key for Code Exchange by OAuth Public Clients”

[26] void

[27] IETF RFC 9396: “OAuth 2.0 Rich Authorization Requests”

[28] IETF RFC 9126: “OAuth 2.0 Pushed Authorization Requests”

[29] ETSI EN 319 122-1 “Electronic Signatures and Infrastructures (ESI); CAdES digital signatures; Part 1: Building blocks and CAdES baseline signatures”

[30] ETSI EN 319 132-1: “Electronic Signatures and Infrastructures (ESI); XAdES digital signatures; Part 1: Building blocks and XAdES baseline signatures”

[31] ETSI EN 319 142-1: “Electronic Signatures and Infrastructures (ESI); PAdES digital signatures; Part 1: Building blocks and PAdES baseline signatures”

[32] ETSI TS 119 182-1: “Electronic Signatures and Infrastructures (ESI); JAdES digital signatures; Part 1: Building blocks and JAdES baseline signatures”

[33] IETF RFC 6960: “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP”

[34] ETSI EN 319 102-1 “Electronic Signatures and Trust Infrastructures (ESI); Procedures for Creation and Validation of AdES Digital Signatures; Part 1: Creation and Validation”.

[35] IETF RFC 7515: “JSON Web Signature (JWS)”.

[36] ETSI EN 319 102-1 "Electronic Signatures and Trust Infrastructures (ESI); Procedures for Creation and Validation of AdES Digital Signatures; Part 1: Creation and Validation".

[37] Cloud Signature Consortium, "Data model for remote signature applications", version 1.0.0.

3.2 Informative references

The following documents, in whole or in part, are informatively referenced in this specification and may be a useful contribution for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or errata) applies.

[i.1] Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC.

[i.2] ETSI SR 019 020: "The framework for standardization of signatures; Standards for AdES digital signatures in mobile and distributed environment".

[i.3] IETF RFC 3447: "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1".

[i.4] IETF RFC 6101: "The Secure Sockets Layer (SSL) Protocol Version 3.0".

[i.5] CEN EN 419 241-1: "Trustworthy Systems Supporting Server Signing - Part 1: General System Security Requirements"

[i.6] ISO/IEC 19790: "Information technology - Security techniques - Security requirements for cryptographic modules"

[i.7] Hickman, Kipp, "The SSL Protocol", Netscape Communications Corp., Feb 9, 1995

[i.8] ETSI TS 119 001: "Electronic Signatures and Infrastructures (ESI); The framework for standardization of signatures; Definitions and abbreviations."

[i.9] void

[i.10] South African Act No. 25 of 30 August 2002: Electronic Communications and Transactions Act, 2002

[i.11] Web Authentication: An API for accessing Public Key Credentials Level 2, 2021

[i.12] OpenID Connect Core 1.0 incorporating errata set 2, 2023

[i.13] IANA "Level of Assurance (LoA) Profiles"

[i.15] IETF RFC 9068: "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens"

[i.16] IETF RFC 7662: "OAuth 2.0 Token Introspection"

4 Terms, definitions and abbreviations

4.1 Terms and definitions

For the purposes of this specification, the following terms and definitions apply.

access token: credentials used to access protected resources. It's a string representing an authorization issued to the client. The string is usually opaque to the client.

Note 1: As defined in IETF RFC 6749 [11].

authentication factor: piece of information and/or process used to authenticate or verify the identity of an entity.

Note 2: As defined in ISO/IEC 19790 [i.6].

EXAMPLE: A password or PIN.

authorization server: The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

Note 3: As defined in IETF RFC 6749 [11].

base64: Base64 as defined by RFC 4648 [6] Section 4, i.e. standard alphabet with padding SHALL be used. See also paragraph about Base64 in Conventions section of this document.

base64url: Denotes the URL-safe Base64 encoding as defined in RFC 4648 [6] Section 5 and further precised in RFC 7515 [35] Section 2 and Appendix C. Padding SHALL NOT be used.

credential: cryptographic object and related data used to support remote digital signatures over the Internet. Consists of the combination of a public/private key pair (also named "signing key" in CEN EN 419 241-1 [i.5]) and an X.509 public key certificate managed by a remote signing service provider on behalf of a user.

data to be signed representation: hash of the data to be signed formatted, which is used to compute the digital signature value.

Note 4: As defined in ETSI TS 119 001 [i.8].

digital signature: data appended to, or a cryptographic transformation (see cryptography) of a data unit that allows a recipient of the data unit to prove the source and integrity of the data unit and protect against forgery e.g. by the recipient [i.8]

Note 5: Digital signature is a technical term. Specifically, but not exclusively, aims at supporting legal terms as electronic signatures, advanced electronic signatures, qualified electronic signatures, electronic seals, advanced electronic seals, and qualified electronic seals as per Regulation (EU) No 910/2014 [i.1].

electronic signature: digital signature created by using a certificate issued to a natural person ensuring the integrity and origin of the document and the signatory commitment to the document content.

Note 6: Electronic signatures used in the present document are meant to specifically, but not exclusively, support the electronic signatures defined as per Regulation (EU) No 910/2014 [i.1].

electronic seal: digital signature created by using a certificate issued to a legal person or business unit ensuring the integrity and origin of the document, without necessarily committing to the content.

Note 7: Electronic seals used in the present document are meant to specifically, but not exclusively, support the electronic seals defined as per Regulation (EU) No 910/2014 [i.1].

identity proofing: process by which the identity of an applicant is verified by the use of evidence attesting to the required identity attributes

Note 8: Depending on how the claims will be used, different assurance levels will be required when verifying the claims.

remote service: service implementing the API described in this specification and delivered on the Internet.

remote signing service provider: service provider managing a set of credentials on behalf of multiple users and allowing them to create a remote signature with a stored credential.

Note 9: A remote signing service provider typically operates an HSM (or functionally equivalent multi-user secure device) and an authentication service. It manages the users and provides a signing service that can be accessed over the Internet by means of the API described in this specification.

Note 10: A remote signing service typically manages signing keys and certificates that are created before the signing operations take place. Another common scenario is when the signing key and the certificate are created in the course of a signing operation. In the present specification, this is referred to as "Short-Lived Credential Signing" (also called "ad-hoc" or "on-the-go" credential signing).

remote signature creation device: signature creation device used remotely from signer perspective to provide control of signing operation on its behalf of the signer.

short-lived credentials: temporary credentials created to sign a specific transaction where those credentials will then expire or be explicitly revoked shortly after being applied in the signature operation. Methods to create and manage short-lived credentials across multiple transactions will be handled in a future release of this specification.

Note 11: Once the end-user has had their claims successfully verified in an identity proofing process, they become eligible to sign with short-lived credentials. The assurance level of the claims associated with the identity will determine the trust level that can be achieved with the short-lived credentials.

signature activation data: set of data used to control a given signature operation, performed by a cryptographic module, on behalf of the signer.

signature activation module: configured software that uses the SAD in order that the signing keys are used under sole control of the signer.

Note 12: As defined in CEN EN 419 241-1 [i.5].

signature application: client application or service calling the remote signing service provider to create a remote signature.

Note 13: A *signature application* may be a *signature creation application*.

signature application provider: service provider managing a signature application and offering it as a service over the Internet or other communication channel.

signature creation application: application that accepts signer's original document and produces a signature or signed document in accordance with AdES [36].

Note 14: A *signature creation application* that invokes a CSC endpoint for remote signing is a special case of a *signature application*.

signer's document representation: hash value of signer's formatted document. As defined in [34].

signer's original document: some document types (e.g. PDF, XML, and Json) require formatting before SDR can be computed. signer's original document is the original document *before* any formatting.

signer's formatted document: some document types (e.g. PDF, XML, and Json) require formatting before SDR can be computed. signer's original document is the original document *after* any formatting. SDR is the hash of signer's formatted document.

Note 15: As an example for PDF, signer's formatted document contains the signature dictionary and other objects such as signature image.

4.2 Abbreviations

AdES: Advanced Electronic Signature

API: application programming interface

DTBSR: Data to be signed representation

HSM: hardware security module

RSCD: remote signature creation device

RSSP: remote signing service provider

SAD: signature activation data

SAM: signature activation module

SCAL1: sole control assurance level 1

Note 16: As defined in CEN EN 419 241-1 [i.5].

SCAL2: sole control assurance level 2

Note 17: As defined in CEN EN 419 241-1 [i.5].

SDR: signer’s document representation

SFD: signer’s formatted document

SOD: signer’s original document

SODR: signer’s original document representation (i.e. hash of SOD).

5 Conventions

This specification defers to the CSC Data Model (DM) [37] “Conventions” for data encodings and naming/casing. Unless an external standard mandates otherwise, Base64 and Base64URL usage (including padding rules) follow the DM. Wherever this API references a DM-defined data type, the DM’s encoding and naming rules SHALL apply.

6 Architectures and use cases

The present specification and the protocols defined herein aim to support different use cases. However, they focus on the scenario of remote signing defined for example as “the creation of remote electronic signatures, where the electronic signature creation environment is managed by a trust service provider on behalf of the signatory” in EU Regulation 910/2014 [i.1], recital §52.

This means that other scenarios for signing in distributed environments assisted by remote servers – like those described in ETSI SR 019 020 [i.2](“Standards for AdES digital signatures in mobile and distributed environment”) – are not covered in the present version of this specification. In particular, use cases where the signing key is contained within a signer's personal device are not covered: for example, signing a document located on a server with a private key contained in a mobile SIM card, or in a cryptographic device connected to a personal computer. These are relevant use cases, although not fitting in the core definition of “remote signature”, so they may be specifically covered in future updates of the specification.

6.1 Supported architectures

The current version of the specification focuses on the interface between the Signature Application and the remote signing service provider (RSSP), which performs the remote signing operations and also manages the lifecycle of the signer’s credentials in coordination with other trust-service components. The following figure shows a typical but not restrictive example of the architecture.

The following roles are used in this architecture:

Driving Application (DA) – interacts with the signer and orchestrates the signing process.
Relying Party (RP) – the entity requesting a signature.
Signature Creation Application (SCA) – prepares and creates the signature or signed document (e.g., CMS, PAdES); it may implement the `signatures/signDoc` endpoint.
Signature Creation Device (SCD) – generates the signature value over the data to be signed; it may implement the `signatures/signHash` endpoint.
Authorization Server (AS) – the OAuth 2.0 component where the signer authorizes the operation.
Remote Signing Service Provider (RSSP) – the trust service that operates the remote signing system.

Deployments may vary:
• Some services expose an SCA, handling the complete document-based signing flow.
• Others expose only an SCD, where the DA or an external SCA prepares the data and embeds the returned signature value.
Both models are valid and interoperable.

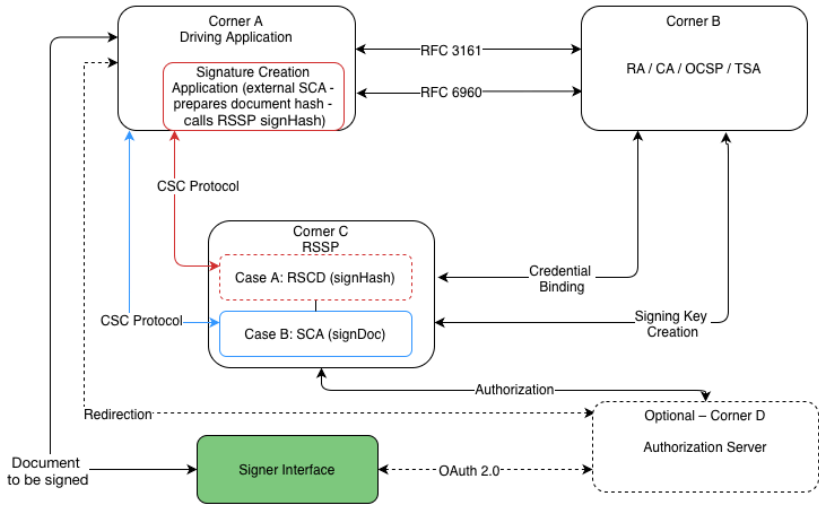


Figure 1: Remote signing corners

Figure 1: Remote signing corners

There are four main corners in the remote signing scenario.

The Signature Application, acting as an SCA or as a DA working with an external SCA, retrieves the document to be signed from the user and, when necessary, obtains certificates, revocation information, and time-stamps from the relevant trust service provider. It requests the remote signing service provider to generate the signature, either over the document itself (document-based flow) or over the document's hash value (digest-based flow), depending on the operation supported. Both approaches use the same authorization, credential, and trust framework.

The RSSP connects to the CA for the credential binding. In some cases, the CA may also be included in the process of creating the signing key. In addition to the credential binding process, the RSSP may also implement credential-management interfaces that allow the secure creation, retrieval, and deletion of signing credentials. These interfaces operate in coordination with the Certification Authority (Corner B) to ensure that credential generation and binding are performed under the same assurance and audit requirements as the signing operations.

Authorization for service or credential access can occur through the Signature Application or via redirection to an OAuth 2.0 Authorization Server (AS). In many deployments, the AS is operated by the RSSP. This maintains the four-corner model (RP ↔ DA ↔ SCA/SCD ↔ RSSP) and keeps signer consent and credential access within a single authorization context.

The redirect-based model employed by OAuth 2.0 allows the RSSP to utilize FIDO/WebAuth [i.11] or 3rd party identity providers (e.g. via OpenID Connect [i.12] or EUDI Wallet-based identity providers) for user authentication and credential authorization. This approach ensures that both document-signing and credential-management operations can be performed under a unified authorization flow, maintaining the same trust boundaries and four-corner interaction model described above.

7 Introduction to the remote service protocols API

Web applications and services use Application Programming Interfaces (APIs) to talk to each other. Technically speaking, in the web service context, an API is a set of programming instructions for accessing a Web-based software application or service.

The remote service protocols API allows a signature application to communicate with a remote service via the Internet by leveraging a sequence of calls to methods.

7.1 Format and syntax of the API

This specification defines Web services APIs that are based on technical standards and protocols such as HTTP and JSON. This API uses HTTP POST requests with JSON payload and JSON responses. JSON is an open-standard media type format as defined by RFC 4627 [5] that uses human-readable text to transmit data objects consisting of attribute-value pairs. These properties make JSON an ideal data-interchange language which is used as the most common data format for asynchronous communications.

The functions offered by the remote service are represented by HTTP RPC endpoints accepting arguments as JSON in the request body and returning results as JSON in the response body. For this reason, the HTTP header of the invocation method SHALL include a Content-Type: application/json header.

The remote service SHALL use HTTP version 1.1 or higher.

A JSON schema corresponding to the API defined in the present specification is available. See [JSON schema and OpenAPI description].

7.2 Remote service base URI

The remote service base URI defines the style and format of the HTTP endpoint URI of a remote service conforming to this specification.

The base URI contains the version number of the APIs that is implemented by the remote signing service provider. In the case of this specification, the version number SHALL be `v2`. Future versions of this specification MAY not be completely backward compatible.

```
https://service.domain.org/xxx/csc/v2/
```

The base URI SHALL start with an arbitrary URL defined by the service provider ('https://service.domain.org/xxx' in the example above) and SHALL end with '/csc/v2'. The endpoints of the API methods documented in this specification SHALL be concatenated to the base URI. An exception is given by the OAuth 2.0 methods, as defined in OAuth 2.0 Authorization, which MAY use URIs that are independent of the service base URI.

7.3 Integrity and confidentiality

A remote service conforming to this specification SHALL guarantee the integrity and confidentiality of the communication channel between the signature application and the remote service.

The integrity and confidentiality of the communication channel between the user and the signature application or the remote service are out of the scope of this specification.

The remote service SHOULD implement Transport Layer Security (TLS) in order to ensure the integrity and confidentiality of the communications. This prevents easy eavesdropping or impersonation if authentication credentials are hijacked. Another advantage of always using TLS is that guaranteed encrypted communications simplifies the authentication schemes, so for example simple mechanisms like Basic HTTP authentication can be used because the elements used in the authentication (username and password) are always transmitted over an encrypted channel.

The remote service MAY use other methods than TLS, for example using VPN.

TLS 1.3 as described in RFC 8446 [19] is, at the time of this writing, the latest version of TLS. Until TLS 1.3 is widely adopted, the previous version TLS 1.2 as described in RFC 5246 [7] SHALL be supported by remote services conforming to this specification and is the RECOMMENDED mechanism to use for interoperability reasons. TLS 1.2 provides access to advanced cipher suites that support elliptic curve cryptography and authenticated encryption with associated data (AEAD) block cipher modes. TLS 1.1 MAY be used, but it is also less secure. TLS 1.0 is considerably less secure and some security certifications like PCI DSS 3.1 explicitly forbid it, so remote services SHOULD NOT support it.

All versions of SSL (SSLv3 as defined in RFC 6101 [i.4] or SSLv2 as defined in [i.7]), the security protocol used before TLS, are considered insecure. Remote services conforming to this specification SHALL NOT implement SSL.

7.4 Remote service information

This specification defines a protocol to connect a signature application to a remote service. Other similar specifications exist in the industry, but they are typically proprietary and incompatible between each other, so if a signature application wants to support multiple remote services, then the development effort would increase significantly.

This specification has been designed to support modular services that may be implemented in line with the capacity and mission of the provider. This means that a remote service that supports this specification MAY implement only a subset of the API methods defined herein. In order to facilitate this approach, this specification defines the **info** method, which all remote services SHALL implement to allow the signature application to discover which of the API methods are supported.

In addition, the **info** method returns information on the remote service which may be useful to a calling application to access the functions and features of the service.

7.5 *clientData* parameter

Most methods allow to provide *clientData* as an optional input parameter. It can contain any arbitrary data from the signature application. This data allows the signature application to handle other application-specific data like, e.g., a transaction identifier.

The remote service MAY use this information, and it MAY also log this data together with information of the call. This parameter MAY expose sensitive data to the remote service. Therefore, it SHOULD be used carefully by signature applications.

7.6 Expressing algorithms

The present document expresses algorithms via Object IDentifiers (OID). OIDs are identifiers standardized by the Internal Telecommunication Union (ITU) and ISO/IEC to identify a specific object. They are represented by numbers, separated by dots, and are constructed in a tree-like structure. A list of the most common OIDs for algorithms used in signatures can be found in chapter 10 of ETSI TS 119 312 [21]. See also the OID repository http://oid-info.com in search of specific OIDs.

8 Authentication and authorization

This specification supports the following types of authentication and authorization:

a. Service authorization and authentication.
b. Credential creation authorization.
c. Credential deletion authorization.
d. Credential authorization.

8.1 Types of authorization

8.1.1 Service authorization and authentication

In order to protect the remote service from unauthorized access, this specification requires the signature application to obtain a valid “access token” to authorize the access to the APIs. This type of authorization is called service authorization. Various types of authorization mechanisms can be supported, and more will be supported in future versions, and the signature application SHALL adopt any of those available from the remote service as stated in the response to the **info** method, as defined in info.

The remote service MAY also adopt an indirect way of authorizing access to the API. The underlying communication channel with the signature application MAY ensure access control in a different way, for example with a private point-to-point LAN connection or through a VPN (Virtual Private Network).

The access to the APIs SHALL be authenticated.

When the authentication is under the control of the signature application provider, then the user SHALL be properly authenticated by this provider before getting access to the remote service. This scenario supports organizations that manage a user community with an existing form of authentication, for example a Bank managing the users from their Internet Banking service. This means that, in order to retrieve the signing credentials associated to a user, this organization would have to take care of the correspondence between the user identifier in their own domain and the user identifier in the remote service’s domain.

When the authentication is under the control of the remote service, the signature application SHALL perform a token-based authentication to the remote service by means of authentication factors collected from the user, preferably via an OAuth 2.0 authorization mechanism, or through HTTP Basic or HTTP Digest authentication. In case the signature application is not under the control of the user, OAuth 2.0 authorization SHOULD be used. In practice, the signature application will require the user to authenticate directly to the remote service using any of the available methods. This would offer an authentication mechanism even in case the signature application and the remote service have not previously established any form of service authentication.

This specification defines two methods to obtain service authorization:

- The oauth2/token method SHALL be used when an OAuth 2.0 authorization mechanism is supported by the remote service. The signature application will not collect any authentication factors from the user, but instead it will redirect to the remote service that will authenticate the user. See OAuth 2.0 Authorization for further information on how to implement OAuth 2.0 authorization.
- The auth/login method SHALL be used when OAuth 2.0 is not available and HTTP Basic or Digest authentication mechanisms are preferred and supported by the remote service. The signature application will collect the authentication factors from the user and will submit them to the remote service to obtain an authorization.

In both cases, if the user grants authorization, the remote service will return a service access token to the signature application. From then on, all authenticated requests to the API methods defined in this specification SHALL use an Authorization header with *Bearer* type followed by that service access token.

If the user does not grant the authorization, the authorization server will return an error message and no access to authenticated API methods will be possible.

8.1.2 Credential creation authorization

It requires authorization from the user to create a credential bound to that user.

This specification only support OAuth 2.0 authorization to grant credential creation authorization:

- The oauth2/authorize and oauth2/token methods are used to obtain an OAuth 2.0 access token that grants credential creation authorization.

8.1.3 Credential deletion authorization

Some RSSPs may have special requirements for authorizing the deletion of a credential. To support such requirements this specification defines credential deletion authorization.

This specification only support OAuth 2.0 authorization to grant credential deletion authorization:

- The oauth2/authorize and oauth2/token methods are used to obtain an OAuth 2.0 access token that grants credential deletion authorization.

8.1.4 Credential authorization

Accessing a credential for remote signing requires an authorization from the user who owns the signing key associated to it. As a special case, the user might also authorize the creation of one or more signatures along with a signature qualifier instead of a particular credential identification. This is especially useful in conjunction with short-lived credentials.

The remote service can manage the authorization in multiple ways, with different technologies and a variable number of authorization factors. This really depends on the implementation and on the policy adopted by the remote service, and MAY also be determined by the level of compliance to industry and regulatory requirements, like in the case of standards like CEN EN 419 241-1 [i.5], which defines different “sole control assurance levels”, SCAL1 and SCAL2.

For a precise description of the difference between SCAL1 and SCAL2 we refer to CEN EN 419 241-1 [i.5]. However, with regard to this specification, two aspects should be noted about SCAL2:

1. The signature activation data, used to authorize a signature, is linked to the document or the documents to be signed.
2. A two-factor authorization is needed to authorize a signature.

This specification defines two methods to obtain credential authorization:

- The credentials/authorize method is used for explicit authorization.
- The oauth2/authorize method is used to obtain an OAuth 2.0 access token that grants credential authorization.

Explicit authorization means that the remote service relies on the signature application to collect, in its own environment, authentication factors like PIN or One-Time Passwords (OTP), according to the parameters returned by the **credentials/info** method, as defined in credentials/info. This method returns the type, format and combination of required or optional authentication factors, such that the signature application can show the proper interactive controls to collect them from the user. The explicit authorization is then done by calling the credentials/authorize method.

A common type of explicit authorization is based on a static PIN - typically defined by the user - associated to the signing key when it is generated. To increase the level of assurance of user control, ensuring that only the authorized user can create a signature with a certain credential, a stronger authorization factor MAY be adopted. A dynamically generated text-based One-Time Password (OTP) is a common strong authorization mechanism. This specification directly supports the combination of various mechanisms which can be used complementary to service authorization to achieve the highest levels of assurance of the user’s sole control, and can be used to support SCAL1 and SCAL2 as defined in CEN 419 241-1 [i.5].

Biometric authentication and phone call drop are other examples of possible authorization mechanisms. As these and other authorization mechanisms require a very peculiar user interface, they can be supported by means of an OAuth 2.0-based authorization scheme.

8.2 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-creation” - used to request credential creation authorization for creating a new credential.
- “credential-deletion” - used to request credential deletion authorization for deleting a credential.
- “credential” - used to request credential authorization for creating one or more signatures with a certain credential or fulfilling the requirements of a certain signature qualifier.

A remote service can implement a single OAuth 2.0 authorization server supporting all aforementioned 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 set up 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 18: 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.

8.2.1 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	*Integer*	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 signature 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.eyJzdWIiOiI3S1lCckpBLWtCOTF5T1Rld1JZRzh5SGdzN3EtbzR
1NiIsImlhdCI6MTUzNzAxMjgwMCwianRpIjoiYjgzZmY4OWEtZWQzZi00NjgxLTgyOGQtNzE2MGI5MTNjYTcyIiwiaXNzIjoiQ1NDI
FNpZ25hdHVyZSBBcHBsaWNhdGlvbiIsImF6cCI6ImE4NzliNDE5LThmZWQtNDcyZS05Yzk3LTJmODk3NTIxODU3ZSJ9.SEwD3KGDPF
X-8IIJE7pC_RJ-0wdOVinEPTHmKKVQb6E&…
```

8.2.2 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 RFC 9126 [28] and use the request URI produced as parameter to the authorization endpoints. This section describes the authorization 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 any of the scopes described in this section. The authorization server MAY also support use of “authorization_details” as defined in RFC 9396 [27] in conjunction with the authorization detail types described in this section. This document uses the term “scope” to cover both access tokens with one of the defined scopes and access tokens with one of the corresponding authorization details.

Note 19: 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 20: 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 a 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 required 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.
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": SHALL be used to obtain an authorization code suitable for service authorization. See Service scope. • "credential-creation": SHALL be used to obtain an authorization code suitable for creating a new credential. See Credential creation scope and authorization details. • "credential-deletion": SHALL be used to obtain an authorization code suitable for deleting a credential. See Credential deletion scope and authorization details. • "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 in Credential scope and authorization details. 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". This parameter SHALL NOT be present if the `authorization_details` parameter is specified.
authorization_details	OPTIONAL	*String*	IETF RFC 9396 [27]	The details of the access request as described in IETF RFC 9396 [27]. This specification defines the following authorization details type: • "credential-creation": SHALL be used to obtain an authorization code suitable for creating a new credential. See Credential creation scope and authorization details. • "credential-deletion": SHALL be used to obtain an authorization code suitable for deleting a credential. See Credential deletion scope and authorization details. • "credential": SHALL be used to obtain an authorization code suitable for credentials authorization. See Credential scope and authorization details. The parameter is OPTIONAL. If this parameter is present, the `scope` parameter SHALL NOT be used. The authorization details SHOULD be used for the above types, since it allows a more detailed information on the documents to be signed.
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 RFC 9126 [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.

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. This parameter SHALL NOT be used if *authorization_details* is used.
clientData	OPTIONAL	*String*	Arbitrary data from the signature application. It can be used to handle a transaction identifier or other application-spe specific data that may be useful for debugging purposes. This parameter SHALL NOT be used if *authorization_details* is used. WARNING: this parameter MAY expose sensitive data to the remote service. Therefore, it SHOULD be used carefully.

See below for scope specific input parameters and authorization details.

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.

Service scope

An access token with "service" scope gives service authorization to access the CSC API.

Access tokens with certain other scopes also grants service authorization:

- An access token with “credential” scope also grants service authorization for credentials/info and signatures/signHash.
- An access token with “credential” scope in conjunction with a credential id or credential qualifier also grants service authorization for signatures/signDoc.
- An access token with “credential-creation” scope also grants service authorization for credentials/create.
- An access token with “credential-deletion” scope also grants service authorization for credentials/delete.

As a consequence, an application that has obtained an access token with one of these scopes does not need an additional access token with scope “service” in order to use these requests. This is useful to save the additional round-trip for service authorization. Using signatures/signDoc with a signature qualifier to create signatures is one example. In this case, the signing application does not need to look up the available certificates before starting the credential authorization process.

Note 21: In the course of authorizing “credential”, “credential-creation”, or “credential-deletion” scope, the authorization server authenticates the client and conveys the client identity in the respective access token (which is equivalent to the service authorization).

Examples

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
```

Credential creation scope and authorization details

It requires credential creation authorization to access the credentials/create endpoint. An oauth2 access token with either “credential-creation” scope or “credential-creation” type authorization details grants credential creation authorization.

Note 22: Authorization could be granted by the user in an Authorization Code flow. Alternatively, authorization could be granted directly to a trusted signature application with the Client Credential flow.

An authorization details object of type “credential-creation” consists of the parameters in the following table:

Parameter	Presence	Value	Description
type	REQUIRED	*String*	The authorization details type identifier. It MUST be set to either `"https://cloudsignatureconsortium.org/2025/credential-creation"` or `"credential-creation"`. It SHOULD be set to `"https://cloudsignatureconsortium.org/2025/credential-creation"`.
acr_values	OPTIONAL	*Array of String*	List of requested Authentication Context Class Reference (ACR) values as defined in RFC 9396 [27]. This parameter allows to control how users authenticate for certificate issuance. LOA values defined in [i.13] MAY be used.
credentialCreationRequest	REQUIRED	*credentialCreationRequest*	Certificate policy and possibly subject data to be authorized. The *credentialCreationRequest* JSON object is defined in the CSC data model [37]. Subject data MAY be specified in the *credentialCreationRequest*. Alternatively, the authorization server MAY collect relevant subject data and make it available in the access token or at the token introspection endpoint.

Examples

Sample Request (Credential creation authorization request with authorization details)

```
GET https://service.domain.org/oauth2/authorize?
  response_type=code&
  client_id=wallet-client-123&
  redirect_uri=https%3A%2F%2Fwallet.example.com%2Fcb&
  code_challenge=nYv8mC9kK2ltc83acc4hc9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
  code_challenge_method=S256&

authorization_details=%5B%7B%22type%22%3A%22https%3A%2F%2Fcloudsignatureconsortium.org%2F2025%2Fcreden
tial-
creation%22%2C%22acr_values%22%3A%5B%22http%3A%2F%2Feidas.europa.eu%2FLoA%2Fhigh%22%5D%2C%22credential
CreationRequest%22%3A%7B%22certificatePolicy%22%3A%220.4.0.194112.1.2%22%2C%22subjectData%22%3A%7B%22f
amily_name%22%3A%22Doe%22%2C%22given_name%22%3A%22Alice%22%2C%22birthdate%22%3A%221980-12-
01%22%7D%7D%7D%5D%0A
```

Decoded *authorization_details* parameter

```
[
  {
    "type": "https://cloudsignatureconsortium.org/2025/credential-creation",
    "acr_values": [ "http://eidas.europa.eu/LoA/high" ],
    "credentialCreationRequest": {
      "certificatePolicy": "0.4.0.194112.1.2",
      "subjectData": {
        "family_name": "Doe",
        "given_name": "Alice",
        "birthdate": "1980-12-01"
      }
    }
  }
]
```

Sample Response

```
HTTP/1.1 302 Found
Location: https://wallet.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
```

Sample Request (Credential creation authorization request)

```
GET https://service.domain.org/oauth2/authorize?
  response_type=code&
  client_id=wallet-client-123&
  redirect_uri=https%3A%2F%2Fwallet.example.com%2Fcb&
  scope=credential-creation&
  state=af0ifjsldkj&
  code_challenge=nYv8mC9kK2ltc83acc4hc9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
  code_challenge_method=S256
```

Sample Response

```
HTTP/1.1 302 Found
Location: https://wallet.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA&state=af0ifjsldkj
```

Credential deletion scope and authorization details

It requires credential deletion authorization to access the credentials/delete endpoint. An oauth2 access token with either "credential-deletion" scope or "credential-deletion" type authorization details grants credential deletion authorization.

Note 23: Authorization could be granted to a trusted signature application with the Client Credential flow.

An authorization details object of type "credential-deletion" consists of the parameters in the following table:

Parameter	Presence	Value	Description
type	REQUIRED	*String*	The authorization details type identifier. It MUST be set to either `"https://cloudsignatureconsortium.org/2025/credential-deletion"` or `"credential-deletion"`. It SHOULD be set to `"https://cloudsignatureconsortium.org/2025/credential-deletion"`.
credentialID	REQUIRED	*String*	Unique identifier of the credential to delete. Credential identifiers can be retrieved with the credentials/list endpoint.

Examples

Sample Request (Credential deletion authorization request with authorization details)

```
GET https://service.domain.org/oauth2/authorize?
  response_type=code&
  client_id=<Oauth2_client_id>&
  redirect_uri=https%3A%2F%2Fexample.com%2Fcb&
  code_challenge=nYv8mC9kK2ltc83acc4hc9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
  code_challenge_method=S256&

authorization_details=%5B%7B%22type%22%3A%22https%3A%2F%2Fcloudsignatureconsortium.org%2F2025%2Fcreden
tial-deletion%22%2C%22credentialID%22%3A%223F5A12%22%7D%5D
```

Decoded *authorization_details* parameter

```
[
  {
    "type": "https://cloudsignatureconsortium.org/2025/credential-deletion",
    "credentialID": "3F5A12"
  }
]
```

Sample Response

```
HTTP/1.1 302 Found
Location: https://wallet.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
```

Sample Request (Credential deletion authorization request)

```
GET https://service.domain.org/oauth2/authorize?
  response_type=code&
  client_id=wallet-client-123&
  redirect_uri=https%3A%2F%2Fwallet.example.com%2Fcb&
  scope=credential-deletion&
  state=af0ifjsldkj&
  code_challenge=nYv8mC9kK2ltc83acc4hc9w6ESC_rEMTJ3bww-uCHaoeK1t8U&
  code_challenge_method=S256
```

Sample Response

```
HTTP/1.1 302 Found
Location: https://wallet.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA&state=af0ifjsldkj
```

Credential scope and authorization details

It requires credential authorization to sign. An oauth2 access token with either “credential” scope or “credential” type authorization details grants credential authorization.

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.

If the `SCAL` attribute returned by the credentials/info method for the current `credentialID` is “2” and the scope is “credential”, then either the parameter `authorization_details` or `hashes` as defined below SHALL be specified.

Input parameters defined in this specification

The table below contains additional parameters to oauth2/authorize. These parameters SHALL NOT be used if the *authorization_details* parameter is used.

Parameter	Presence	Value	Description
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”. It SHALL NOT be used if *authorization_details* is present. At least one of the two values `credentialID` and `signatureQualifier` SHALL be present, both values MAY be present. 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.
signatureQualifier	REQUIRED Conditional	*signatureQualifier*	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”. It SHALL NOT be used if *authorization_details* is present. At least one of the two values `credentialID` and `signatureQualifier` SHALL be present, both values MAY be present.
numSignatures	REQUIRED Conditional	*Integer*	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”. It SHALL NOT be used if *authorization_details* is present.
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. This parameter SHALL be used only if the scope of the OAuth 2.0 authorization request is “credential”. If the `SCAL` attribute returned by the credentials/info method for the current `credentialID` is not “2” this parameter is OPTIONAL. If the `SCAL` attribute returned by the credentials/info method for the current `credentialID` is “2” and the scope is “credential”, then either this parameter or the `authorization_details` parameter SHALL be specified. 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.
hashAlgorithmOID	REQUIRED Conditional	*String*	String containing the OID of the hash algorithm used to generate the hash values.
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].

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. The *authorization_details* parameter of oauth2/authorize SHALL NOT be used if the custom parameters above are used. The authorization details object is the union of the following:

- *signatureCreationApproval* (see the CSC data model [37])
- The parameters in the table below

Parameter	Presence	Value	Description
type	REQUIRED	*String*	The authorization details type identifier. It MUST be set to either `"https://cloudsignatureconsortium.org/2025/credential"` or `"credential"`. It SHOULD be set to `"https://cloudsignatureconsortium.org/2025/credential"`.
locations	OPTIONAL	*Array of String*	The locations of remote signing service providers as defined in RFC 9396 [i.4]. This designates the locations of the API to which access is authorized.

Examples

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 documentDigests in 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%3A%22credential%22%2C%22signatureQualifier%22%3A%22eu_eidas_qes
%22%2C%22numSignatures%22%3A1%2C%22documentDigests%22%3A%5B%7B%22label%22%3A%22Example+Contract%22%2C%
22hash%22%3A%22sTOgwOm%2B474gFj0q0x1iSNspKqbcse4IeiqlDg%2FHWuI%3D%22%2C%22signed_props%22%3A+%5B%7B%22
commitment-type-
indication%22%3A+%221.2.840.113549.1.9.16.6.5%22%7D%5D%2C%22circumstantialData%22%3A+%22signingTime%3D
20241129170916%2B00%2700%27%22%7D%5D%2C%22hashAlgorithmOID%22%3A%222.16.840.1.101.3.4.2.1%22%7D%5D
```

Decoded `authorization_details` parameter

```
[
   {
      "type":"credential",
      "signatureQualifier":"eu_eidas_qes",
      "numSignatures":1,
      "documentDigests":[
         {
            "label":"Example Contract",
            "hash":"sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
            "signed_props": [
              {
                "commitment-type-indication": "1.2.840.113549.1.9.16.6.5"
              }
            ],
            "circumstantialData": "signingTime=20241129170916+00'00'"
         }
      ],
      "hashAlgorithmOID":"2.16.840.1.101.3.4.2.1"
   }
]
```

Sample Response (Credential authorization with documentDigests in authorization_details)

```
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%3A%22credential%22%2C%22signatureQualifier%22%3A%22eu_eidas_qes
%22%2C%22numSignatures%22%3A2%2C%22documentDigests%22%3A%5B%7B%22hash%22%3A%22sTOgwOm%2B474gFj0q0x1iSN
spKqbcse4IeiqlDg%2FHWuI%3D%22%2C%22label%22%3A%22Example%20Contract%22%7D%2C%7B%22hash%22%3A%22HZQzZmM
AIWekfGH0%2FZKW1nsdt0xg3H6bZYztgsMTLw0%3D%22%2C%22label%22%3A%22Example%20Terms%20of%20Service%22%7D%5
D%2C%22hashAlgorithmOID%22%3A%222.16.840.1.101.3.4.2.1%22%7D%5D
```

Decoded `authorization_details` parameter

```
[
   {
      "type":"credential",
      "signatureQualifier":"eu_eidas_qes",
      "numSignatures":2,
      "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
```

8.2.3 oauth2/pushed_authorize

This is the OAuth 2.0 pushed authorization endpoint as defined in RFC 9126 [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 mechanism 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 an 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_eidas_qes%22,%
22documentDigests%22:%5B%7B%22hash%22:%22sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=%22,%22label%22:%
22Example%20Contract%22%7D,%7B%22hash%22:%22HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=%22,%22label%2
2:%22Example%20Terms%20of%20Service%22%7D%5D,%22hashAlgorithmOID%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
```

8.2.4 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].

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 24: **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 25: 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".
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 set up 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 set up 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.
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	*Integer*	The lifetime in seconds of the service access token. If omitted, the default expiration time is 3600 sec. (1 hour).

Note 26: 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
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
}
```

8.2.5 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 27: **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.

8.2.5.1 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 28: 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.
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

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

token=_TiHRG-bA-H3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw&
token_type_hint=refresh_token&
client_id=<OAuth2_client_id>&
client_secret=<OAuth2_client_secret>&
clientData=12345678
```

cURL Example

```
curl -i -X POST
     -H "Content-Type: application/x-www-form-urlencoded"
     -d 'token=_TiHRG-bA-H3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw&
         token_type_hint=refresh_token&
         client_id=<OAuth2_client_id>&
         client_secret=<OAuth2_client_secret>&
         clientData=12345678'
     https://www.domain.org/oauth2/revoke
```

Sample Response

```
HTTP/1.1 204 No Content
```

8.3 Authentication and authorization for electronic seals

8.3.1 Introduction

The eIDAS regulation (Regulation (EU) No 910/2014 [i.1]) defines two basic concepts: an electronic signature, created by a natural person used to sign the content of a document, and an electronic seal based on a certificate of a legal person used to prove the origin and integrity of the document. From a mere technical point of view, both electronic signatures and electronic seals are digital signatures. However, the usage of the CSC API in order to create an electronic signature or an electronic seal can be different depending on the above-cited legal context. The present section discusses the usage of the CSC API for creating electronic seals, which in the context of the present document are digital signatures created by using a certificate issued to a legal person. This ensures the integrity and origin of the document, without necessarily committing to the content.

Note 29: This definition is not limited to the legal definition of electronic seals in Regulation (EU) No 910/2014 [i.1].

In many cases, electronic seals are created in automated processes and often a large number of documents are to be sealed in one session. In the present document, there are two different possible authorizations. The first one is the authorization to get access to the API, and the second one is the authorization to use the signing credential for the seal/signature creation. The following section describe how these authorizations can be done with the purpose of creating an electronic seal.

8.3.2 Service authorization and authentication for electronic seals

Several methods allow access to the CSC API without the need for regular human interaction, which would not be very practical in the case of sealing a large number of documents.

8.3.2.1 Login / password

HTTP basic or HTTP digest authentication can be used to provide access to the API. The login and password MAY be linked to the signature application or to the certificate owner.

8.3.2.2 OAuth with client credentials grant

The usage of OAuth 2.0 with client credential grant allows granting access to the signing application. It does not convey any user specific identifier. This authenticates the client, any user specific information is indicated within the respective CSC API call or provided implicitly or separately.

8.3.2.3 Mutual TLS

The signing server can be configured to use TLS connections, requiring clients that attempt to connect to get authenticated. A client SHALL use a client certificate in order to authenticate. The client certificate SHALL contain information allowing the signing server to authenticate the client application/user. The signing server MAY be configured to accept TLS connections only from a limited group of allowed clients.

> An example can just be a scenario where the usage of the sealing credentials is limited only to successfully authenticated TLS connections using client certificates authentication connections. This method does not create any token. In an additional use case, the remote signing service provider has a specific end point (outside the CSC specification) which can be accessed via TLS authentication + API key + secret which creates an Access Token. And this access token is used later on to access the API. In case of seals, no extra authorization is used to access the private key. Used with short-lived credentials.

Note 30: By defining an empty set of authentication object types, the RSSP can decide to not need any more actions.

> In addition to the mutual TLS, a token can be created based on login / password + OTP by a non CSC end point, and is then used for signing together with a PIN. This can be used with long-term certificates

8.3.3 Credential authorization for electronic seals

The credential authorization allows the usage of a specific key. There are three possible strategies, to avoid human interaction for each signature.

- The first is the usage of an authorization means that can be fully automized, for example the usage of a PIN.
- It is also possible to not require any additional actions, if the access token is already sufficient.
- The third one, consists in creating a SAD for a high but limited number of signatures. Since the creation of the SAD is an operation which is not repeated very often, it can be created in a non-fully automated process. This allows a more complex authorization, and to be more precise in what this authorization includes.

9 Creating a remote signature

Remote signature services allow generating digital signatures remotely by means of an RSCD operated as a service. An RSSP is an organization that manages the RSCD on behalf of the signers.

In general, each time a remote signature is required, a strong authentication mechanism SHOULD be invoked. Strong authentication requiring the user to authorize to the signature application multiple times in a rapid sequence using authorization mechanisms like OTP can be cumbersome. In order to improve the signer's experience, the strong authentication MAY be allowed to occur only once per signing session (for example with a single OTP) covering multiple signatures.

The current specification supports the following three use cases:

1. The remote signature of a single hash;
2. The remote signature of multiple hashes passed in a single signature operation;
3. The remote signature of multiple hashes passed across multiple signature operations occurring within a single signing session.

A RSSP SHALL support at least case 1, with credentials authorization occurring every time a signature is created.

The RSSP decides whether to support multi-signature transactions (use cases 2 and 3) or not. In some cases, regulatory or security requirements may forbid multi-signature transactions. The *multisign* output value of the **credentials/info** method, as defined in credentials/info, provides information if multi-signature transactions are supported by a specific credential or not.

A multi-signature transaction can be created by invoking the **signatures/signHash** method, as defined in signatures/signHash, and submitting multiple hash values in one run (use case 2, suitable for "batch signing" of multiple documents) or by invoking **signatures/signHash** multiple times (use case 3, suitable for creating multiple signatures from a single user in a PDF document). In both cases, the authorization mechanism adopted by the signature application SHALL explicitly specify the total number of signatures to be authorized and the remote signing service SHALL prevent signature applications from creating more signatures than authorized.

See Interaction among elements and components to understand the workflows supported in this specification and the sequence of API calls to be invoked to create the supported types of remote signatures.

10 Error handling

Errors are returned by the remote service using standard HTTP status code syntax. Additional information is included in the body of the response from an API request using JSON.

The HTTP protocol defines a list of standard status codes that are referenced in this specification to help the signature application deal with these responses accordingly. For the events described in Table 2, the remote service SHALL support the corresponding HTTP status codes.

Table 2 – Supported HTTP Status Codes

Standard Status Code	Description
200 OK	Response to a successful API method request.
204 No Content	Response to a successful API method request in case no content is returned.
302 Found	Response used to redirect the user to an OAuth 2.0 authorization endpoint.
400 Bad Request	Returned due to unsupported, invalid or missing required parameters.
401 Unauthorized	Returned when a bad or expired authorization token is used.
429 Too Many Requests	Returned when a request is rejected due to rate limiting.
500 Internal Server Error	Returned when the server encounters an unexpected condition.
501 Not Implemented	Returned when an unimplemented method is requested.
503 Service Unavailable	Returned when the server is currently unable to handle the request due to temporary overloading or maintenance conditions.

Status codes 429 and 50x are applicable to the remote service overall and are not specific to any API methods. For this reason, they are not mentioned in the error tables for each method specifically.

10.1 Error messages

Just as an HTML error page shows a useful error message to a visitor, the remote service implementing the API described in this specification SHALL provide a useful error message in case something goes wrong. When an error is detected, the remote service SHALL return the corresponding HTTP status code and SHALL return the information on the error in the body of the HTTP response using the “application/json” media type, as defined by RFC 4627 [5]. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings as shown in the following example:

```
HTTP/1.1 400 Bad Request
Date: Mon, 03 Dec 2018 12:00:00 GMT
Content-Type: application/json;charset=utf-8
Content-Length: ...
{
   "error": "invalid_request",
   "error_description": "The access token is not valid"
}
```

The *error_description* parameter is OPTIONAL but highly RECOMMENDED to provide a human-readable text string containing additional information to assist the user in understanding the error that occurred.

The remote service can also define custom error messages by using messages that are not defined in this specification.

The following table contains definitions for errors that are common to more than one API methods. Therefore, they’re presented only once in this section instead of being repeated for all API methods.

Table 3 – Predefined common Error Messages

Error	Error Description
invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
unauthorized_client	The client is not authorized to use this method.
access_denied	The user, authorization server or remote service denied the request.
unsupported_response_type	The authorization server does not support obtaining an authorization code using this method.
invalid_scope	The requested scope is invalid, unknown, or malformed.
server_error	The authorization server encountered an unexpected condition that prevented it from fulfilling the request.
temporarily_unavailable	The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server.
expired_token	The access or refresh token is expired or has been revoked.
invalid_token	The token provided is not a valid OAuth access or refresh token.

11 The remote service APIs

In order to simplify the navigation of this specification, the following table summarizes all the API methods defined in the present specification. The **info** method, as defined in info, SHALL be implemented. All other methods are OPTIONAL.

Table 4 – API methods summary

API Method	Description
info	Returns information on the remote service and the list of API methods it has implemented.
auth/login	Authorize the remote service with HTTP Basic or Digest authentication.
auth/revoke	Revoke the service access token or refresh token.
credentials/list	Returns the list of credentials associated to a user.
credentials/info	Returns information on a signing credential, its associated certificate and a description of the supported authorization mechanism.
credentials/authorize	Authorize the access to the credential for signing.
credentials/extendTransaction	Extend the validity of a multi-signature transaction.
credentials/sendOTP	Start the online OTP mechanism associated to a credential.
signatures/signHash	Calculate a raw digital signature from one or more hash values.
signatures/signDoc	Creates one or more AdES signatures for documents or document digests.
signatures/timestamp	Return a time stamp token for the input hash value.
oauth2/authorize*	Initiate an OAuth 2.0 authorization flow.
oauth2/token*	Obtain an OAuth 2.0 access token or refresh token.
oauth2/revoke*	Revoke an OAuth 2.0 access token or refresh token.

Note 31: Although **oauth2/authorize** , **oauth2/token**, **oauth2/pushed_authorize**, and **oauth2/revoke**, as defined in OAuth 2.0 Authorization, do not specify regular CSC API methods but rather endpoints managed by the OAuth 2.0 authorization server, they're listed in Table 4 to provide a complete overview of the endpoints that can be supported by a remote service conforming to this specification.

11.1 info

Description

Returns information about the remote service and the list of the API methods it supports. This method SHALL be implemented by any remote service conforming to this specification.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
lang	OPTIONAL	*String*	Request a preferred language of the response to the remote service, specified according to RFC 5646 [9]. If present, the remote service SHALL provide language-specif ic responses using the specified language. If the specified language is not supported then it SHALL provide these responses in the language as specified in the *lang* output parameter.

Output

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

Attribute	Presence	Value	Description
specs	REQUIRED	*String*	The version of this specification implemented by the provider. The format of the string is Major.Minor.x.y , where Major is a number equivalent to the API version (e.g. 2 for API v2) and Minor is a number identifying the version update, while x and y are subversion numbers. The value corresponding to this specification is “2.2.0.0”.
name	REQUIRED	*String*	The commercial name of the remote service. The maximum size of the string is 255 characters.
logo	REQUIRED	*String*	The URI of the image file containing the logo of the remote service which SHALL be published online. The image SHALL be in either JPEG or PNG format and not larger than 256x256 pixels.
region	REQUIRED	*String*	The ISO 3166-1 [22] Alpha-2 code of the Country where the remote service provider is established (e.g. ES for Spain).
lang	REQUIRED	*String*	The language used in the responses, specified according to RFC 5646 [9].
description	REQUIRED	*String*	A free form description of the remote service in the *lang* language. The maximum size of the string is 255 characters.
authType	REQUIRED	*Array of String*	One or more values corresponding to the service authorization mechanisms supported by the remote service to authorize the access to the API: • “external”: in case the authorization is managed externally (e.g. using a VPN or a private LAN). • “TLS”: in case the authorization is provided by means of TLS client certificate authentication. • “basic”: in case of HTTP Basic Authentication. • “digest”: in case of HTTP Digest Authentication. • “oauth2code”: in case of OAuth 2.0 with authorization code flow. • “oauth2client”: in case of OAuth 2.0 with client credentials flow.
oauth2Servers	REQUIRED Conditional	*Array of OAuth2Server*	List of OAuth 2.0 authorization servers supported by the remote service for service authorization and/or credential authorization. If • the *authType* parameter contains “oauth2code” or “oauth2client” or • the remote service supports the value “oauth2code” for the *auth/mode* parameter returned by **credentials/info** (as specified in credentials/info), exactly one of the parameters “oauth2”, “oauth2Issuer”, or “oauth2Servers” SHALL be present.
oauth2	REQUIRED Conditional	*String*	The base URI of the OAuth 2.0 authorization server endpoint supported by the remote service for service authorization and/or credential authorization. If • the *authType* parameter contains “oauth2code” or “oauth2client” or • the remote service supports the value “oauth2code” for the *auth/mode* parameter returned by **credentials/info** (as specified in credentials/info), exactly one of the parameters “oauth2”, “oauth2Issuer”, or “oauth2Servers” SHALL be present. This URI SHALL be combined with the path components described in OAuth 2.0 Authorization in order to build the actual endpoint URLs.
oauth2Issuer	REQUIRED Conditional	*String*	The issuer URL of the OAuth 2.0 authorization server as defined in IETF RFC 8414 [23] supported by the remote service for service authorization and/or credential authorization. If • the *authType* parameter contains “oauth2code” or “oauth2client” or • the remote service supports the value “oauth2code” for the *auth/mode* parameter returned by **credentials/info** (as specified

Attribute	Presence	Value	Description
			in credentials/info), exactly one of the parameters "oauth2", "oauth2Issuer", or "oauth2Servers" SHALL be present. The OAuth endpoint URLs are obtained from the OAuth Server metadata as described in IETF RFC 8414 [23].
supportsRar	OPTIONAL	*Boolean*	If this parameter is present and either "oauth2" or "oauth2issuer" is present, this parameters SHALL be "true" if the authorization server supports "authorization_details" as documented in oauth2/authorize and "false" if it does not. If this parameter is not present and either "oauth2" or "oauth2issuer" is present, the default value is "false". This parameter SHALL NOT be present if "oauth2Servers" is present.
supportedHashTypes	REQUIRED	*Array of String*	List of values supported for `hashType` as defined in oauth2/authorize. Details on the `hashType` required to authorize different combinations of signing method, parameters, and assurance level SHOULD be specified in the signature creation policy.
asynchronousOperationMode	OPTIONAL	*Boolean*	This parameter shall be "true" if the remote signing server supports also asynchronous signature mechanism. The default value is "false". An omitted parameter or the value "false" indicates that the remote signing server manages signature requests only in synchronous operation mode.
methods	REQUIRED	*Array of String*	The list of names of all the API methods described in this specification that are implemented and supported by the remote service.
validationInfo	OPTIONAL	*Boolean*	This parameter SHALL be "true" if the remote signing server supports the "validationInfo" response parameter of the method **signatures/signDoc** in not mandatory cases. An omitted parameter or the value "false" indicates that the remote signing server does not support "validationInfo" in those cases.
signAlgorithms	REQUIRED	*JSON Object*	Object including one or more signature algorithms supported by the RSSP.
documentTypes	REQUIRED Conditional	*Array of String*	A list of document types supported by signatures/signDoc. SHALL be present if signatures/signDoc is supported and the RSSP supports input formats other than `sod`.
signature_formats	REQUIRED	*JSON Object*	Object including one or more signature formats supported by the RSSP.
conformance_levels	REQUIRED	*Array of String*	The list of names of all signature conformance levels supported by the RSSP as defined in the Input parameter table in signatures/signDoc. If the RSSP does not support signatures/signDoc, the array SHOULD be empty.

An `OAuth2Server` object is a JSON object composed of the following attributes:

Parameter	Presence	Value	Description
label	OPTIONAL	*String*	Human-readable description of the authorization server.
baseUri	REQUIRED Conditional	*String*	See definition of *oauth2* above (Output). Either *baseUri* or *issuerIdentifier* SHALL be present. *baseUri* and *issuerIdentifier* MUST NOT both be present.
issuerIdentifier	REQUIRED Conditional	*String*	See definition of *oauth2Issuer* above (Output). Either *baseUri* or *issuerIdentifier* SHALL be present. *baseUri* and *issuerIdentifier* MUST NOT both be present.
authType	REQUIRED	*Array of String*	One or more values corresponding to the service authorization mechanisms supported by the remote service to authorize the access to the API: • "oauth2code": in case of OAuth 2.0 with authorization code flow. • "oauth2client": in case of OAuth 2.0 with client credentials flow.
supportsRar	OPTIONAL	*Boolean*	If present, this parameters SHALL be "true" if the authorization server supports "authorization_details" as documented in oauth2/authorize and "false" if it does not. The default value is "false".

The `signAlgorithms` is a JSON Object composed of the following parameters:

- `algos`

- `algoParams`

specified according to the following table.

Parameter	Presence	Value	Description
algos	REQUIRED	*Array of String*	The list of signature algorithms supported by the RSSP as defined in the Input parameter table in signatures/signHash. The supported signature algorithms SHOULD follow the recommendations of ETSI TS 119 312 [21] and SHALL be expressed as defined in Expressing algorithms clause.
algoParams	REQUIRED Conditional	*Array of String*	The list of eventual signature parameters as defined in the Input parameter table in signatures/signHash.

The `signature_formats` is a JSON Object composed by the following parameters:

- `formats`
- `envelope_properties`
- `allowMix`

specified according to the following table.

Parameter	Presence	Value	Description
formats	REQUIRED	*Array of String*	The list of signature formats supported by the RSSP as defined in the Input parameter table in signatures/signDoc. If the RSSP does not support signatures/signDoc, the array SHOULD be empty.
envelope_properties	REQUIRED Conditional	*Array of Array of String*	The list of the properties concerning the signed envelope, whose possible values depend on the value of the *formats* parameter entries, as defined in the Input parameter table in signatures/signDoc. The number of arrays included in the *envelope_properties* array SHALL equal the number of entries in the *formats* array. The values included in the array at position i of the *envelope_properties* array SHALL refer to the signature format value included at position i of the *formats* array. An empty array at the position i of the *envelope_properties* array indicates that the RSSP supports the default signed envelope property for the signature format specified at the position i of the *formats* array, as defined in the Input parameter table in signatures/signDoc.
allowMix	OPTIONAL	*boolean*	If true, signatures/signDoc SHALL support multiple combinations of format and envelope types in a single call. If false, signatures/signDoc MAY fail if a call includes more than one format and envelope type. Default is "false".

Note 32: **info** is a mandatory API method, so it MAY be excluded from the list of API method names returned by the *methods* parameter.
The endpoints **oauth2/authorize** , **oauth2/token**, **oauth2/pushed_authorize** and **oauth2/revoke**, as defined in OAuth 2.0 Authorization, do not specify regular API methods but rather endpoints managed by the OAuth 2.0 authorization server, therefore they MAY be excluded from the list of API method names returned by the *methods* parameter.

Examples

Sample Request

```
POST /csc/v2/info HTTP/1.1
Host: service.domain.org
Content-Type: application/json

{}
```

cURL Example

```
curl -i -X POST
     -H "Content-Type: application/json"
     -d '{}'
     https://service.domain.org/csc/v2/info
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "specs": "2.2.0.0",
   "name": "ACME Trust Services",
   "logo": "https://service.domain.org/images/logo.png",
   "region": "IT",
   "lang": "en-US",
   "description": "An efficient remote signature service",
   "authType": ["basic", "oauth2code"],
   "oauth2": "https://www.domain.org/",
   "methods": ["auth/login", "auth/revoke", "credentials/list",
      "credentials/info", "credentials/authorize",
      "credentials/sendOTP",
      "signatures/signHash", "signatures/signDoc"],
   "signAlgorithms":
   {
    "algos": ["1.2.840.10045.4.3.2", "1.2.840.113549.1.1.1", "1.2.840.113549.1.1.10"]
   },
   "documentTypes": [ "sod", "sfd" ],
   "signature_formats":
   {
    "formats": ["C", "X", "P"],
    "envelope_properties": [["Detached", "Attached", "Parallel"],
                            ["Enveloped", "Enveloping", "Detached"],
                            ["Certification", "Revision"]],
    "allowMix": true
   },
   "conformance_levels": ["AdES-B-B", "AdES-B-T"]
}
```

11.2 auth/login

Description

Obtain an access token for service authorization from the remote service using HTTP Basic Authentication or HTTP Digest authentication, as defined in RFC 7235 [2], using the *userID* and *password* assigned to the user. These authentication factors SHALL be passed directly in the HTTP header as an authorization grant to obtain a service access token to use for the subsequent API requests within the same session.

The OPTIONAL *rememberMe* parameter can be used, under the control of the user, in order to extend a successful authentication for subsequent sessions and to avoid the user to authenticate again within a predefined period of time. In this case, a refresh token will be obtained, which can be used in the *refresh_token* parameter in subsequent calls as an alternative to passing *userID* and *password* again for obtaining a new access token.

Note 33: The RECOMMENDED mechanism for service authorization is OAuth 2.0 (see OAuth 2.0 Authorization). HTTP Basic Authentication is an unsafe mechanism, and therefore it SHOULD NOT be used, especially by signature application running as a service. It should only be used when there is a high degree of trust between the user and the signature application and when other authorization types like OAuth 2.0 are not available. This method may also be deprecated in future releases of this specification.

Input

The *userID* and *password* strings SHALL be encoded as defined in RFC 7235 [2] and provided in the HTTP Authorization header. If available, a refresh token MAY be alternatively used to re-authenticate the user after an access token has expired. This method allows the following parameters:

Parameter	Presence	Value	Description
refresh_token	REQUIRED Conditional	*String*	The long-lived refresh token returned from a previous call to this method with HTTP Basic Authentication. This MAY be used as an alternative to the Authorization header to reauthenticate the user according to the method described in RFC 6749 [11] par. 1.5. In such case the encoded *userId* and *password* SHALL not be provided in the HTTP Authorization header. NOTE: This refresh token MAY not be compatible with refresh tokens obtained by means of OAuth 2.0 authorization (see **oauth2/token** in oauth2/token).
rememberMe	OPTIONAL	*Boolean*	A boolean value typically corresponding to an option that the user may activate during the authentication phase to "stay signed in" and maintain a valid authentication across multiple sessions: • "true": if the remote service supports user reauthentication, a *refresh_token* will be returned and the signature application may use it on a subsequent call to this method instead of passing an Authorization header. • "false": a *refresh_token* will not be returned. If the parameter is omitted, it will default to "false". This mechanism is based on the method described in RFC 6749 [11] section 1.5.
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:

Attribute	Presence	Value	Description
access_token	REQUIRED	*String*	The short-lived service access token used to authenticate the subsequent API requests within the same session. This token SHALL be used as the value of the "Authorization: Bearer" in the HTTP header of the API requests. When receiving an API call with an expired token, the remote service SHALL return an error and require a new auth/login request.
refresh_token	OPTIONAL Conditional	*String*	The long-lived refresh token used to re-authenticate the user on the subsequent session. The value is returned if the *rememberMe* parameter in the request is "true" and the remote service supports user reauthentication. This mechanism is based on the method described in RFC 6749 [11] par. 1.5. NOTE: This *refresh_token* MAY not be compatible with refresh tokens obtained by means of OAuth 2.0 authorization.
expires_in	OPTIONAL	*Integer*	The lifetime in seconds of the service access token. If omitted, the default expiration time is 3600 (1 hour).

Note 34: Access tokens and refresh tokens are credentials used to access protected resources. These tokens are strings representing a service authorization issued to the client. The strings MAY represent specific authorization criteria, but they SHOULD be opaque to the client.

Note 35: An existing refresh token MAY be automatically revoked if the user to whom it was issued performs a new service authorization with the *rememberMe* parameter set to "true". It is up to the remote service to support a single or multiple refresh tokens per user.

Note 36: The lifetime of the *refresh_token* is determined by the RSSP.

Error Case	Status Code	Error	Error Description
The authorization header does not match the basic HTTP authentication pattern ("Basic [base64]") - if refresh token is not present	401 (unauthorized)	invalid_request	Malformed authentication parameter.
Decoded credentials are not in the form "username:password"	400 (bad request)	invalid_request	Malformed username-password.
Invalid refresh_token parameter format	400 (bad request)	invalid_request	Invalid string parameter: refresh_token
Invalid refresh_token value	400 (bad request)	invalid_request	Invalid refresh_token
Authentication error – login failed	400 (bad request)	authentication_error	An error occurred during authentication process

Examples

Sample Request

```
POST /csc/v2/auth/login HTTP/1.1
Host: service.domain.org
Authorization: Basic Y2xpZW50X2lkOmNsaWVudF9zZWNyZXQ=
Content-Type: application/json

{
   "rememberMe": true
}
```

cURL Example

```
curl -i -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Basic Y2xpZW50X2lkOmNsaWVudF9zZWNyZXQ="
     -d '{"rememberMe": true}'
     https://service.domain.org/csc/v2/auth/login
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
   "access_token": "4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA",
   "refresh_token":
   "_TiHRG-bA-H3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
   "expires_in": 3600
}
```

11.3 auth/revoke

Description

Revoke a service access token or refresh token that was obtained from the remote service or an associated authorization server. The revocation process is aligned with the OAuth 2.0 revocation mechanism described in RFC 7009 [13] and can be applied to both tokens issued through calls to remote service methods (e.g. **auth/login** as defined in auth/login) and tokens issued as a result of an OAuth 2.0 flow (e.g. **oauth2/token** as defined in oauth2/token). 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 SHALL also invalidate any existing 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 token type before reaching the usage limit.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
token	REQUIRED	*String*	The token that the signature application wants to get revoked.
token_type_hint	OPTIONAL	*String* access_token | refresh_token	An OPTIONAL hint about the type of the token submitted for revocation. If the parameter is omitted, the remote service SHOULD try to identify the token across all the available tokens.
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
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	Malformed authorization header.
Missing or not String "token" parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter token
"token_hint" parameter present, not equal to "access_token" nor "refresh_token"	400 (bad request)	invalid_request	Invalid string parameter token_type_hint
Invalid access_token or refresh_token	400 (bad request)	invalid_request	Invalid string parameter token

Examples

Sample Request

```
POST /csc/v2/auth/revoke HTTP/1.1
Host: service.domain.org
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json

{
   "token": "_TiHRG-bA-H3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
   "token_type_hint": "refresh_token",
   "clientData": "12345678"
}
```

cURL Example

```
curl -i -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{"token": "_TiHRG-bA-H3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
          "token_type_hint": "refresh_token",
          "clientData": "12345678"}'
     https://service.domain.org/csc/v2/auth/revoke
```

Sample Response

```
HTTP/1.1 204 No Content
```

11.4 credentials/create

Description

Creates a new signing credential. If requested, it can also return the new signing certificate, the whole associated certificate chain, additional information about the signing certificate and/or information about the authorization mechanism required to authorize the access to the credential for remote signing. This method requires service and Credential creation authorization.

The remote service MUST collect the subject data needed for the certificate signing request. When Rich Authorization Requests (RAR) are supported, the subject data MAY be provided by the signature application in the authorization request. Alternatively, the subject data MAY be collected by the authorization server and made available to the remote service. The following is a non-exhaustive list of examples of ways that the authorization server can make the subject data available to the remote service:

- In the authorization details of the access token (see Credential creation scope and authorization details),
- as custom claims on the access token [i.15], or
- at a token introspection endpoint [i.16].

The remote service MAY ignore or overwrite all or some of the subject data provided by the signature application.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
credentialCreationRequest	REQUIRED	*credentialCreationRequest*	*credentialCreationRequest* object as defined in the CSC data model [37]. The new credential SHALL be issued in accordance with the *certificatePolicy* given in the *credentialCreationRequest*. The certificate for the new credential SHOULD contain the *subjectData* given in the *credentialCreationRequest*. The *subjectDN* of the certificate for the new credential SHOULD be based on the *subjectData* entries given in the *credentialCreationRequest*. The RSSP SHOULD specify the format of *subjectData*. `</td>`
credentialInfo	OPTIONAL	*Boolean*	Request to return the main information included in the public key certificate and the public key certificate itself or the certificate chain associated to the credentials. The default value is “false”, so if the parameter is omitted then the information will not be returned.
certificates	OPTIONAL	*String* none | single | chain	Specifies which certificates from the certificate chain SHALL be returned in *certs/certificates*. • “none”: No certificate SHALL be returned. • “single”: Only the end entity certificate SHALL be returned. • “chain”: The full certificate chain SHALL be returned. The default value is “single”, so if the parameter is omitted then the method will only return the end entity certificate. This parameter MAY be specified only if the parameter *credentialInfo* is “true”. If the parameter *credentialInfo* is not “true” and this parameter is specified its value SHALL be ignored. If the certificate chain is requested, it is returned as an array of base64-encoded X.509 certificates with the end entity certificate being the first entry.
certInfo	OPTIONAL	*Boolean*	Request to return various parameters containing information from the end entity certificate(s). This is useful in case the signature application wants to avoid decoding the certificate(s). The default value is “false”, so if the parameter is omitted then the information will not be returned. This parameter MAY be specified only if the parameter *credentialInfo* is “true”. If the parameter *credentialInfo* is not “true” and this parameter is specified its value SHALL be ignored.
authData	OPTIONAL	*Array of authentication objects*	The authentication objects defining the authentication mechanisms and protocol that a client application SHALL implement for subsequent credential usage (see credentials/authorize).

Note 37: If the RSSP accepts *subjectData* from external parties (e.g. the signature application), the RSSP should provide those external parties with information about the type and format of the supported *subjectData* attributes. The RSSP may, for instance, accept a subset of the entries from an id token as specified in [i.17].

Output

This method returns a *credentialInfo* object using the “application/json” format (see credentials/list).

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
Missing or invalid “credentialCreationRequest” parameter	400 (bad request)	invalid_request	Missing (or invalid type) parameter “credentialCreationRequest”
Missing or not String “certificatePolicy” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter certificatePolicy
Invalid “certificatePolicy” parameter	400 (bad request)	invalid_request	Invalid parameter certificatePolicy
Invalid “subjectData” parameter	400 (bad request)	invalid_request	Invalid parameter subjectData
Invalid “certificates” parameter	400 (bad request)	invalid_request	Invalid parameter “certificates”
Invalid “certInfo” parameter	400 (bad request)	invalid_request	Invalid parameter “certInfo”
Invalid “authData” parameter	400 (bad request)	invalid_request	Invalid parameter “authData”

Examples

Sample Request

```
POST /csc/v2/credentials/create HTTP/1.1
Host: service.domain.org
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQ23ndFhKxF9P2_TiHRG-bA
Content-Type: application/json

{
  "credentialCreationRequest": {
    "certificatePolicy": "0.4.0.194112.1.2",
    "subjectData": {
      "family_name": "Doe",
      "given_name": "Alice",
      "birthdate": "1980-12-01"
    }
  },
  "credentialInfo": true,
  "certificates": "chain"
}
```

cURL Example

```
curl -i -X POST \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQ23ndFhKxF9P2_TiHRG-bA" \
  -d '{
        "credentialCreationRequest": {
          "certificatePolicy": "0.4.0.194112.1.2",
          "subjectData": {
            "family_name": "Doe",
            "given_name": "Alice",
            "birthdate": "1980-12-01"
          }
        },
        "credentialInfo": true,
        "certificates": "chain"
      }' \
  https://service.domain.org/csc/v2/credentials/create
```

Sample Response

```
HTTP/1.1 201 Created
Content-Type: application/json

{
   "credentialID": "GX0112348",
   "signatureQualifier": "eu_eidas_qes",
   "key": {
      "status": "enabled",
      "algo": [
         "1.2.840.113549.1.1.1"
      ],
      "len":2048
   },
   "cert":{
      "status":"valid",
      "certificates":[
         "<base64-encoded_X.509_end_entity_certificate>",
         "<base64-encoded_X.509_intermediate_CA_certificate>",
         "<base64-encoded_X.509_root_CA_certificate>"
      ],
      "issuerDN":"<X.500_issuer_DN_printable_string>",
      "serialNumber":"5AAC41CD8FA22B953640",
      "subjectDN":"<X.500_subject_DN_printable_string>",
      "validFrom":"20180101100000Z",
      "validTo":"20190101095959Z"
   },
   "auth": {
      "mode": "oauth2code"
    },
    "SCAL": 2,
    "multisign":5,
    "lang":"en-US"
}
```

11.5 credentials/delete

Description

Delete an existing signing credential. This method requires service and Credential deletion authorization.

In applications where credential deletion authorization does not require explicit authorization from the credential owner, the RSSP SHOULD inform the credential owner when a credential is deleted.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
credentialDeletionRequest	REQUIRED	*credentialDeletionRequest*	Identity of the credential to delete as well as possible request to revoke the certificate. Defined in the CSC data model [37].

Output

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

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
Missing or not String “credentialID” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter “credentialID”
Invalid “credentialID” parameter	400 (bad request)	invalid_request	Invalid parameter “credentialID”
The “credentialID” parameters in *credentialDeletionRequest* disagrees with the “credentialID” in the authorization details	400 (bad request)	invalid_request	Contradicting “credentialID” parameters

Examples

Sample Request

/csc/v2/credentials/delete — DM-compliant request (simple delete)

```
POST /csc/v2/credentials/delete HTTP/1.1
Host: service.domain.org
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json

{
  "credentialDeletionRequest": {
    "credentialID": "cred_8f2a3ff"
  }
}
```

cURL Example

```
curl -i -X POST \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA" \
  -d '{ "credentialDeletionRequest":{"credentialID":"cred_8f2a3ff"} }' \
  https://service.domain.org/csc/v2/credentials/delete
```

Sample Response

```
HTTP/1.1 204 No Content
```

Sample Request

/csc/v2/credentials/delete — DM-compliant request (delete + revoke with reason)

```
POST /csc/v2/credentials/delete HTTP/1.1
Host: service.domain.org
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json

{
  "credentialDeletionRequest": {
    "credentialID": "cred_8f2a3ff",
    "revoke": true,
    "revocationReason": 1
  }
}
```

cURL Example

```
curl -i -X POST \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA" \
  -d '{"credentialDeletionRequest":
{"credentialID":"cred_8f2a3ff","revoke":true,"revocationReason":1}}' \
  https://service.domain.org/csc/v2/credentials/delete
```

Sample Response

```
HTTP/1.1 204 No Content
```

Error Response (Malformed Authorization header)

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": "invalid_request",
  "error_description": "Malformed authorization header."
}
```

Error Response (Missing / non-string credentialID)

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": "invalid_request",
  "error_description": "Missing (or invalid type) string parameter \"credentialID\""
}
```

Error Response (Invalid credentialID)

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": "invalid_request",
  "error_description": "Invalid parameter \"credentialID\""
}
```

11.6 credentials/list

Description

Returns the list of credentials associated with a user identifier. A user MAY have one or multiple credentials hosted by a single remote signing service provider.
If requested, it can also return the signing certificate, the whole associated certificate chain, additional information about the signing certificate and/or information about the authorization mechanism required to authorize the access to the credentials for remote signing.
If the user is authenticated directly by the RSSP then the *userID* is implicit and SHALL NOT be specified.
This method can also be used in case of a community of users, to let the client retrieve the list of credentials assigned to a specific user of the community. In this case the *userID* SHALL be passed explicitly to retrieve the list of credentialIDs for a specific user. Managing a community of users that are authenticated by the client using a specific authentication framework is out of the scope of this specification.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
userID	REQUIRED Conditional	*String*	The identifier associated to the identity of the credential owner. This parameter SHALL NOT be present if the service authorization is user-specific (see NOTE below). In that case the *userID* is already implicit in the service access token passed in the Authorization header. If a user-specific service authorization is present, it SHALL NOT be allowed to use this parameter to obtain the list of credentials associated to a different user. The remote service SHALL return an error in such case.
credentialInfo	OPTIONAL	*Boolean*	Request to return the main information included in the public key certificate and the public key certificate itself or the certificate chain associated to the credentials. The default value is “false”, so if the parameter is omitted then the information will not be returned.
certificates	OPTIONAL Conditional	*String* none | single | chain	Specifies which certificates from the certificate chain SHALL be returned in *certs/certificates*. • “none”: No certificate SHALL be returned. • “single”: Only the end entity certificate SHALL be returned. • “chain”: The full certificate chain SHALL be returned. The default value is “single”, so if the parameter is omitted then the method will only return the end entity certificate(s). This parameter MAY be specified only if the parameter *credentialInfo* is “true”. If the parameter *credentialInfo* is not “true” and this parameter is specified its value SHALL be ignored.
certInfo	OPTIONAL Conditional	*Boolean*	Request to return various parameters containing information from the end entity certificate(s). This is useful in case the signature application wants to retrieve some details of the certificate(s) without having to decode it first. The default value is “false”, so if the parameter is omitted then the information will not be returned. This parameter MAY be specified only if the parameter *credentialInfo* is “true”. If the parameter *credentialInfo* is not “true” and this parameter is specified its value SHALL be ignored.
authInfo	OPTIONAL Conditional	*Boolean*	Request to return various parameters containing information on the authorization mechanisms supported by the corresponding credential (auth group). The default value is “false”, so if the parameter is omitted then the information will not be returned. This parameter MAY be specified only if the parameter *credentialInfo* is “true”. If the parameter *credentialInfo* is not “true” and this parameter is specified its value SHALL be ignored.
onlyValid	OPTIONAL Conditional	*Boolean*	Request to return only credentials usable to create a valid signature. The default value is “false”, so if the parameter is omitted then the method will return all credentials available to the owner. The remote service MAY NOT support this parameter. When the parameter is supported SHALL be returned in output.
lang	OPTIONAL	*String*	The *lang* as defined in the Input parameter table in info.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in oauth2/authorize.

Note 38: User-specific service authorization include the following *authType*: “basic”, “digest” and “oauth2code”. Non-user-specific service authorization include the following *authType*: “external”, “TLS” or “oauth2client”.

Output

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

Attribute	Presence	Value	Description
credentialIDs	REQUIRED	*Array of String*	One or more credentialID(s) associated with the provided or implicit *userID*.
credentialInfos	OPTIONAL Conditional	*Array of CredentialInfo*	The contents of *credentialInfo* object are described below. If the *credentialInfo* parameter is not “true”, this value SHALL NOT be returned.
onlyValid	REQUIRED Conditional	*Boolean*	This value SHALL be returned true when the input parameter “onlyValid” was true, and the RSSP supports this feature, i.e. the RSSP only returns credentials which can be used for signing. If the values is false or the output parameter is omitted, then the list may contain credentials which cannot be used for signing.

The ‘credentialInfo Object’ is a JSON Object composed by the attributes specified in the following table.

Attribute	Presence	Value	Description
credentialID	REQUIRED	*String*	The credentialID identifying one of the credentials associated with the provided or implicit *userID*.
description	OPTIONAL	*String*	A free form description of the credential in the *lang* language. The maximum size of the string is 255 characters.
signatureQualifier	OPTIONAL	*signatureQualifier*	Identifier qualifying the type of signature this credential is suitable for (see signatures/signDoc).
key/status	REQUIRED	*String* enabled | disabled	The status of the signing key of the credential: • "enabled": the signing key is enabled and can be used for signing. • "disabled": the signing key is disabled and cannot be used for signing. This MAY occur when the owner has disabled it or when the RSSP has detected that the associated certificate is expired or revoked.
key/algo	REQUIRED	*Array of String*	The list of OIDs of the supported key algorithms. For example: 1.2.840.113549.1.1.1 = RSA encryption, 1.2.840.10045.4.3.2 = ECDSA with SHA256.
key/len	REQUIRED	*Integer*	The length of the cryptographic key in bits.
key/curve	REQUIRED Conditional	*String*	The OID of the ECDSA curve. The value SHALL only be returned if *keyAlgo* is based on ECDSA.
cert/status	OPTIONAL	*String* valid | expired | revoked | suspended	The status of validity of the end entity certificate. The value is OPTIONAL, so the remote service SHOULD only return a value that is accurate and consistent with the actual validity status of the certificate at the time the response is generated.
cert/certificates	REQUIRED Conditional	*Array of String*	One or more base64-encoded X.509v3 certificates from the certificate chain. If the *certificates* parameter is "chain", the entire certificate chain SHALL be returned with the end entity certificate at the beginning of the array. If the *certificates* parameter is "single", only the end entity certificate SHALL be returned. If the *certificates* parameter is "none", this value SHALL NOT be returned.
cert/issuerDN	REQUIRED Conditional	*String*	The Issuer Distinguished Name from the X.509v3 end entity certificate as UTF-8-encoded character string according to RFC 4514 [4]. This value SHALL be returned when *certInfo* is "true".
cert/serialNumber	REQUIRED Conditional	*String*	The Serial Number from the X.509v3 end entity certificate represented as hex-encoded string format. This value SHALL be returned when *certInfo* is "true".
cert/subjectDN	REQUIRED Conditional	*String*	The Subject Distinguished Name from the X.509v3 end entity certificate as UTF-8-encoded character string, according to RFC 4514 [4]. This value SHALL be returned when *certInfo* is "true".
cert/validFrom	REQUIRED Conditional	*String*	The validity start date from the X.509v3 end entity certificate as character string, encoded as GeneralizedTime (RFC 5280 [8]) (e.g. "YYYYMMDDHHMMSSZ"). This value SHALL be returned when *certInfo* is "true".
cert/validTo	REQUIRED Conditional	*String*	The validity end date from the X.509v3 end entity certificate as character string, encoded as GeneralizedTime (RFC 5280 [8]) (e.g. "YYYYMMDDHHMMSSZ"). This value SHALL be returned when *certInfo* is "true".
auth/mode	REQUIRED	*String* explicit | oauth2code	Specifies one of the authorization modes. For more information also see OAuth 2.0 Authorization: • "explicit": the authorization process is managed by the signature application, which collects authentication factors of various types. • "oauth2code": the authorization process is managed by the remote service using an OAuth 2.0 mechanism based on authorization code as described in Section 1.3.1 of RFC 6749 [11].
auth/expression	OPTIONAL Conditional	*String*	An expression defining the combination of authentication objects required to authorize usage of the private key. If empty, an "AND" of all authentication objects is implied. Supported operators are: "AND" | "OR" | "XOR" | "(" | ")" This value SHALL NOT be returned if *auth/mode* is not "explicit".

Attribute	Presence	Value	Description
auth/objects	REQUIRED Conditional	*Array of authentication object types*	The authentication object types available for this credential. This value SHALL only be returned if *auth/mode* is “explicit”.
SCAL	OPTIONAL	*String* 1 | 2	Specifies if the RSSP will generate for this credential a signature activation data (SAD) or an access token with scope “credential” that contains a link to the hash to-be-signed: • “1”: The hash to-be-signed is not linked to the signature activation data. • “2”: The hash to-be-signed is linked to the signature activation data. This value is OPTIONAL and the default value is “1”. See Note below.
multisign	REQUIRED	*Integer* ≥ 1	A number equal or higher to 1 representing the maximum number of signatures that can be created with this credential with a single authorization request (e.g. by calling ***credentials/ signHash*** method, as defined in signatures/signHash, once with multiple hash values or calling it multiple times). The value of *numSignatures* specified in the authorization request SHALL NOT exceed the value of this value.
lang	OPTIONAL	*String*	The *lang* as defined in the Output parameter table in info.

Note 39: As described in the difference between SCAL1 and SCAL2 in Credential authorization, the value “2” only gives information on the link between the hash and the SAD (or access token with scope “credential”), it does not give information if a full SCAL2 as described in CEN TS 119 241-1 [i.5] is implemented.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
Not empty “userID” parameter in case of user-specific authorization	400 (bad request)	invalid_request	userID parameter MUST be null
Invalid “userID” format in case of no user-specific authorization	400 (bad request)	invalid_request	Invalid parameter userID
When present, invalid “certificates” parameter	400 (bad request)	invalid_request	Invalid parameter certificates

Examples

Sample Request

```
POST /csc/v2/credentials/list HTTP/1.1
Host: service.domain.org
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json

{
   "credentialInfo": true,
   "certificates": "chain",
   "certInfo": true,
   "authInfo": true
}
```

cURL Example

```
curl -i -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{"credentialInfo": true,
          "certificates": "chain",
          "certInfo": true,
          "authInfo": true}'
     https://service.domain.org/csc/v2/credentials/list
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "credentialIDs": [ "GX0112348", "HX0224685" ]
   "credentialInfos":
    [
        {
            "credentialID": "GX0112348",
            "key":
            {
                "status": "enabled",
                "algo": [ "1.2.840.113549.1.1.11", "1.2.840.113549.1.1.10" ],
                "len": 2048
            },
            "cert":
            {
                "status": "valid",
                "certificates":
                [
                    "<base64-encoded_X.509_end_entity_certificate>",
                    "<base64-encoded_X.509_intermediate_CA_certificate>",
                    "<base64-encoded_X.509_root_CA_certificate>"
                ],
                "issuerDN":"<X.500_issuer_DN_printable_string>",
                "serialNumber": "5AAC41CD8FA22B953640",
                "subjectDN": "<X.500_subject_DN_printable_string>",
                "validFrom": "20200101100000Z",
                "validTo": "20230101095959Z"
            },
            "auth": {
                "mode": "explicit",
                "expression": "PIN AND OTP",
                "objects": [
                    {
                        "type": "Password",
                        "id": "PIN",
                        "format": "N",
                        "label": "PIN",
                        "description": "Please enter the signature PIN"
                    },
                    {
                        "type": "Password",
                        "id": "OTP",
                        "format": "N",
                        "generator": "totp",
                        "label": "Mobile OTP",
                        "description": "Please enter the 6 digit code you received by SMS"
                    }
                ]
            }
            "multisign": 5,
            "lang": "en-US"
        },
        {
            "credentialID": "HX0224685",
            ………
            ………
        }
    ]
}
```

11.7 credentials/info

Description

Retrieves the credential. If requested, it can also return the signing certificate, the whole associated certificate chain, additional information about the signing certificate and/or information about the authorization mechanism required to authorize the access to the credential for remote signing.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
credentialID	REQUIRED	*String*	The unique identifier associated to the credential.
certificates	OPTIONAL	*String* none | single | chain	The *certificates* as defined in the Input parameter table in credentials/list.
certInfo	OPTIONAL	*Boolean*	The *certInfo* as defined in the Input parameter table in credentials/list.
authInfo	OPTIONAL	*Boolean*	The *authInfo* as defined in the Input parameter table in credentials/list.
lang	OPTIONAL	*Strings*	The *lang* as defined in the Input parameter table in info.
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:

Attribute	Presence	Value	Description	
description	OPTIONAL	*String*	The *description* as defined in the credentialInfo Object attribute table in credentials/list.	
signatureQualifier	OPTIONAL	*signatureQualifier*	Identifier qualifying the type of signature this credential is suitable for (see signatures/signDoc).	
key/status	REQUIRED	*String* enabled | disabled	The *key/status* as defined in the credentialInfo Object attribute table in credentials/list.	
key/algo	REQUIRED	*Array of String*	The *key/algo* as defined in the credentialInfo Object attribute table in credentials/list.	
key/len	REQUIRED	*Integer*	The *key/len* as defined in the credentialInfo Object attribute table in credentials/list.	
key/curve	REQUIRED Conditional	*String*	The *key/curve* as defined in the credentialInfo Object attribute table in credentials/list.	
cert/status	OPTIONAL	*String* valid | expired | revoked | suspended	The *cert/status* as defined in the credentialInfo Object attribute table in credentials/list.	
cert/certificates	REQUIRED Conditional	*Array of String*	The *cert/certificates* as defined in the credentialInfo Object attribute table in credentials/list.	
cert/issuerDN	REQUIRED Conditional	*String*	The *cert/issuerDN* as defined in the credentialInfo Object attribute table in credentials/list.	
cert/serialNumber	REQUIRED Conditional	*String*	The *cert/serialNumber* as defined in the credentialInfo Object attribute table in credentials/list.	
cert/subjectDN	REQUIRED Conditional	*String*	The *cert/subjectDN* as defined in the credentialInfo Object attribute table in credentials/list.	
cert/validFrom	REQUIRED Conditional	*String*	The *cert/validFrom* as defined in the credentialInfo Object attribute table in credentials/list.	
cert/validTo	REQUIRED Conditional	*String*	The *cert/validTo* as defined in the credentialInfo Object attribute table in credentials/list.	
cert/qcStatements	REQUIRED	*Array of Strings*		qcStatements: is an array of OIDs formated as strings that SHALL contain any value of qcStatements field (e.g.: [“0.4.0.1862.1.1”,

Attribute	Presence	Value	Description	
				“0.4.0.1862.1.4”, “0.4.0.1862.1.6.1”]). The qcStatements field is defined according to *rfc 3739* and *ETSI EN 319 412-5*. e.g. - id-etsi-qcs-QcCompliance - id-etsi-qcs-QcSSCD - id-etsi-qcs-QcType
cert/policy	OPTIONAL	*Array of String*	certificate policy OID under which the certificate was issued	
auth/mode	REQUIRED	*String* explicit | oauth2code	The *auth/mode* as defined in the credentialInfo Object attribute table in credentials/list.	
auth/expression	OPTIONAL Conditional	*String*	The *auth/expression* as defined in the credentialInfo Object attribute table in credentials/list.	
auth/objects	REQUIRED Conditional	*Array of authentication object types*	The *auth/objects* as defined in the credentialInfo Object attribute table in credentials/list.	
SCAL	OPTIONAL	*String* 1 | 2	The *SCAL* as defined in the credentialInfo Object attribute table in credentials/list. See the Note in credentials/list about the *SCAL* attribute.	
multisign	REQUIRED	*Integer* ≥ 1	The *multisign* as defined in the credentialInfo Object attribute table in credentials/list.	
lang	OPTIONAL	*String*	The *lang* as defined in the Output parameter table in info.	

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
Missing or not String “credentialID” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter credentialID
Invalid “credentialID” parameter	400 (bad request)	invalid_request	Invalid parameter credentialID
Invalid “certificates” parameter	400 (bad request)	invalid_request	Invalid parameter certificates

Examples

Sample Request

```
POST /csc/v2/credentials/info HTTP/1.1
Host: service.domain.org
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json

{
   "credentialID": "GX0112348",
   "certificates": "chain",
   "certInfo": true,
   "authInfo": true
}
```

cURL Example

```
curl -i -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{"credentialID": "GX0112348",
          "certificates": "chain",
          "certInfo": true,
          "authInfo": true }'
     https://service.domain.org/csc/v2/credentials/info
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "key":{
      "status":"enabled",
      "algo":[
         "1.2.840.113549.1.1.1",
         "0.4.0.127.0.7.1.1.4.1.3"
      ],
      "len":2048
   },
   "cert":{
      "status":"valid",
      "certificates":[
         "<base64-encoded_X.509_end_entity_certificate>",
         "<base64-encoded_X.509_intermediate_CA_certificate>",
         "<base64-encoded_X.509_root_CA_certificate>"
      ],
      "issuerDN":"<X.500_issuer_DN_printable_string>",
      "serialNumber":"5AAC41CD8FA22B953640",
      "subjectDN":"<X.500_subject_DN_printable_string>",
      "validFrom":"20180101100000Z",
      "validTo":"20190101095959Z"
   },
    "auth": {
        "mode": "explicit",
        "expression": "PIN AND OTP",
        "objects": {
            {
                "type": "Password",
                "id": "PIN",
                "format": "N",
                "label": "PIN",
                "description": "Please enter the signature PIN"
            },
            {
                "type": "Password",
                "id": "OTP",
                "format": "N",
                "generator": "totp",
                "label": "Mobile OTP",
                "description": "Please enter the 6 digit code you received by SMS"
            }
        }
    }
   "multisign":5,
   "lang":"en-US"
}
```

11.8 credentials/authorize

Description

Authorize the access to the credential for remote signing, according to the authorization mechanisms associated to it. This method returns the Signature Activation Data (SAD) required to authorize the **signatures/signHash** method, as defined in signatures/signHash or **signatures/signDoc** method, as defined in signatures/signDoc.

Authentication objects and corresponding values collected from the user SHALL be included in the request according to the requirements specified by the **credentials/info** method, as defined in credentials/info.
This method SHALL be used in case of "explicit" authorization. This method SHALL also be used in case that no authentication objects are required, to trigger a possible authorization mechanism managed by the remote service. This method SHALL NOT be used in case of "oauth2" credential authorization; instead, any of the available OAuth 2.0 authorization mechanisms SHALL be used.

The *numSignatures* parameter SHALL indicate the total number of signatures to authorize. In case of multi-signature transactions where the **signatures/signHash** method is invoked multiple times, the signature application SHALL obtain a new SAD by invoking the **credentials/extendTransaction** method, as defined in credentials/extendTransaction, before the current SAD expires. In such cases the hashes to be signed may not all be available when the authorization is performed, for example in case of multiple signatures applied to a PDF file with a single credential. Further hashes should then be passed as an input to **credentials/extendTransaction** to make each SAD calculation dependent on the data to be signed. This approach may break the support of SCAL 2 requirements, therefore a remote signing service MAY fail if the *hash* parameter does not contain a number of hash values corresponding to the value in *numSignatures*.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
credentialID	REQUIRED	*String*	The *credentialID* as defined in the Input parameter table in credentials/info.
numSignatures	REQUIRED	*Integer*	The number of signatures to authorize. Multi-signature transactions can be obtained by using a combination of passing an array of hash values and calling the **signatures/signHash** method, as defined in signatures/signHash, multiple times.
hashes	REQUIRED Conditional	*Array of String*	One or more base64-encoded hash values to be signed. It allows the server to bind the *SAD* to the hash(es), thus preventing an authorization to be used to sign a different content. If the *SCAL* parameter returned by **credentials/info** method, as defined in credentials/info, for the current credentialID is "2" the *hash* parameter SHALL be used and the number of hash values SHOULD correspond to the value in *numSignatures*. If the SCAL parameter is "1", the *hash* parameter is OPTIONAL.
hashAlgorithmOID	REQUIRED Conditional	*String*	String containing the OID of the hash algorithm used to generate the hash values.
authData	REQUIRED Conditional	*Array of authentication objects*	The authentication objects as described by the authentication object types in **credentials/info**. It SHALL be used only when *auth/mode* from **credentials/info** is "explicit".
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.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in oauth2/authorize.

Authentication objects

An **authentication object** type describes the data structure and protocol of authentication mechanisms, much the same way as it is done in the PKCS#15 standard. Authentication object types are returned by the *credentials/info* method, such that the Signature Application can show the proper interactive controls to collect them from the user.

Each authentication object type is associated with a `type` property, defining both the data structure that a client application SHALL provide and the protocol that SHALL be processed. The `id` property is used to identify the associated authentication object type in a concrete authentication object data structure.

The number and type of authentication object types is provider specific.

Depending on the authentication object type the Signature Application collects concrete authentication object data and drives the associated protocol. The authentication object data is sent using the method *credentials/authorize*.

The following is an example authentication object type, describing the need for a password entry:

```
{
    "type": "Password",
    "id": "PIN",
    "label": "Personal PIN",
    "format" : "A"
}
```

This indicates to the client that it needs to send an alphanumeric password within a later authorization request, identifying it as “PIN”. When requesting user input, the client may present the required data as “Personal PIN” to the user.

In consequence, the client might send an authentication object as seen in the following example:

```
{
    "id": "PIN",
    "value": "1234"
}
```

This example assumes that the client has received a PIN value of “1234”, which is conveyed to the authorization endpoint.

See the following sections for a thorough description of these data structures.

Out-of-band response

“Out-of-band response” is used here whenever an authentication object is sent to the service provider by using some protocol and session not associated and described in this API specification. This can be for example an SMS, phone or email channel.

With an out-of-band response the call to *credentials/authorize* does not have any knowledge about the state of the out-of-band task. Processing of the call can be implemented using a polling or blocking approach. As such, the processing can either

- terminate with an HTTP 200, returning the specified result tokens.
- terminate with an HTTP 202 as an indication that the out-of-band result is not yet available. The client has to re-issue a request to *credentials/authorizeCheck* (polling). Eventually the request will terminate with an error or HTTP 200.

Common properties

The following properties are common to all authentication object types as they are received from *credentials/info*.

Name	Presence	Description
type	REQUIRED	The type of the authentication object. This describes the data structure and protocol. The value SHALL be one of the tokens defined in this specification. A provider MAY not support all token types.
id	REQUIRED	The unique identifier of the authentication object.
label	OPTIONAL	A label to be presented to the user. It is used to identify the requested authentication data in human-readable manner.
description	OPTIONAL	A description to be presented to the user. It carries instructions on how to provide the authentication data.

The following properties are common to all authentication objects as they are sent via *credentials/authorize*.

Name	Presence	Description
id	REQUIRED	The unique identifier of the authentication object.

Password, in band response

The **Password** type simply requires the client to collect authentication information from the user and send it to the provider in-band.

Be aware that from a provider point of view an OTP generated statically / offline by a client side token is simply a “Password” type, too.

This authentication object type allows for the definition of

- Simple password authentication
- “offline” OTP generation
- Combinations thereof, e.g. the requirement of having two PINs entered (4 eyes).

Authentication type properties:

Name	Presence	Value	Description
type	REQUIRED	“Password”	
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 can have issued multiple tokens and allows the user to identify the required one using this property.

Authentication object properties:

Name	Presence	Description
value	REQUIRED	The concrete password value.

Example I authentication type:

```
{
    "type": "Password",
    "id": "PIN",
    "label": "PIN",
    "format" : "N"
}
```

Example I authentication object:

```
{
    "id": "PIN",
    "value": "1234"
}
```

Example II authentication type:

```
{
    "type": "Password",
    "id": "OTP",
    "label": "OTP",
    "generator" : "b23",
    "format" : "A"
}
```

Example II authentication object:

```
{
    "id": "OTP",
    "value": "3rfd45s"
}
```

Password, out of band response

The **PasswordOOB** indicates that by some unspecified mechanism an authentication object is sent to the service provider.

This authentication object type allows for the definition of

- SMS, phone or email authorization
- Provider-specific authorization without user agent intervention

Authentication type properties:

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"
}
```

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 an 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
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 an 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.
- An 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"
}
```

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 an 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.
- An 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.

Output

With HTTP status code 200, the method returns the Signature Activation Data using the “application/json” format:

Attribute	Presence	Value	Description
SAD	REQUIRED	*String*	The Signature Activation Data (SAD) to be used as input to the **signatures/signHash** method, as defined in signatures/signHash.
expiresIn	OPTIONAL	*Integer*	The lifetime in seconds of the SAD. If omitted, the default expiration time is 3600 (1 hour).

With HTTP status code 202 the method indicates that some authorization is still underway. The result contains a handle that can be used to poll the state of the authorization via **credentials/authorizeCheck**.

Attribute	Presence	Value	Description
handle	REQUIRED	*String*	An opaque handle that can be used to request the state of the authorization.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
Missing or not String “credentialID” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter credentialID
Invalid “credentialID” parameter	400 (bad request)	invalid_request	Invalid parameter credentialID
Signing key for “credentialID” is disabled	400 (bad request)	invalid_request	The credential identified by credentialID is disabled
Missing or not integer “numSignatures” parameter	400 (bad request)	invalid_request	Missing (or invalid type) integer parameter numSignatures
“numSignatures” < 1	400 (bad request)	invalid_request	Invalid value for parameter numSignatures
“numSignatures” > “multisign”	400 (bad request)	invalid_request	Numbers of signatures is too high
Invalid authentication data	400 (bad request)	invalid_authentication_data	The authentication data is invalid
Credential locked	400 (bad request)	invalid_request	Credential locked

Note 40: In case wrong authentication data is provided several times, the remote signing service MAY lock the credential or the usage of respective authentication objects. The policy adopted by the RSSP in this regard is out of the scope of this specification.

Examples

Sample Request

```
POST /csc/v2/credentials/authorize HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "credentialID":"GX0112348",
   "numSignatures":2,
   "hashes":[
      "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
      "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
   ],
   "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
   "authData": [
        {
            "id": "PIN",
            "value": "123456"
        },
        {
            "id": "OTP",
            "value": "738496"
        }
   ],
   "clientData":"12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{ "credentialID": "GX0112348",
           "numSignatures": 2,
           "hashes": [ "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
           "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
           ],
       "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
           "authData": [
                {
                    "id": "PIN",
                    "value": "123456"
                },
                {
                    "id": "OTP",
                    "value": "738496"
                }
           ],
           "clientData": "12345678" }'
     https://service.domain.org/csc/v2/credentials/authorize
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "SAD":"_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw"
}
```

Sample Response

```
HTTP/1.1 202 OK
Content-Type: application/json;charset=UTF-8

{
   "handle": "878287f37b2bv293bv2bv237bv297bvbv"
}
```

11.9 credentials/authorizeCheck

Description

After a credentials/authorize with HTTP result code 202, the client may use the handle returned to poll the authorization state.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
handle	REQUIRED	*String*	The handle value returned from **credentials/authorize**.

Output

With HTTP status code 200, the method returns the Signature Activation Data using the “application/json” format:

Attribute	Presence	Value	Description
SAD	REQUIRED	*String*	The Signature Activation Data (SAD) to be used as input to the **signatures/signHash** method, as defined in signatures/signHash.
expiresIn	OPTIONAL	*Integer*	The lifetime in seconds of the SAD. If omitted, the default expiration time is 3600 (1 hour).

With HTTP status code 202 the method indicates that some authorization is still underway. The result contains a handle that can be used to poll the state of the authorization via repeated calls to **credentials/authorizeCheck**.

Attribute	Presence	Value	Description
handle	REQUIRED	*String*	An opaque handle that can be used to request the state of the authorization.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
Invalid “handle” parameter	400 (bad request)	invalid_request	Invalid parameter handle
Invalid authentication data	400 (bad request)	invalid_authentication_data	The authentication data is invalid
Credential locked	400 (bad request)	invalid_request	Credential locked

Note 41: In case wrong authentication data is provided several times, the remote signing service MAY lock the credential or the usage of respective authentication objects. The policy adopted by the RSSP in this regard is out of the scope of this specification.

Examples

Sample Request

```
POST /csc/v2/credentials/authorizeCheck HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "handle":"878287f37b2bv293bv2bv237bv297bvbv"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{ "handle":"878287f37b2bv293bv2bv237bv297bvbv" }'
     https://service.domain.org/csc/v2/credentials/authorizeCheck
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw"
}
```

Sample Response

```
HTTP/1.1 202 OK
Content-Type: application/json;charset=UTF-8

{
   "handle": "878287f37b2bv293bv2bv237bv297bvbv"
}
```

11.10 credentials/getChallenge

Description

Get a challenge for the referenced authentication object.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
credentialID	REQUIRED	*String*	The identifier associated to the credential.
authObjectID	REQUIRED	*String*	The identifier of the authentication object we need a challenge for.

Output

With HTTP status code 200, the method returns a challenge:

Attribute	Presence	Value	Description
challenge	REQUIRED	*String*	The authentication object challenge.

With HTTP status code 204, the method indicates that a challenge has been sent by out-of-band means, returning no output values.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
Invalid “credentialID” parameter	400 (bad request)	invalid_request	Invalid parameter credentialID
Invalid “authObjectID” parameter	400 (bad request)	invalid_request	Invalid parameter authObjectID

Examples

In-band challenge

Sample Request

```
POST /csc/v2/credentials/getChallenge HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "credentialID": "GX0112348",
   "authObjectID": "fallback question"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{
        "credentialID": "GX0112348",
        "authObjectID": "fallback question"
     }'
     https://service.domain.org/csc/v2/credentials/getChallenge
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "challenge": "What's your mother's birth name?"
}
```

Out-of-band challenge

Sample Request

```
POST /csc/v2/credentials/getChallenge HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "credentialID": "GX0112348",
   "authObjectID": "OTP"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{
        "credentialID": "GX0112348",
        "authObjectID": "OTP"
     }'
     https://service.domain.org/csc/v2/credentials/getChallenge
```

Sample Response

```
HTTP/1.1 204 OK
```

11.11 credentials/extendTransaction

Description

Extends the validity of a multi-signature transaction authorization by obtaining a new Signature Activation Data (SAD). This method SHALL be used in case of multi-signature transaction when the API method **signatures/signHash**, as defined in signatures/signHash, is invoked multiple times with a single credential authorization event.
It can also be used to renew a SAD, before it expires, when signature operations take longer than allowed by the *expiresIn* value. Expired SAD cannot be extended.

The RSSP SHALL invalidate the SAD when the number of authorized signatures, specified with *numSignatures* in the credential authorization event, has been created.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
credentialID	REQUIRED	*String*	The *credentialID* as defined in the Input parameter table in credentials/info.
hashes	REQUIRED Conditional	*Array of String*	One or more base64-encoded hash values to be signed. It allows the server to bind the new *SAD* to the hash, thus preventing an authorization to be used to sign a different content. It SHALL be used if the *SCAL* parameter returned by **credentials/info**, as defined in credentials/info, for the current *credentialID* is "2" , otherwise it is OPTIONAL.
hashAlgorithmOID	REQUIRED Conditional	*String*	String containing the OID of the hash algorithm used to generate the hash values.
SAD	REQUIRED	*String*	The current unexpired Signature Activation Data. This token is returned by the **credentials/authorize**, as defined in credentials/authorize, or by the previous call to **credentials/extendTransaction**.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in oauth2/authorize.

Note 42: This method can be used for applying multiple signatures to a PDF document from a single user, e.g. to sign separately different parts of the document, with a single credential authorization event. The PDF standard adopts nested signatures so the hashes for multiple signatures can only be calculated after the previous signature has been created. This method allows to calculate a new SAD based on new hash values that were not available when the credential authorization event occurred. The sequence diagram in Create a remote multi-signatures transaction with a PDF document shows this use case.

Output

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

Attribute	Presence	Value	Description
SAD	REQUIRED	*String*	The new Signature Activation Data required to sign multiple times with a single authorization.
expiresIn	OPTIONAL	*Integer*	The lifetime in seconds of the SAD. If omitted, the default expiration time is 3600 (1 hour).

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	Malformed authorization header.

Note 43: In case a wrong PIN or OTP is provided several times, the remote signing service MAY lock the credential or the usage of the PIN or OTP. The policy adopted by the RSSP in this regard is out of the scope of this specification.

Examples

Sample Request

```
POST /csc/v2/credentials/extendTransaction HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "credentialID":"GX0112348",
   "hashes":[
      "WlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
   ],
   "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
   "SAD":"_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
   "clientData":"12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{ "credentialID": "GX0112348",
             "hashes": [ "WlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0=" ],
       "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
             "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
             "clientData": "12345678" }'
     https://service.domain.org/csc/v2/credentials/extendTransaction
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "SAD":"1/UsHDJ98349h9fgh9348hKKHDkHWVkl/8hsAW5usc8_5="
}
```

11.12 credentials/sendOTP

Note 44: New implementations of signature applications should request a challenge from the service provider using credentials/getChallenge instead of invoking **credentials/sendOTP**.

Description

Start an online One-Time Password (OTP) generation mechanism associated with a credential and managed by the remote service. This will generate a dynamic one-time password that will be delivered to the user who owns the credential through an agreed communication channel managed by the remote service (e.g. SMS, email, app, etc.). This method SHOULD only be used with “online” OTP generators. In case of “offline” OTP, the signature application SHOULD NOT invoke this method because the OTP can be generated autonomously by the user.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
credentialID	REQUIRED	*String*	The credentialID as defined in the Input parameter table in credentials/info.
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
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	Malformed authorization header.
The "credentialID" parameter or not of type String	400 (bad request)	invalid_request	Missing (or invalid type) string parameter credentialID
Invalid "credentialID" parameter	400 (bad request)	invalid_request	Invalid parameter credentialID
OTP locked	400 (bad request)	invalid_request	OTP locked

Note 45: In case a wrong PIN or OTP is provided several times, the remote signing service MAY lock the credential or the usage of the PIN or OTP. The policy adopted by the RSSP in this regard is out of the scope of this specification.

Examples

Sample Request

```
POST /csc/v1/credentials/sendOTP HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
{
"credentialID": "GX0112348",
"clientData": "12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{ "credentialID": "GX0112348",
           "clientData": "12345678" }'
https://service.domain.org/csc/v1/credentials/sendOTP
```

Sample Response

```
HTTP/1.1 204 No Content
```

11.13 signatures/signHash

Description

Calculate the remote digital signature of one or multiple hash values provided in input.

This method requires service and Credential authorization.

The signing application MUST pass an access token with scope "service" or "credential" in the "Authorization" HTTP header as defined in RFC 6750 [12].

If the credential authorization mode is "explicit", the signing application MUST pass Signature Activation Data (SAD) in the `SAD` request parameter (see below). SAD may be obtained from credentials/authorize.

If the credential authorization mode is “oauth2code” and the access token passed in the “Authorization” HTTP header has scope “service”, the signing application MUST pass an access token with scope “credential” in the `SAD` request parameter. This is not required if the access token passed in the “Authorization” HTTP header has scope “credential”.

In case of multi-signature transactions, the SAD SHALL be updated with credentials/extendTransaction every time this method is invoked until the maximum number of authorized signatures has been generated.

Input

The input for this method is the union of the following:

- *signingAlgorithm* (see the CSC data model [37])
- The parameters in the table below

Parameter	Presence	Value	Description
credentialID	REQUIRED	*String*	The credentialID as defined in the Input parameter table in credentials/info.
SAD	REQUIRED Conditional	*String*	The Signature Activation Data returned by the Credential Authorization methods. Not needed if the signing application has passed an access token in the “Authorization” HTTP header with scope “credential”, which is also good for the credential identified by `credentialID`. Note: For backward compatibility, signing applications MAY pass access tokens with scope “credential” in this parameter.
hashes	REQUIRED	*Array of String*	One or more hash values to be signed. This parameter SHALL contain the base64-encoded raw message digest(s).
hashAlgorithmOID	REQUIRED Conditional	*String*	The OID of the algorithm used to calculate the hash value(s). This parameter SHALL be ignored if the hash algorithm is implicitly specified by *signAlgo* or *signAlgoParams* (see *signingAlgorithm* in the CSC data model [37]). Only hashing algorithms as strong or stronger than SHA256 SHALL be used. The hash algorithm SHOULD follow the recommendations of ETSI TS 119 312 [21].
operationMode	OPTIONAL	*String*	The type of operation mode requested to the remote signing server. It SHALL take one of the following values: • “A”: an asynchronous operation mode is requested. • “S”: a synchronous operation mode is requested. The default value is “S”, so if the parameter is omitted then the remote signing server will manage the request in synchronous operation mode.
validity_period	OPTIONAL Conditional	*Integer*	Maximum period of time, expressed in milliseconds, until which the server SHALL keep the request outcome(s) available for the client application retrieval. This parameter MAY be specified only if the parameter *operationMode* is “A”. If the parameter *operationMode* is not “A” and this parameter is specified its value SHALL be ignored. The RSSP SHOULD define in its service policy the default and maximum values of this parameter. If the RSSP does not define in its service policy any default and maximum values of this parameter it means that any value MAY be passed in this parameter.
response_uri	OPTIONAL Conditional	*String*	Value of one location where the server will notify the signature creation operation completion, as a URI value. This parameter MAY be specified only if the parameter *operationMode* is “A”. If the parameter *operationMode* is not “A” and this parameter is specified its value SHALL be ignored. If the parameter operationMode is “A” and this parameter is omitted then the remote signing server will not make any notification.
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:

Attribute	Presence	Value	Description
signatures	REQUIRED Conditional	*Array of String*	One or more base64-encoded signed hash(s). In case of multiple signatures, the signed hashes SHALL be returned in the same order as the corresponding hashes provided as an input parameter. This value SHALL be returned when *operationMode* is not “A”.
responseID	REQUIRED Conditional	*String*	An opaque handle that can be used to request the state of the signing process via signatures/signPolling. This value SHALL be returned when *operationMode* is “A”.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
Missing or not String “SAD” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter SAD
Invalid “SAD” parameter	400 (bad request)	invalid_request	Invalid parameter SAD
Missing or not String “credentialID” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter credentialID
Invalid “credentialID” parameter	400 (bad request)	invalid_request	Invalid parameter credentialID
Missing or not Array “hashes” parameter	400 (bad request)	invalid_request	Missing (or invalid type) array parameter hashes
Empty “hashes” parameter	400 (bad request)	invalid_request	Empty hashes array
Invalid base64 hash element	400 (bad request)	invalid_request	Invalid Base64 hash string parameter
Unauthorized hash	400 (bad request)	invalid_request	Hash is not authorized by neither SAD nor an access token with scope “credential”.
Missing or not String “signAlgo” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter signAlgo
Invalid “signAlgo” parameter	400 (bad request)	invalid_request	Invalid parameter signAlgo
Missing or not String “signAlgoParams” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter signAlgoParams
Invalid “signAlgoParams” parameter	400 (bad request)	invalid_request	Invalid parameter signAlgoParams
Missing or not String “hashAlgorithmOID” parameter when hash algorithm is not implied by “signAlgo” or “signAlgoParams”	400 (bad request)	invalid_request	Missing (or invalid type) string parameter hashAlgorithmOID
“hashAlgorithmOID” parameter contradicting with “signAlgo” or “signAlgoParams” parameter	400 (bad request)	invalid_request	String parameter hashAlgorithmOID contradicts with signAlgo signAlgoParams parameter
When present, invalid “hashAlgorithmOID” parameter	400 (bad request)	invalid_request	Invalid parameter hashAlgorithmOID
When present, invalid “operationMode” parameter	400 (bad request)	invalid_request	Invalid parameter operationMode
When present, invalid “validity_period” parameter	400 (bad request)	invalid_request	Invalid parameter validity_period
When present, out of bounds “validity_period” parameter	400 (bad	invalid_request	Out of bounds parameter validity_period

Error Case	Status Code	Error	Error Description
	request)		
When present, invalid “response_uri” parameter	400 (bad request)	invalid_request	Invalid parameter response_uri
When present, invalid “clientData” format (not string)	400 (bad request)	invalid_request	Invalid parameter clientData
Invalid “hashes” length	400 (bad request)	invalid_request	Invalid digest value length
The OTP used to generate the “SAD” is invalid	400 (bad request)	invalid_otp	The OTP is invalid
Expired “SAD”	400 (bad request)	invalid_request	SAD expired
Expired credential	400 (bad request)	invalid_request	Signing certificate ‘O=[organization],CN=[common_name]’ is expired.

Examples

Sample Request

```
POST /csc/v2/signatures/signHash HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "credentialID":"GX0112348",
   "SAD":"_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
   "hashes":[
      "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
      "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
   ],
   "hashAlgorithmOID":"2.16.840.1.101.3.4.2.1",
   "signAlgo":"1.2.840.113549.1.1.1",
   "clientData":"12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer
     4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{ "credentialID": "GX0112348",
           "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
           "hashes": [ "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
           "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
           ],
           "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
           "signAlgo": "1.2.840.113549.1.1.1",
           "clientData": "12345678"}'
     https://service.domain.org/csc/v2/signatures/signHash
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "signatures":
   [
      "KedJuTob5gtvYx9qM3k3gm7kbLBwVbEQRl26S2tmXjqNND7MRGtoew==",
      "Idhef7xzgtvYx9qM3k3gm7kbLBwVbE98239S2tm8hUh85KKsfdowel=="
   ]
}
```

Sample Request

```
POST /csc/v2/signatures/signHash HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "credentialID":"GX0112348",
   "SAD":"_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
   "hashes":[
      "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
      "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
   ],
   "hashAlgorithmOID":"2.16.840.1.101.3.4.2.1",
   "signAlgo":"1.2.840.113549.1.1.1",
   "operationMode": "A",
   "clientData":"12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer
     4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{ "credentialID": "GX0112348",
           "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
           "hashes": [ "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
           "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
           ],
           "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
           "signAlgo": "1.2.840.113549.1.1.1",
           "operationMode": "A",
           "clientData": "12345678"}'
     https://service.domain.org/csc/v2/signatures/signHash
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "responseID":"158112-652341-khj"
}
```

Sample Request

```
POST /csc/v2/signatures/signHash HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 5/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "credentialID":"GX0112348",
   "hashes":[
      "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
      "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
   ],
   "hashAlgorithmOID":"2.16.840.1.101.3.4.2.1",
   "signAlgo":"1.2.840.113549.1.1.1",
   "operationMode": "A",
   "clientData":"12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer
     5/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{ "credentialID": "GX0112348",
           "hashes": [ "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
           "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
           ],
           "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
           "signAlgo": "1.2.840.113549.1.1.1",
           "operationMode": "A",
           "clientData": "12345678"}'
     https://service.domain.org/csc/v2/signatures/signHash
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
   "responseID":"158112-652341-khj"
}
```

11.14 signatures/signDoc

Description

Create one or more AdES signatures. Either the documents to be signed or the SDRs (in this specification it is intended to be the hash values of the documents to be signed) SHALL be provided to the method. An AdES signature will be created for each of these input components. Other components are used to select the type of signature that will be created for each document or document representation.

This method requires service and Credential authorization as defined in signatures/signHash.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
credentialID	REQUIRED Conditional	*String*	The credentialID as defined in the Input parameter table in credentials/info. At least one of the two values *credentialID* and *signatureQualifier* SHALL be present. Both values MAY be present.
signatureQualifier	REQUIRED Conditional	*signatureQualifier*	Identifier of the signature type to be created, e.g. “eu_eidas_qes” to denote a Qualified Electronic Signature according to eIDAS. The *signatureQualifier* JSON object is defined in the CSC data model [37]. At least one of the two values *credentialID* and *signatureQualifier* SHALL be present. Both values MAY be present.
SAD	REQUIRED Conditional	*String*	The Signature Activation Data returned by the Credential Authorization methods. Not needed if the signing application has passed an access token with scope “credential” in the “Authorization” HTTP header, which is also good for the credential identified by `credentialID` or the signature qualifier identified by `signatureQualifier`. Note: For backward compatibility, signing applications MAY pass access tokens with scope “credential” in this parameter.
documentDigests	REQUIRED Conditional	*Array of signatureCreationRequest*	Array of `signatureCreationRequest` objects as defined in the CSC data model [37]. All `signatureCreationRequest` objects MUST be with *documentRepresentations*. This parameter or the parameter *documents* MUST be present in a request. Otherwise the method SHALL return an error condition. If the `hashes` parameter was set during authorization, then the RSSP SHALL verify that each of the hashes in this parameter corresponds to one of the hashes provided in the authorization. If the parameter `hashAlgorithmOID` is passed and is in contradiction with the value of the *signAlgo* field in this parameter the method SHALL return an error condition.
documents	REQUIRED Conditional	*Array of signatureCreationRequest*	Array of `signatureCreationRequest` objects as defined in the CSC data model [37]. All `signatureCreationRequest` objects MUST be with *documentData*. This parameter or the parameter *documentDigests* MUST be present in a request. Otherwise the method SHALL return an error condition. If the `hashes` parameter was set during authorization, then the RSSP SHALL verify that the hash of each of the documents in this parameter corresponds to one of the hashes provided in the credential authorization.
hashAlgorithmOID	REQUIRED Conditional	*String*	Hashing algorithm OID used to calculate the hash values. This parameter SHALL be ignored if the hash algorithm is implicitly specified by *signAlgo* or *signAlgoParams* in the *documentDigests* or *documents* parameter. Only hashing algorithms as strong or stronger than SHA256 SHALL be used. The hash algorithm SHOULD follow the recommendations of ETSI TS 119 312 [21].
operationMode	OPTIONAL	*String*	The *operationMode* as defined in the Input parameter table in signatures/signHash.
validity_period	OPTIONAL Conditional	*Integer*	The *validity_period* as defined in the Input parameter table in signatures/signHash.
response_uri	OPTIONAL Conditional	*String*	The *response_uri* as defined in the Input parameter table in signatures/signHash.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in oauth2/authorize.
returnValidationInfo	OPTIONAL	*Boolean*	This parameter SHALL be set to “true” to request the service to return the “validationInfo” as defined below. The default value is “false”, i.e. no “validationInfo” info is provided. This parameter SHALL be supported in conjunction with “signature_format” “P”, “conformance_level” “AdES-B-LT” and use of “documentDigests”. For all other cases, the *info* methods states if this feature is supported or not.

The possible input and output formats for **signatures/signDoc** are listed in the following table.

signature_format	signed_envelope_property	Input Parameter		Output Parameter	
		documents	documentDigests	DocumentWithSignature	SignatureObject
CAdES	Attached	Signer's original document	-	CAdES signed document (ie. with SD in `eContent`)	-
	Detached	Signer's original document	-	-	`SignedData` as defined in ETSI EN 319 122-1 [29]
		-	SDR	-	`SignedData` as defined in ETSI EN 319 122-1 [29]
	Parallel	Signer's original document	-	-	`SignerInfo` as defined in ETSI EN 319 122-1 [29]
		-	SDR	-	`SignerInfo` as defined in ETSI EN 319 122-1 [29]
PAdES	Certification	Signer's original document / signer's formatted document	-	PAdES signed document	-
		-	SDR	-	CMS `SignedData` as defined in ETSI EN 319 142-1 [31]
	Revision	Signer's original document / signer's formatted document	-	PAdES signed document	-
		-	SDR	-	CMS `SignedData` as defined in ETSI EN 319 142-1 [31]
XAdES	Enveloped	Signer's original document / signer's formatted document	-	XAdES signed document	-
	Enveloping	Signer's original document / signer's formatted document	-	-	`Signature` with signed data included as defined in ETSI EN 319 132-1 [30]
	Detached	Signer's original document / signer's formatted document	-	-	`Signature` as defined in ETSI EN 319 132-1 [30]
		-	SDR	-	`Signature` as defined in ETSI EN 319 132-1 [30]
JAdES	Attached	Signer's original document / signer's	-	JAdES signed document	-

signature_format	signed_envelope_property	Input Parameter		Output Parameter	
		documents	documentDigests	DocumentWithSignature	SignatureObject
		formatted document			
	Detached	Signer's original document / signer's formatted document	-	-	JAdES detached signature as defined in ETSI TS 119 182-1 [32]
		-	SDR	-	JAdES detached signature as defined in ETSI TS 119 182-1 [32]
	Parallel	Signer's original document / signer's formatted document	-	-	`signature` as defined in ETSI TS 119 182-1 [32]
		-	SDR	-	`signature` as defined in ETSI TS 119 182-1 [32]

Note 46: When signing signer's original document at SCAL2, the RSSP MAY require `circumstantialData` to be present in the authorization as described in oauth2/authorize and/or as parameter to **signatures/signDoc**.

Output

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

Parameter	Presence	Value	Description
DocumentWithSignature	REQUIRED Conditional	*Array of String*	One or more base64-encoded signatures enveloped within the documents. This element SHALL carry a value only if the client application requested the creation of signature(s) enveloped within the signed document(s) and when *operationMode* is not "A".
SignatureObject	REQUIRED Conditional	*Array of String*	One or more base64-encoded signatures that are detached from or contain the signed data. This element SHALL carry a value only if the client application requested the creation of detached or enveloping signature(s) and when *operationMode* is not "A".
responseID	REQUIRED Conditional	*String*	An opaque handle that can be used to request the state of the signing process via signatures/signPolling. This value SHALL be returned when *operationMode* is "A".
validationInfo	REQUIRED Conditional	*JSON Object*	The *validationInfo* is a JSON Object containing validation data. This element SHALL be included in the signing response if requested using the input parameter *returnValidationInfo* and when *operationMode* is not "A".

The `validationInfo` is a JSON Object composed by the following parameters:

- `ocsp`
- `crl`
- `certificates`

specified according to the following table.

Parameter	Presence	Value	Description
ocsp	REQUIRED Conditional	*Array of String*	*ocsp* is an array of base64 encoded strings containing the DER-encoded ASN.1 data structures of type `OCSPResponse` according to RFC 6960 [33]. This value SHALL be included if at least one OCSP response is needed to validate the created signature and timestamps contained in the signature. It SHALL contain all needed OCSP responses. If for the same certificate an OCSP response and a CRL is available, the OCSP response SHOULD be included.
crl	REQUIRED Conditional	*Array of String*	*crl* is an array of base64 encoded strings containing the DER-encoded ASN.1 data structures of type `CertificateList` according to RFC 5280 [8]. This value SHALL be included if at least one CRL is needed to validate the created signature and timestamps contained in the signature. It SHALL contain all needed CRLs.
certificates	REQUIRED Conditional	*Array of String*	*certificates* contains one or more base64-encoded X.509v3 certificates from the certificate chain used to create the respective signature and timestamps included in the signature. This value SHALL be included if at least one certificate is needed to validate the created signature and timestamps, which is not yet included in the signature. It SHALL contain all needed certificates.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
Missing or not String “SAD” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter SAD
Invalid “SAD” parameter	400 (bad request)	invalid_request	Invalid parameter SAD
Missing or not String “credentialID” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter credentialID
Invalid “credentialID” parameter	400 (bad request)	invalid_request	Invalid parameter credentialID
When present, invalid object “documentDigests” parameter	400 (bad request)	invalid_request	Invalid object parameter documentDigests
When present, invalid array “documents” parameter	400 (bad request)	invalid_request	Invalid array parameter documents
Empty documentDigests and documents parameters	400 (bad request)	invalid_request	Empty documentDigests and documents objects
Both documentDigests and documents parameters have been passed	400 (bad request)	invalid_request	Both documentDigests and documents parameters passed
Invalid base64 hashes element	400 (bad request)	invalid_request	Invalid Base64 hashes string parameter
Invalid base64 documents element	400 (bad request)	invalid_request	Invalid Base64 documents string parameter
Unauthorized documentDigests or documents	400 (bad request)	invalid_request	documentDigests or documents are not authorized by the SAD.
Unauthorized attribute	400 (bad request)	invalid_request	One or more attributes in the documents parameter is not authorized.
Missing or not String “signAlgo” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter signAlgo
Invalid “signAlgo” parameter	400 (bad request)	invalid_request	Invalid parameter signAlgo
Missing or not String “signAlgoParams” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter signAlgoParams
Invalid “signAlgoParams” parameter	400 (bad request)	invalid_request	Invalid parameter signAlgoParams
Missing or not String “hashAlgorithmOID” parameter when hash algorithm is not implied by “signAlgo” or “signAlgoParams”	400 (bad request)	invalid_request	Missing (or invalid type) string parameter hashAlgorithmOID

Error Case	Status Code	Error	Error Description
“hashAlgorithmOID” parameter contradicting with “signAlgo” or “signAlgoParams” parameter	400 (bad request)	invalid_request	String parameter hashAlgorithmOID contradicts with signAlgo or signAlgoParams parameter
When present, invalid “hashAlgorithmOID” parameter	400 (bad request)	invalid_request	Invalid parameter hashAlgorithmOID
When present, invalid “signature_format” parameter	400 (bad request)	invalid_request	Invalid parameter signature_format
When “documents” is passed, missing or not String “signature_format” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter signature_format
When present, invalid “conformance_level” parameter	400 (bad request)	invalid_request	Invalid parameter conformance_level
When present, invalid “signed_envelope_property” parameter	400 (bad request)	invalid_request	Invalid parameter signed_envelope_property
More than one “signature_format” or “signed_envelope_property” when “allowMix” is false	400 (bad request)	invalid_request	Only a single signature_format and signed_envelope_property is allowed
When present, invalid “signed_props” parameter	400 (bad request)	invalid_request	Invalid parameter signed_props (list of invalid attributes)
When present, invalid “operationMode” parameter	400 (bad request)	invalid_request	Invalid parameter operationMode
When present, invalid “validity_period” parameter	400 (bad request)	invalid_request	Invalid parameter validity_period
When present, out of bounds “validity_period” parameter	400 (bad request)	invalid_request	Out of bounds parameter validity_period
When present, invalid “response_uri” parameter	400 (bad request)	invalid_request	Invalid parameter response_uri
When present, invalid “clientData” format (not string)	400 (bad request)	invalid_request	Invalid parameter clientData
Invalid “hashes” element length	400 (bad request)	invalid_request	Invalid digest value length
Expired “SAD”	400 (bad request)	invalid_request	SAD expired
Expired credential	400 (bad request)	invalid_request	Signing certificate ‘O=[organization],CN=[common_name]’ is expired.
Document or documentDigest to be signed does not match one of the authorized hashes	403 (bad request)	invalid_hash	Document or documentDigest does not match authorized hash

Examples

Sample Request

```
POST /csc/v2/signatures/signDoc HTTP/1.1 Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
{
    "credentialID": "GX0112348",
    "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
    "documentDigests": [
        {
            "hashes": [ "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=" ],
            "signature_format": "P",
            "conformance_level": "AdES-B-T",
            "signAlgo": "1.2.840.113549.1.1.1"
        },
        {
            "hashes": [ "HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=" ],
            "signature_format": "C",
            "conformance_level": "AdES-B-B",
            "signAlgo": "1.2.840.113549.1.1.1"
        }
    ],
    "documents": [
        {
            "document": "Q2VydGlmaWNhdGVTZXJpYWxOdW1iZ…KzBTWWVJWWZZVXptU3V5MVU9DQo=",
            "documentType": "sod",
            "signature_format": "P",
            "conformance_level": "AdES-B-T",
            "signAlgo": "1.2.840.113549.1.1.1"
        },
        {
            "document": "Q2VydGlmaWNhdGVTZXJpYWxOdW1iZXI7U3… emNNbUNiL1cyQT09DQo=",
            "documentType": "sod",
            "signature_format": "C",
            "conformance_level": "AdES-B-B",
            "signed_envelope_property": "Attached",
            "signAlgo": "1.2.840.113549.1.1.1"
        }
    ],
    "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
    "clientData": "12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{
    "credentialID": "GX0112348",
    "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
    "documentDigests": [
        {
            "hashes": [ "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=" ],
            "signature_format": "P",
            "conformance_level": "AdES-B-T",
            "signAlgo": "1.2.840.113549.1.1.1"
        },
        {
            "hashes": [ "HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=" ],
            "signature_format": "C",
            "conformance_level": "AdES-B-B",
            "signAlgo": "1.2.840.113549.1.1.1"
        }
    ],
    "documents": [
        {
            "document": "Q2VydGlmaWNhdGVTZXJpYWxOdW1iZ…KzBTWWVJWWZZVXptU3V5MVU9DQo=",
            "documentType": "sod",
            "signature_format": "P",
            "conformance_level": "AdES-B-T",
            "signAlgo": "1.2.840.113549.1.1.1"
        },
        {
            "document": "Q2VydGlmaWNhdGVTZXJpYWxOdW1iZXI7U3… emNNbUNiL1cyQT09DQo=",
            "documentType": "sod",
            "signature_format": "C",
            "conformance_level": "AdES-B-B",
            "signed_envelope_property": "Attached",
            "signAlgo": "1.2.840.113549.1.1.1"
        }
    ],
    "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
    "clientData": "12345678"
}'
https://service.domain.org/csc/v2/signatures/signDoc
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
    "DocumentWithSignature":
    [
        "MILuLgYJKoZIhvcNAQcCoILuHz… ehEeR5ZRi5+WV5T1FpO",
        "MIL4IAYJKoZIhvcNAQcCoIL4…YavvBxkVwJ3dFD9KbCi1qW3TxTI="
    ],
    "SignatureObject":
    [
        "MIAGCSqAMIACAQExDzANBglghkgBZQMEAgEFADCABgkqhkiG…Ss4rEsQV4AAAAAAAAA==",
        "MIAGCSqGSIb3DQEHAqCAMIACAQExDzANBglghkgBZQMEqhki…W7pP1ZJFKuF2YAAAAAAA"
    ]
}
```

Sample Request

```
POST /csc/v2/signatures/signDoc HTTP/1.1 Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
{
    "signatureQualifier": "eu_eidas_qes",
    "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
  "documentDigests": [
    {
        "hashes": "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
        "signature_format": "P",
        "conformance_level": "AdES-B-T",
        "signAlgo": "1.2.840.113549.1.1.1"
    }
  ],
  "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
  "clientData": "12345678",
  "returnValidationInfo":true
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{
        "signatureQualifier": "eu_eidas_qes",
    "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
    "documentDigests": [
         {
            "hashes": "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
            "signature_format": "P",
            "conformance_level": "AdES-B-T",
            "signAlgo": "1.2.840.113549.1.1.1"
          }
  ],
  "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1",
  "clientData": "12345678",
  "returnValidationInfo":true }'
https://service.domain.org/csc/v2/signatures/signDoc
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
    "SignatureObject":
    [
        "MIAGCSqAMIACAQExDzANBglghkgBZQMEAgEFADCABgkqhkiG…Ss4rEsQV4AAAAAAAAA==",
        "MIAGCSqGSIb3DQEHAqCAMIACAQExDzANBglghkgBZQMEqhki…W7pP1ZJFKuF2YAAAAAAA"
    ],
    "validationInfo":{
        "ocsp":[
           "MIIJg...jSc="
        ],
        "crl":[
           "MIIC4...X7M="
        ]
        "certificates":[
           "<base64-encoded_X.509_certificate>"
        ]
    }
}
```

Sample Request

```
POST /csc/v2/signatures/signDoc HTTP/1.1 Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 6/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
{
    "signatureQualifier": "qes_eidas",
    "documentDigests": [
    {
      "hashes": [
        "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
        "HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0="
      ],
        "signature_format": "P",
      "conformance_level": "AdES-B-T",
      "signAlgo": "1.2.840.113549.1.1.1",
    },
  ],
    "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1"
    "clientData": "12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 6/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{  "signatureQualifier": "qes_eidas",
            "documentDigests": [
        {
          "hashes": [
            "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
            "HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0="
          ],
          "signature_format": "P",
          "conformance_level": "AdES-B-T",
          "signAlgo": "1.2.840.113549.1.1.1",
        },
      ],
            "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1"
            "clientData": "12345678" }'
https://service.domain.org/csc/v2/signatures/signDoc
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
{
    "SignatureObject":
    [
        "MIAGCSqAMIACAQExDzANBglghkgBZQMEAgEFADCABgkqhkiG…Ss4rEsQV4AAAAAAAAA==",
        "MIAGCSqGSIb3DQEHAqCAMIACAQExDzANBglghkgBZQMEqhki…W7pP1ZJFKuF2YAAAAAAA"
    ]
}
```

11.15 signatures/signPolling

Description

Request to the server to return the responses corresponding to previously sent (initial) digital signature value(s) or signature(s) creation request when processed in asynchronous mode.

If the user is authenticated directly by the RSSP then the *userID* is implicit and SHALL NOT be specified.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
requestID	REQUIRED	*String*	The *responseID* generated by the server to uniquely identify the response to a previous asynchronous signature request (see signatures/signHash and signatures/signDoc).
userID	REQUIRED Conditional	*String*	The *userID* as defined in the Input parameter table in credentials/list.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in oauth2/authorize.

Output

With HTTP status code 200 the method returns the following values using the “application/json” format:

Parameter	Presence	Value	Description
signatures	REQUIRED Conditional	*Array of String*	The *signatures* as defined in the Output attribute table in signatures/signHash. This element SHALL carry a value only if the client application requested the creation of digital signature value(s). This value SHALL be returned when the requested digital signature(s) creation has been completed.
DocumentWithSignature	REQUIRED Conditional	*Array of String*	The *DocumentWithSignature* as defined in the Output attribute table in signatures/signDoc. This value SHALL be returned when the requested signature(s) creation has been completed.
SignatureObject	REQUIRED Conditional	*Array of String*	The *SignatureObject* as defined in the Output attribute table in signatures/signDoc. This value SHALL be returned when the requested signature(s) creation has been completed.
validationInfo	REQUIRED Conditional	*JSON Object*	The *validationInfo* as defined in the Output attribute table in signatures/signDoc. This value SHALL be returned when the requested signature(s) creation has been completed and signatures/signDoc was called with the *returnValidationInfo* value set to "true".

With HTTP status code 202 the method indicates that the processing has not yet been completed.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	Malformed authorization header.
Missing or not String "requestID" Parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter requestID
Invalid requestID parameter	400 (bad request)	invalid_request	Invalid parameter requestID
Not empty "userID" parameter in case of user-specific authorization	400 (bad request)	invalid_request	userID parameter SHALL be null
Invalid "userID" format in case of no user-specific authorization	400 (bad request)	invalid_request	Invalid parameter "userID"
When present, invalid "clientData" format (not string)	400 (bad request)	invalid_request	Invalid parameter clientData

Examples

Sample Request

```
POST /csc/v2/signatures/signPolling  HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "requestID":"158112-652341-khj",
   "clientData":"12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{ "requestID":"158112-652341-khj",
           "clientData": "12345678" }'
     https://service.domain.org/csc/v2/signatures/signPolling
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
    "signatures":
    [
        "KedJuTob5gtvYx9qM3k3gm7kbLBwVbEQRl26S2tmXjqNND7MRGtoew==",
        "Idhef7xzgtvYx9qM3k3gm7kbLBwVbE98239S2tm8hUh85KKsfdowel=="
    ]

}
```

11.16 signatures/timestamp

Description

Generate a time-stamp token for the input hash value. The time-stamp token can be generated directly by the RSSP or by a Time Stamping Authority connected to it.

The reason to implement this method instead of providing time-stamp services through widespread RFC 3161 [2] protocols directly is to facilitate the creation of long-term validation digital signatures and to support billing operations. In both cases, the RSSP provider can offer pre-configured time-stamp services instead of requiring the signature application to obtain time-stamp services from a different provider.

Input

This method allows the following parameters:

Parameter	Presence	Value	Description
hash	REQUIRED	*String*	The base64-encoded hash value to be time stamped. The remote service SHALL use this value to encode the value of MessageImprint.hashedMessage as defined in RFC 3161 [2].
hashAlgo	REQUIRED	*String*	The OID of the algorithm used to calculate the hash value. The remote service SHALL use this value to encode the value of MessageImprint.hashAlgorithm as defined in RFC 3161 [2].
nonce	OPTIONAL	*String*	A large random number with a high probability that it is generated by the signature application only once. The value SHALL be represented as hex-encoded string.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in oauth2/authorize.

Note 47: RFC 3161 [2] contains more detailed definitions of time stamp parameters that can be used in the context of this specification.

Output

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

Parameter	Presence	Value	Description
timestamp	REQUIRED	*String*	The base64-encoded time-stamp token as defined in RFC 3161 [2] as updated by RFC 5816 [10]. If the *nonce* parameter is included in the request then it SHALL also be included in the time-stamp token, otherwise the response SHALL be rejected.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	Malformed authorization header.
The“hash” parameter is missing or not of type String.	400 (bad request)	invalid_request	Missing (or invalid type) string parameter hash
Empty hash parameter	400 (bad request)	invalid_request	Empty hash parameter
Invalid “hash” length	400 (bad request)	invalid_request	Invalid digest value length
Invalid base64 hash element	400 (bad request)	invalid_request	Invalid Base64 hash string parameter
Invalid “hashAlgo” parameter	400 (bad request)	invalid_request	Invalid parameter hashAlgo
Invalid or non-numeric “nonce” parameter	400 (bad request)	invalid_request	Invalid parameter nonce

Examples

Sample Request

```
POST /csc/v2/signatures/timestamp HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
   "hash":"sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
   "hashAlgo":"2.16.840.1.101.3.4.2.1",
   "clientData":"12345678"
}
```

cURL Example

```
curl -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{ "hash": "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
           "hashAlgo": "2.16.840.1.101.3.4.2.1",
           "clientData": "12345678" }'
     https://service.domain.org/csc/v2/signatures/timestamp
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{

"timestamp":"MGwCAQEGCSsGAQQB7U8CATAxMA0GCWCGSAFlAwQCAQUABCCrCqnrjH0VxXyQQlfnFJRx1jjrviTs7/GjKghr2Amlu
QIIVs5D8OUB4p4YDzIwMTQxMTE5MTEzMjM5WjADAgEBAgkAnWn2SSIWlXk="
}
```

12 OpenAPI description

An OpenAPI 3.0 description file is provided, as defined by the OpenAPI Initiative (OAI) https://www.openapis.org, containing a full description of the CSC API. The OpenAPI file contains:

1. General information about the API like, for example, the APIs version, the Cloud Signature Consortium contact information;
2. Information about the RESTful path URL and an example of server URL access points;
3. Authorization schemas required to access the CSC API;
4. A description of every method of the CSC protocol including input objects and returned HTTP responses.

The OpenAPI description file can be used by developers or testers to automatically generate a CSC compliant server interfaces or client stubs.

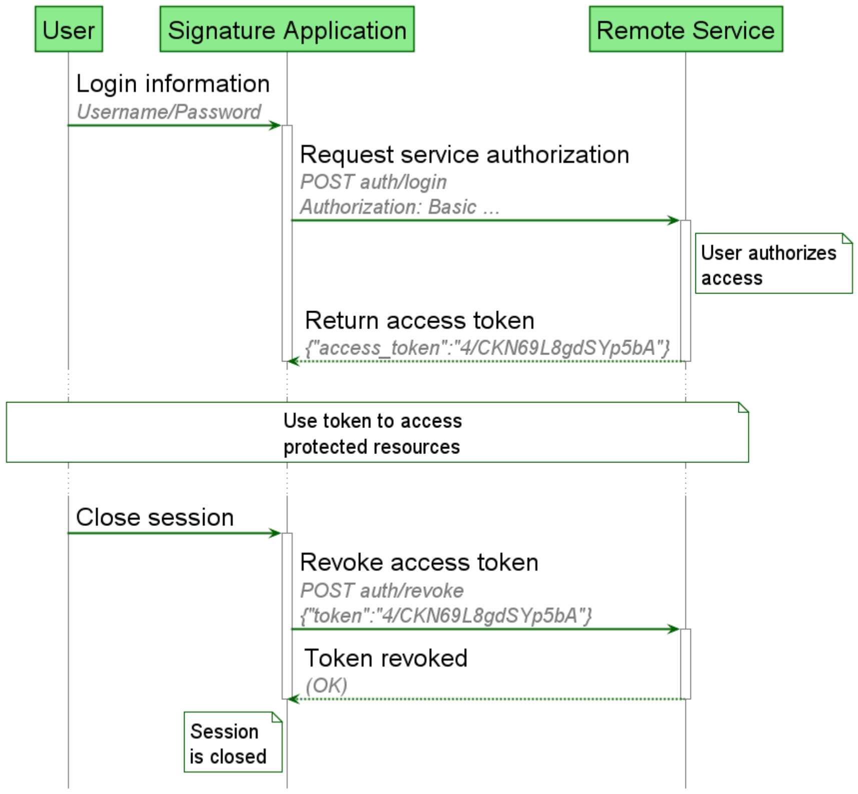
The OpenAPI description file is available from the website of the Cloud Signature Consortium at: https://cloudsignatureconsortium.org/resources/download-api-specifications/.

13 Interaction among elements and components

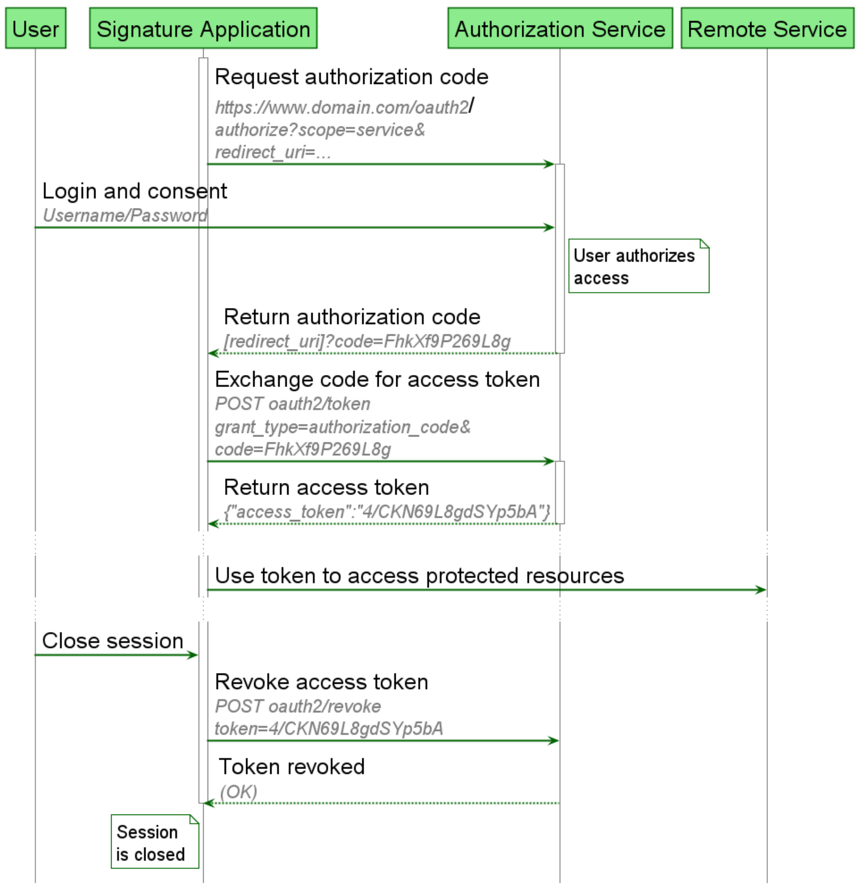
The building blocks of a remote signature solution interact with the API methods described in this specification. The following sections describe the sequence diagrams of some of the most common operations required to obtain a service authorization, credential authorization and to request a remote signature.

Note 48: The sample requests and responses that are provided in the diagrams are only a partial representation of complete transactions and are aimed at showing the most important parameters and information. See the example in the previous sections of this specification for complete and detailed descriptions.

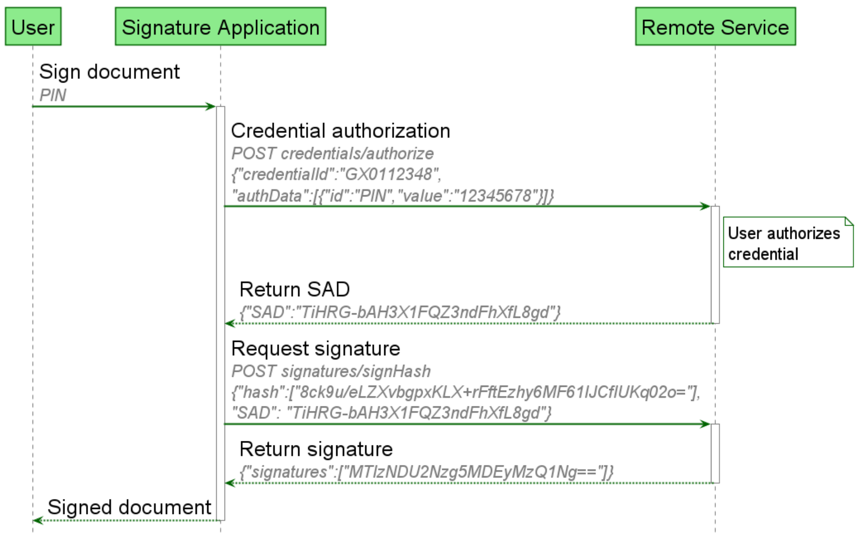
13.1 Remote signing service authorization using Basic Authentication



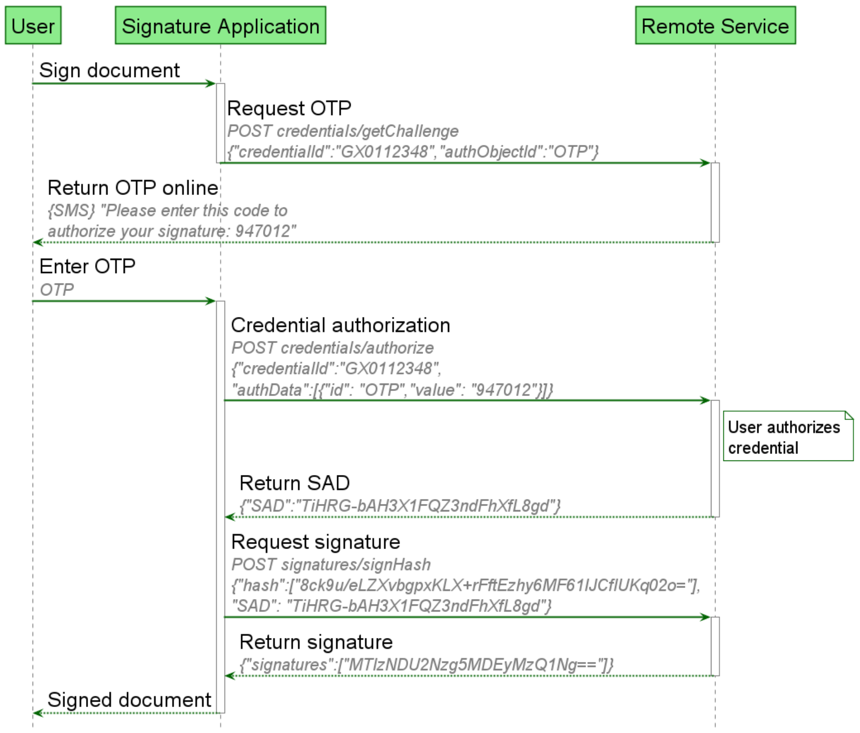
13.2 Remote signing service authorization using OAuth 2.0 with Authorization Code flow



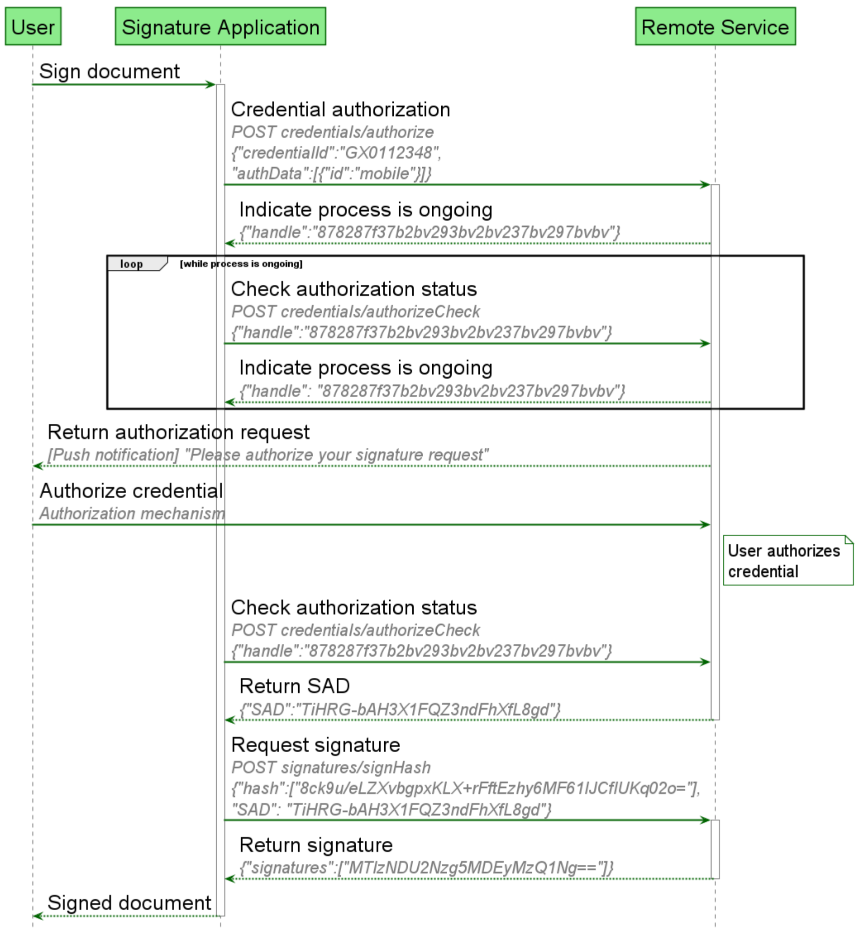
13.3 Create a remote signature with a credential protected by a PIN


User
Signature Application
Remote Service
Sign document
PIN
Credential authorization
POST credentials/authorize
{"credentialId":"GX0112348",
"authData":[{"id":"PIN","value":"12345678"}]}
User authorizes
credential
Return SAD
{"SAD":"TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Request signature
POST signatures/signHash
{"hash":["8ck9u/eLZXvbgpxKLX+rFftEzhy6MF61lJCfIUKq02o="],
"SAD": "TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Return signature
{"signatures":["MTIzNDU2Nzg5MDEyMzQ1Ng=="]}
Signed document

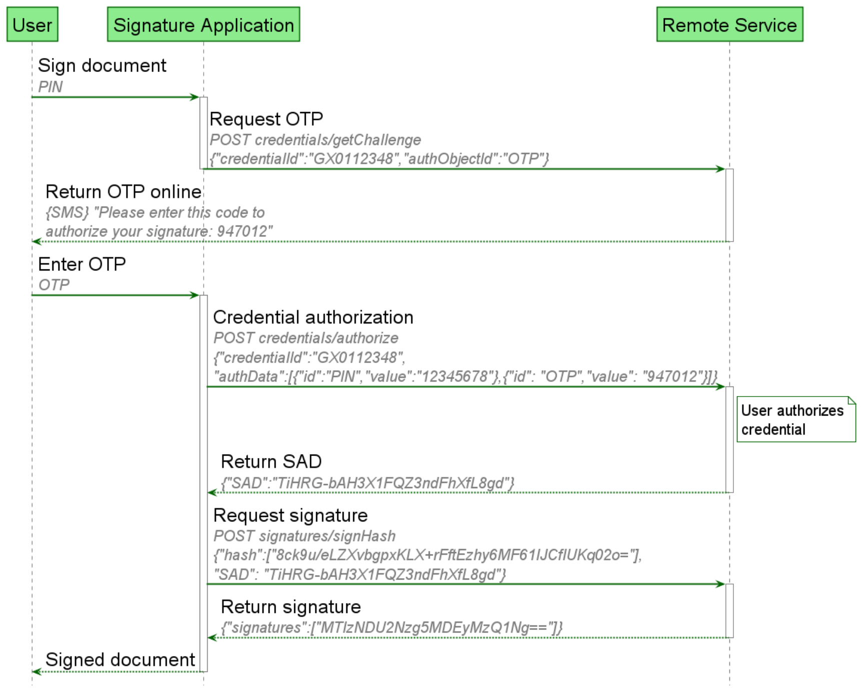
13.4 Create a remote signature with a credential protected by an “online” OTP (based on SMS)


User
Signature Application
Remote Service
Sign document
Request OTP
POST credentials/getChallenge
{"credentialId":"GX0112348","authObjectId":"OTP"}
Return OTP online
{SMS} "Please enter this code to authorize your signature: 947012"
Enter OTP
OTP
Credential authorization
POST credentials/authorize
{"credentialId":"GX0112348", "authData":[{"id": "OTP","value": "947012"}]}
User authorizes credential
Return SAD
{"SAD":"TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Request signature
POST signatures/signHash
{"hash":["8ck9u/eLZXvbgpxKLX+rFftEzhy6MF61IJCfIUKq02o="], "SAD": "TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Return signature
{"signatures":["MTIzNDU2Nzg5MDEyMzQ1Ng=="]}
Signed document

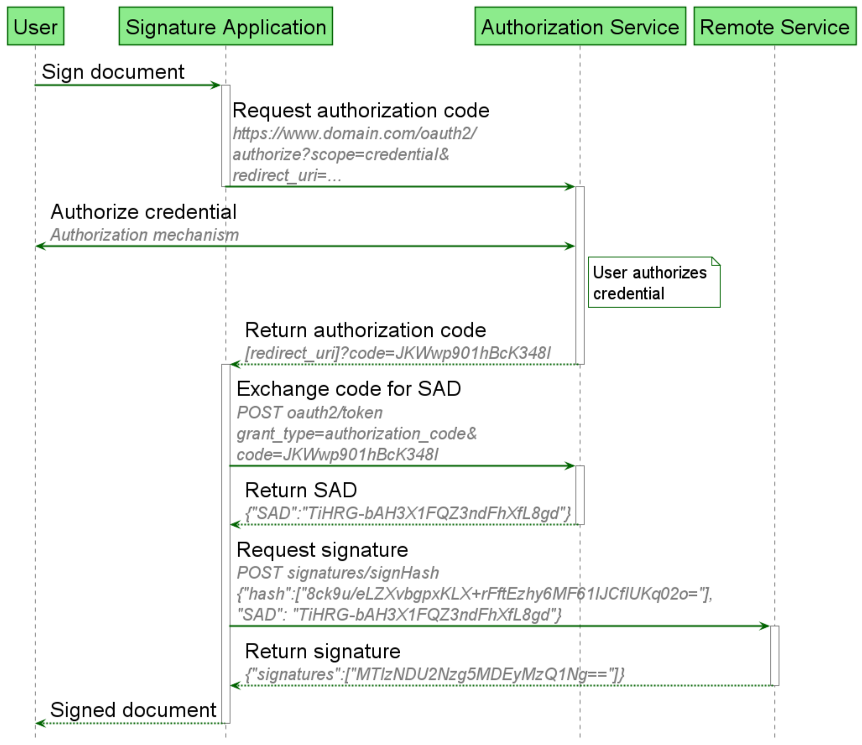
13.5 Create a remote signature with a credential protected by a mobile app


User
Signature Application
Remote Service
Sign document
Credential authorization
POST credentials/authorize
{"credentialId":"GX0112348",
"authData":[{"id":"mobile"}]}
Indicate process is ongoing
{"handle":"878287f37b2bv293bv2bv237bv297bvbv"}
loop
[while process is ongoing]
Check authorization status
POST credentials/authorizeCheck
{"handle":"878287f37b2bv293bv2bv237bv297bvbv"}
Indicate process is ongoing
{"handle": "878287f37b2bv293bv2bv237bv297bvbv"}
Return authorization request
[Push notification] "Please authorize your signature request"
Authorize credential
Authorization mechanism
User authorizes credential
Check authorization status
POST credentials/authorizeCheck
{"handle":"878287f37b2bv293bv2bv237bv297bvbv"}
Return SAD
{"SAD":"TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Request signature
POST signatures/signHash
{"hash":["8ck9u/eLZXvbgpxKLX+rFftEzhy6MF61IJCfIUKq02o="],
"SAD": "TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Return signature
{"signatures":["MTIzNDU2Nzg5MDEyMzQ1Ng=="]}
Signed document

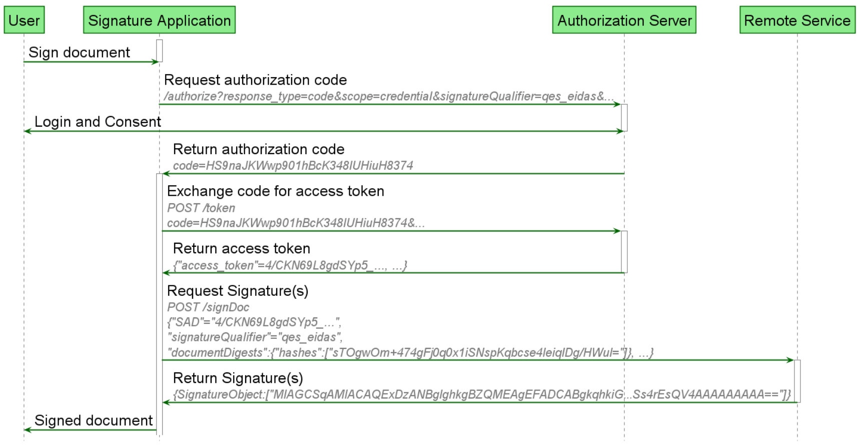
13.6 Create a remote signature with a credential protected by a PIN and an "online" OTP (based on SMS)


User
Signature Application
Remote Service
Sign document
PIN
Request OTP
POST credentials/getChallenge
{"credentialId":"GX0112348","authObjectId":"OTP"}
Return OTP online
{SMS} "Please enter this code to
authorize your signature: 947012"
Enter OTP
OTP
Credential authorization
POST credentials/authorize
{"credentialId":"GX0112348",
"authData":[{"id":"PIN","value":"12345678"},{"id": "OTP","value": "947012"}]}
User authorizes credential
Return SAD
{"SAD":"TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Request signature
POST signatures/signHash
{"hash":["8ck9u/eLZXvbgpxKLX+rFtEzhy6MF61lJCflUKq02o="],
"SAD": "TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Return signature
{"signatures":["MTIzNDU2Nzg5MDEyMzQ1Ng=="]}
Signed document

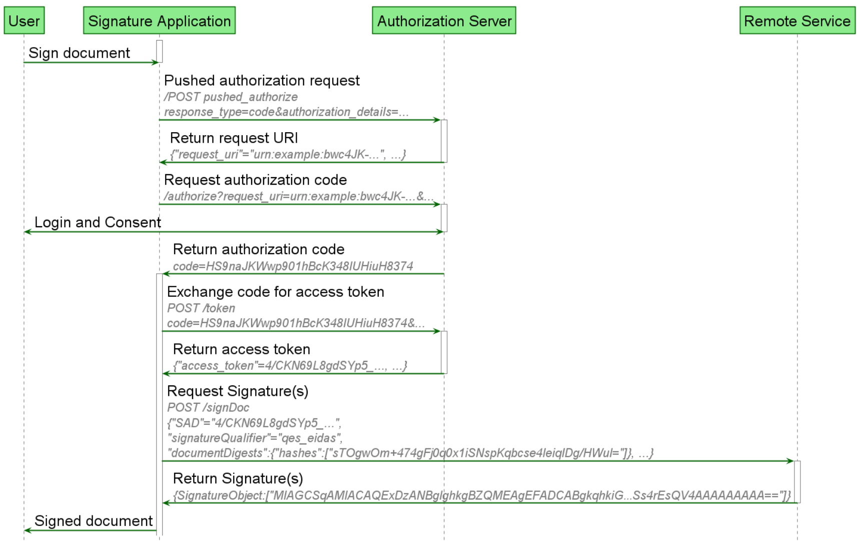
13.7 Create a remote signature with a credential protected by OAuth 2.0 with Authorization Code flow


User
Signature Application
Authorization Service
Remote Service
Sign document
Request authorization code
https://www.domain.com/oauth2/
authorize?scope=credential&
redirect_uri=...
Authorize credential
Authorization mechanism
User authorizes credential
Return authorization code
[redirect_uri]?code=JKWwp901hBcK348l
Exchange code for SAD
POST oauth2/token
grant_type=authorization_code&
code=JKWwp901hBcK348l
Return SAD
{"SAD":"TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Request signature
POST signatures/signHash
{"hash":["8ck9u/eLZXvbgpxKLX+rFtEzhy6MF61lJCflUKq02o="],
"SAD": "TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Return signature
{"signatures":["MTIzNDU2Nzg5MDEyMzQ1Ng=="]}
Signed document

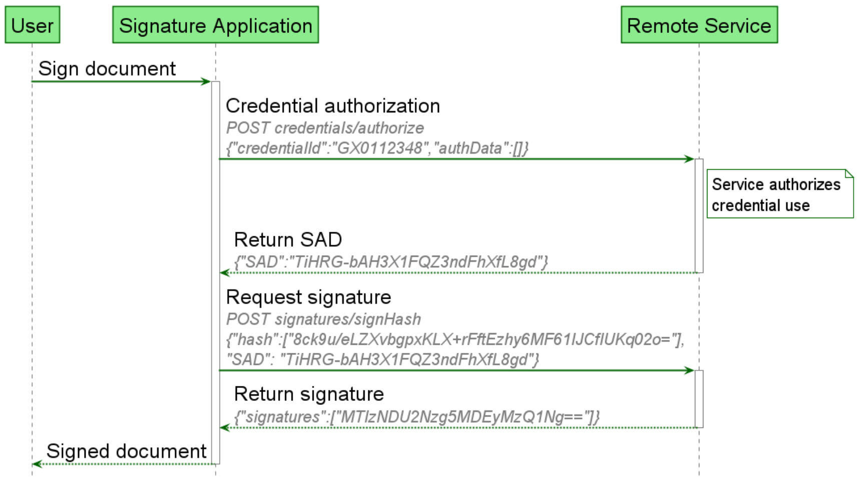
13.8 Create a remote signature with credential and signature qualifier with OAuth 2.0 Authorization Code flow


User
Signature Application
Authorization Server
Remote Service
Sign document
Request authorization code
/authorize?response_type=code&scope=credential&signatureQualifier=qes_eidas&...
Login and Consent
Return authorization code
code=HS9naJKWwp901hBcK348IUHiuH8374
Exchange code for access token
POST /token
code=HS9naJKWwp901hBcK348IUHiuH8374&...
Return access token
{"access_token"=4/CKN69L8gdSYp5_..., ...}
Request Signature(s)
POST /signDoc
{"SAD"="4/CKN69L8gdSYp5_...",
"signatureQualifier"="qes_eidas",
"documentDigests":{"hashes":["sTOgwOm+474gFj0q0x1iSNspKqbcse4leiqlDg/HWuI="]}, ...}
Return Signature(s)
{SignatureObject:["MIAGCSqAMIACAQExDzANBglghkgBZQMEAgEFADCABgkqhkiG...Ss4rEsQV4AAAAAAAAA=="]}
Signed document

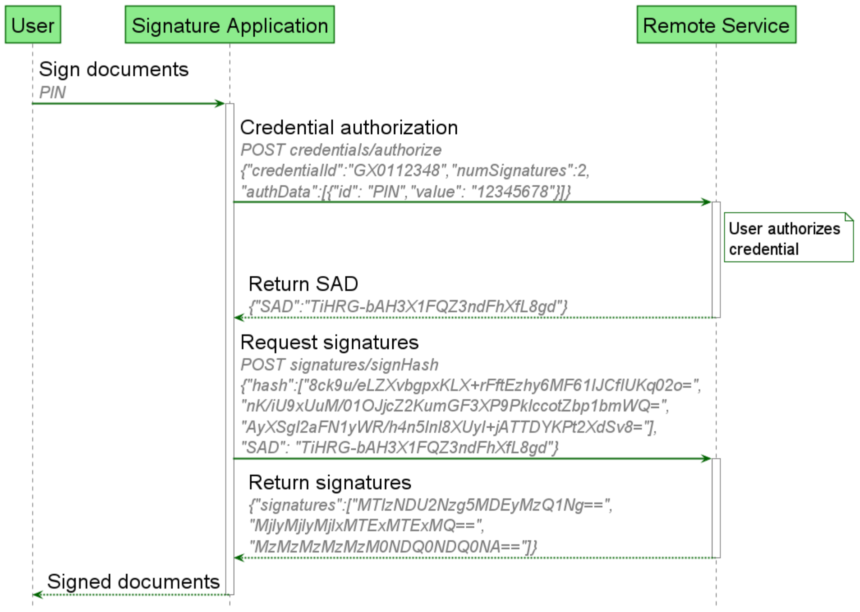
13.9 Create a remote signature with OAuth 2.0 Authorization Code flow and Pushed and Rich Authorization Request


User
Signature Application
Authorization Server
Remote Service
Sign document
Pushed authorization request
/POST pushed_authorize
response_type=code&authorization_details=...
Return request URI
{"request_uri"="urn:example:bwc4JK-...", ...}
Request authorization code
/authorize?request_uri=urn:example:bwc4JK-...&...
Login and Consent
Return authorization code
code=HS9naJKWwp901hBcK348IUHiuH8374
Exchange code for access token
POST /token
code=HS9naJKWwp901hBcK348IUHiuH8374&...
Return access token
{"access_token"=4/CKN69L8gdSYp5_..., ...}
Request Signature(s)
POST /signDoc
{"SAD"="4/CKN69L8gdSYp5_...",
"signatureQualifier"="qes_eidas",
"documentDigests":{"hashes":["sTOgwOm+474gFj0q0x1iSNspKqbcse4leiqlDg/HWuI="]}, ...}
Return Signature(s)
{SignatureObject:["MIAGCSqAMIACAQExDzANBglghkgBZQMEAgEFADCABgkqhkiG...Ss4rEsQV4AAAAAAAAA=="]}
Signed document

13.10 Create a remote signature with a credential protected by RSSP-managed authorization

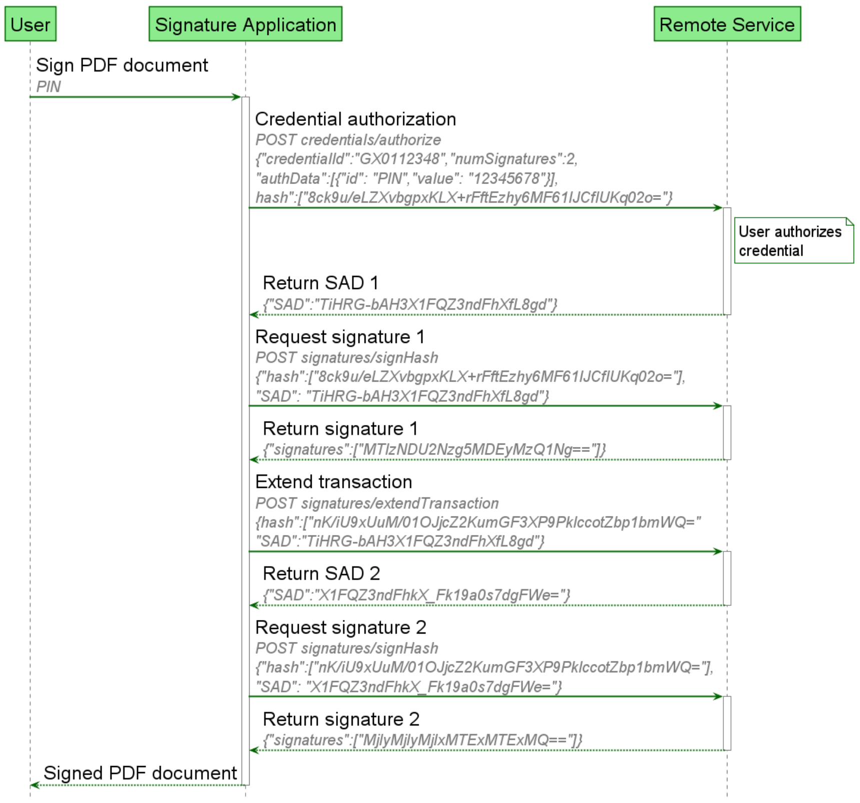

User
Signature Application
Remote Service
Sign document
Credential authorization
POST credentials/authorize
{"credentialId":"GX0112348","authData":[]}
Service authorizes credential use
Return SAD
{"SAD":"TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Request signature
POST signatures/signHash
{"hash":["8ck9u/eLZXvbgpxKLX+rFftEzhy6MF61lJCflUKq02o="],
"SAD": "TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Return signature
{"signatures":["MTIzNDU2Nzg5MDEyMzQ1Ng=="]}
Signed document

13.11 Create multiple remote signatures from a list of hash values


User
Signature Application
Remote Service
Sign documents
PIN
Credential authorization
POST credentials/authorize
{"credentialId":"GX0112348","numSignatures":2,
"authData":[{"id": "PIN","value": "12345678"}]}
User authorizes credential
Return SAD
{"SAD":"TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Request signatures
POST signatures/signHash
{"hash":["8ck9u/eLZXvbgpxKLX+rFftEzhy6MF61lJCflUKq02o=",
"nK/iU9xUuM/01OJjcZ2KumGF3XP9PklccotZbp1bmWQ=",
"AyXSgI2aFN1yWR/h4n5lnl8XUyl+jATTDYKPt2XdSv8="],
"SAD": "TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Return signatures
{"signatures":["MTIzNDU2Nzg5MDEyMzQ1Ng==",
"MjlyMjlyMjlxMTExMTExMQ==",
"MzMzMzMzM0NDQ0NDQ0NA=="]}
Signed documents

13.12 Create a remote multi-signatures transaction with a PDF document

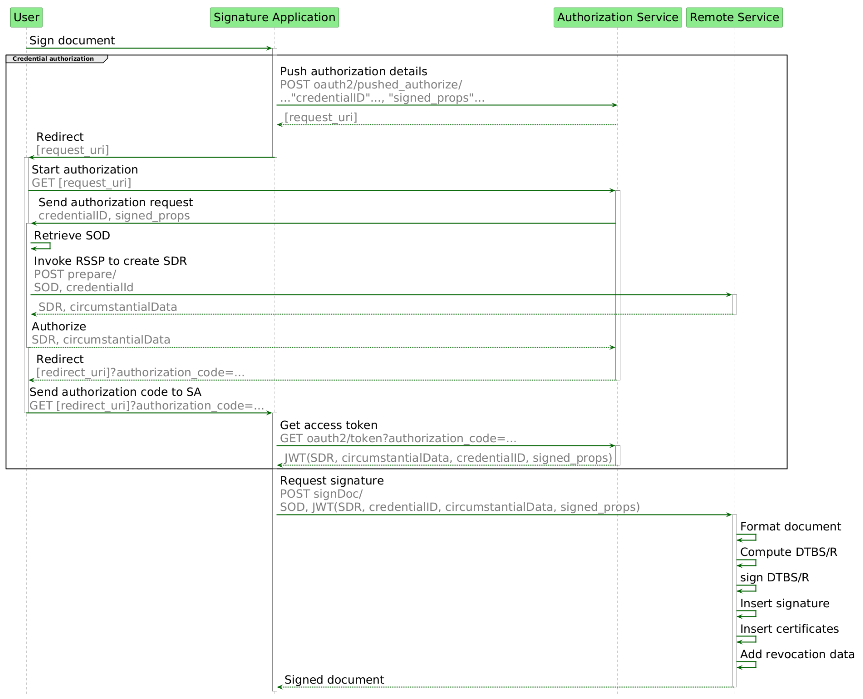
This diagram shows the case of a PDF document that is signed multiple times by the same signer. A single credential authorization can be performed to authorize multiple signatures. However only the initial hash of the document is available at authorization time. A new hash will be generated to calculate the following signatures. For this reason, the **credentials/extendTransaction** method is used to supply the new hash to obtain the SAD to calculate a new signature. See credentials/extendTransaction for more information.


User
Signature Application
Remote Service
Sign PDF document
PIN
Credential authorization
POST credentials/authorize
{"credentialId":"GX0112348","numSignatures":2,
"authData":[{"id": "PIN","value": "12345678"}],
hash":["8ck9u/eLZXvbgpxKLX+rFftEzhy6MF61IJCfIUKq02o="}
User authorizes credential
Return SAD 1
{"SAD":"TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Request signature 1
POST signatures/signHash
{"hash":["8ck9u/eLZXvbgpxKLX+rFftEzhy6MF61IJCfIUKq02o="],
"SAD": "TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Return signature 1
{"signatures":["MTIzNDU2Nzg5MDEyMzQ1Ng=="]}
Extend transaction
POST signatures/extendTransaction
{hash":["nK/iU9xUuM/01OJjcZ2KumGF3XP9PklccotZbp1bmWQ="
"SAD":"TiHRG-bAH3X1FQZ3ndFhXfL8gd"}
Return SAD 2
{"SAD":"X1FQZ3ndFhkX_Fk19a0s7dgFWe="}
Request signature 2
POST signatures/signHash
{"hash":["nK/iU9xUuM/01OJjcZ2KumGF3XP9PklccotZbp1bmWQ="],
"SAD": "X1FQZ3ndFhkX_Fk19a0s7dgFWe="}
Return signature 2
{"signatures":["MjlyMjlyMjlxMTExMTExMQ=="]}
Signed PDF document

13.13 Authorize SDR for signDoc

When calling signatures/signDoc the RSSP may require that SDR has been authorized. Since SDR cannot be computed deterministically from signer's original document, but depends on choices made in the RSSP, the signature application will need assistance from the RSSP to get authorization for SDR.

This example shows a possible approach that uses advanced features of the authorization server that invokes or lets the user agent invoke a remote service to compute SDR as part of the authorization flow. This approach avoids involving the signature application in the computation of SDR.


User
Signature Application
Authorization Service
Remote Service
Sign document
Credential authorization
Push authorization details
POST oauth2/pushed_authorize/
..."credentialID"..., "signed_props"...
[request_uri]
Redirect
[request_uri]
Start authorization
GET [request_uri]
Send authorization request
credentialID, signed_props
Retrieve SOD
Invoke RSSP to create SDR
POST prepare/
SOD, credentialId
SDR, circumstantialData
Authorize
SDR, circumstantialData
Redirect
[redirect_uri]?authorization_code=...
Send authorization code to SA
GET [redirect_uri]?authorization_code=...
Get access token
GET oauth2/token?authorization_code=...
JWT(SDR, circumstantialData, credentialID, signed_props)
Request signature
POST signDoc/
SOD, JWT(SDR, credentialID, circumstantialData, signed_props)
Format document
Compute DTBS/R
sign DTBS/R
Insert signature
Insert certificates
Add revocation data
Signed document

CLOUD SIGNATURE CONSORTIUM

14 Change history

14.1 Changes since version 1.0.4.0

- Add certificate info into credentials/list method: It is now allowed to provide directly in the credentials/list method the detailed information of the certificates.
- Add asymmetric signing: The possibility was added to use asynchronous call as was already proposed in ETSI TS 119 432.
- Add signing of documents: It is not only possible to create a cryptographic signature of a hash, but also to create an AdES signature on a hash or a document. The functionality is more powerful than the one introduced in ETSI TS 119 432 since it allows different signature formats for different documents within one call, which is especially useful in case a certificate is only created for one signature authorization, and this authorization should cover different types of documents. It also allows to request JAdES signatures.
- Possibility to use authorization request payload (PAR) and rich authorization requests (RAR) in the OAuth authorization.
- Allow to use only the credential OAuth authorization (without service authorization) for signing.
- Add a chapter on the usage of the CSC protocol for creating electronic seals.
- When creating a PAdES signature based on the hash document, provide the revocation information so that this can be included in the final signed document.
- Allow to request only credential which are valid, i.e. which can be used to sign, in the credentials/list endpoint
- Add explanation how to define algorithms via OIDs.
- Allow to request signature authorization via OAuth on a credential of a specific type, without specifying the credential ID. This is useful for short-lived credentials which are only created for a specific signature

process.
- Make explicit credential authorization more flexible: The explicit credential authorization allows to use and combine different authorization types. This makes the implicit credential authorization useless, because it can be expressed as part of the explicit credential authorization.
- Each time hash values are provided, provide also the hash algorithm

14.2 Changes since version 2.0.0.2

- Amendments to CSC version 2.0.0.2.
- Add a table to clarify input/output of signatures/signDoc.
- Add signed attributes to OAuth 2.0 authorization.
- Provide OpenAPI schema file for v2.1.
- Deprecate JSON schema file format and credentials/sendOTP.
- Add possibility to support multiple OAuth 2.0 authorization servers.

14.3 Changes since version 2.1.0.1

- Credential Management: Added endpoints for creating and deleting credentials. Improved authorization handling and updated the credential information endpoint. Introduced new data model types for credential creation and deletion requests.
- Data Model Alignment: Integrated the Data Model v1.0.0. Reorganized and simplified the model structure. Moved binding sections to separate documentation. Added reusable and specific data types to enhance consistency.
- Architectural Updates: Improved the explanation of core architectures. Introduced a clear definition for the Signature Creation Application. Refined the wallet-centric model and separated RP and DA roles. Improved section 8 of the API to align with the updated architecture.
- QES Transaction and Signing: Expanded support for QES transaction data and bindings. Clarified QES data content and the use of signature qualifiers. Added an alternative response channel for QES using responseURI. Introduced new transaction types, including certificate issuance.
- Encoding and Data Format Improvements: Standardized Base64 and Base64URL encoding across the specification. Added format patterns for URIs, byte data, and enums. Introduced a dedicated hash data type to ensure consistent cryptographic references.
- Documentation and Layout: Added a new Data Model documentation set. Improved rendering and readability with Mermaid diagrams, better font support, and searchable PDFs. Added diagrams, examples, and acknowledgements. Updated examples to reflect new architectural and data model definitions.