CLOUD SIGNATURE CONSORTIUM Standard

Architectures and protocols for remote signature applications

Published version 1.0.3.0 (2018-12)

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

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, Andreas Vollmert, Arno Fiedler, Bernd Wild, Carlos Ares, Cornelia Enke, David Ruana, Davide Barelli, Enrico Entschew, Francesco Barcellini, Franck Leroy, Giuliana Marzola, Giuseppe Damiano, Håvard Grindheim, Iñigo Barreira, Jon Ølnes, Kapil Khattar, 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, 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.

Legal notices

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

The present document does not create legal rights and does not imply that intellectual property rights are transferred to the recipient or other third parties. The adoption of the specification contained herein does not constitute any rights of affiliation or membership to the Cloud Signature Consortium VZW.

This document is provided “as is” and the Cloud Signature Consortium, its members and the individual contributors, are not responsible for any errors or omissions.

The Trademark and Logo of the Cloud Signature Consortium are registered, and their use is reserved to the members of the Cloud Signature Consortium VZW. Questions and comments on this document can be sent to info@cloudsignatureconsortium.org .

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] IETF RFC 7521: "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants"

[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-10: "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".

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

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].

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 a X.509 public key certificate managed by a remote signing service provider on behalf of a user.

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 4: 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 5: A remote signing service typically manages signing keys and certificates that are created before the signing operations take place. A common scenario is also when the signing key and the certificate are created in the course of a signing operation (also called “ad-hoc” or “on-the-go” credentials). It is possible to operate ad-hoc credentials with this specification by creating the signing key and the certificate just before accessing them. Methods to create ad-hoc credentials during the authorization or the signature operations will be handled in a future release of this specification.

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

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 6: 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.

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

4.2 Abbreviations

API: application programming interface

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

SCAL2: sole control assurance level 2

NOTE 2: As defined in CEN EN 419 241-1 [i.5].

5 Conventions

5.1 Text conventions

This specification adopts the following text conventions to help identify various types of information.

Table 1 – Text conventions

Text convention	Example
The vertical bar (|) indicates a possible value for selection or outcome and SHALL be interpreted as "or".	YES | NO
Text in colored boxes is example code.	`POST /csc/v1/credentials/info HTTP/1.1`
Bold text indicates the name of an API method.	**credentials/list**
Italic text indicates the name of an API input or output parameter.	*access_token*

In general, API names as well as API input or output parameters defined in this specification use the "camelCase" notation, like *authType* or **credentials/extendTransaction**. However, names and parameters that are defined in other standards, like those in the domain of authentication and related to OAuth 2.0, are used here in their original format to facilitate understanding and interoperability, using "snake_case", like *refresh_token* e.g. two names separated by an underscore.

5.2 Base-64

When data is required to be Base64-encoded, it SHALL be encoded as "base64" as defined in RFC 4648 [6]. To avoid JSON representation issues line breaks SHALL NOT be used within Base64-encoded data. When data is base64url-encoded it SHALL be encoded as "base64url" as defined in RFC 4648 [6].

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], whereas §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 specifications focuses on the interface between the Signature Application and the remote signing service provider. The following figure shows a typical but not restrictive example of the architecture.

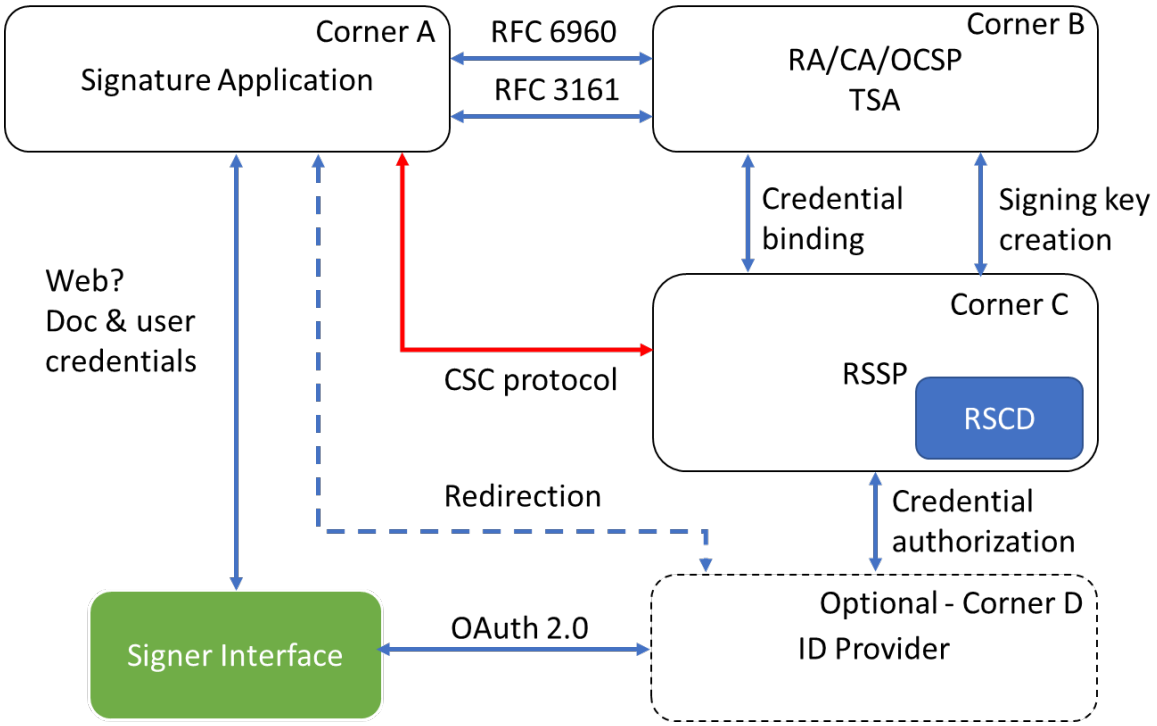


Figure 1: Remote signing corners

There are four main corners in the remote signing scenario.

The Signature Application retrieves the document to be signed from the user, and, if needed the certificates, revocation information and time-stamps from the corresponding trust service provider. It requests the remote signing service provider to create the signature of the hash value.

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.

The authorization for service or credential access can be done either passing through the signature application or using a redirection to an external OAuth 2.0 authorization server (AS). In many cases, the authorization server is part of the RSSP.

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 Section 12.

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 `v1` . Future versions of this specification MAY not be completely retro-compatible.

```
https://service.domain.org/csc/v1/
```

The `service.domain.org` hostname is used in this specification as an example and it SHOULD be replaced with a hostname registered by the remote signing service provider. 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 section 8.3, which MAY use URIs that are independent from 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 TSL, 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.

8 Authentication and authorization

This specification supports two types of authentication and authorization:

a. Service authorization and authentication.
b. Credential authorization.

8.1 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 section 11.1.

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 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.

Two methods are defined in this specification to obtain an access token to authorize the access to the remote service API:

- 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 Section 8.3 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 the authorization, the remote service will return a service access token to the signature application. From then on, all authenticated requests to API methods SHALL use an Authorization header with *Bearer* type followed by the service access token.

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

8.2 Credential authorization

Accessing a credential for remote signing requires an authorization from the user who owns it to control the signing key associated to it.

The RSSP 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 RSSP, 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 regards 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.

Three different types of credential authorization are defined and supported in this specification:

- Explicit authorization
- Implicit authorization
- OAuth 2.0 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 section 11.5. This method returns the type, format and combination of required or optional authentication factors, such that the signature application could show the proper interactive controls to collect them from the user.

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 could 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. PIN and OTP are supported directly in this specification and can be used in combination 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].

Implicit authorization means that the remote service is taking care of the authorization process autonomously, by engaging with the user without any intermediation from the signature application. In this case, the signature application will invoke the credential authorization methods without passing any authentication factors, as these would be implicitly managed by the remote service directly with the user.

For example, the RSSP can support SCAL2 as defined in CEN 419 241-1 [i.5] using implicit authorization providing a completely independent two-factor authorization mechanism that does not require any user interaction to occur within the signature application.

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.

8.3 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 show a web page managed by the remote service where the user will be authenticated according to the specific mechanism implemented there. After a successful authentication, the authorization server of the remote service will return an authorization code or an access token to the signature application. This access token will be used later to authorize access to the remote service's resources.

This specification supports the following types of OAuth 2.0 flows as described in RFC 6749 [11]:

- Authorization Code flow
- Client Credentials flow

The use of the Implicit Grant Flow is explicitly forbidden in this specification due to security flaws.

Any provider implementing an OAuth 2.0 authorization flow is strongly advised to follow the recommendations from OAuth 2.0 Security Best Current Practice [20].

OAuth 2.0 authorization mechanisms can be used for both Service and Credentials authorization. A remote service can therefore implement a single OAuth 2.0 authorization server supporting two different scopes for “service” and “credential” authorization.

Before using an OAuth 2.0 authorization mechanism, the signature application SHALL obtain from the remote service the client credentials (a Client ID and conditionally a Client Secret) and register one or more Redirect URI address with it. The means through which the signature application obtains these information from the remote service are beyond the scope of this specification.

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 section 8.3.3, 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 section 8.3.4, if supported. Tokens MAY also be revoked by calling the remote service’s **auth/revoke** method, as defined in section 11.3, if supported.

The **info** method, as defined in section 11.1, specifies a base URI for all OAuth 2.0 endpoints. The URI path components of the supported OAuth 2.0 API methods specified in sections 8.3.2, 8.3.3, and 8.3.4 SHALL be concatenated to the OAuth 2.0 base URI.

8.3.1 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 section 8.3.2.

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

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

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

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

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

JWT_Header

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

JWT_Payload

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

JWT_Signature

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

Parameters

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

Implementation notes

- The RSSP SHALL securely share the OAuth 2.0 *client_id* and *client_secret* with the signature application as part of the OAuth 2.0 configuration (see section 8.3).
- 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.eyJzdWIiOiI3S1lCckpBLWtCOT
F5T1Rld1JZRzh5SGdzN3EtbzR1NiIsImlhdCI6MTUzNzAxMjgwMCwianRpIjoiYjgzZmY4OWEtZWQzZ
i00NjgxLTgyOGQtNzE2MGI5MTNjYTcyIiwiaXNzIjoiQ1NDIFNpZ25hdHVyZSBBcHBsaWNhdGlvbiIs
ImF6cCI6ImE4NzliNDE5LThmZWQtNDcyZS05Yzk3LTJmODk3NTIxODU3ZSJ9.SEwD3KGDPFX-
8IIJE7pC_RJ-0wdOVinEPTHmKKVQb6E&…
```

8.3.2 oauth2/authorize (OAuth 2.0 Authorization Code)

Description: Starts the OAuth 2.0 authorization server using an Authorization Code flow, as described in Section 1.3.1 of RFC 6749 [11], to request authorization for the user to access the remote service resources. The authorization is returned in the form of an authorization code, which the signature application SHALL then use to obtain an access token with the **oauth2/token** method. The authorization server SHOULD support two access token scopes: “service” and “credential”. These scopes SHALL only be used separately to obtain an access token suitable for service and credential authorization respectively.

At the end of the authorization process, 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, which SHALL be pre-registered with the remote service by the signature application to avoid abuse by unauthorized clients.

NOTE 1: **oauth2/authorize** does not specify a regular CSC API method, but rather the URI path component of the address of the web page allowing the user to sign-in to the remote service to authorize the signature application or to authorize a credential. The complete URL to invoke the OAuth 2.0 authorization server is obtained by adding **oauth2/authorize** to the base URI of the authorization server as returned in the *oauth2* parameter by the **info** method, as defined in section 11.1, and it does not necessarily include the base URI of the remote service API.

NOTE 2: 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: In order to maintain full compatibility with the OAuth 2.0 standard, the following parameters SHALL be passed as a query string with the authorization endpoint URI using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body.

NOTE 3: The list of parameters is split between standard parameters that are defined in the OAuth 2.0 standard (see RFC 6749 [11]) and parameters that are defined in this specification. These parameters SHALL be combined in a single query string.

NOTE 4: Although RFC 3986 [3] doesn’t define length limits on URIs, there are practical limits imposed by browsers and web servers. It is RECOMMENDED not to exceed an URI length of 2083 characters for maximum interoperability.

Input parameters defined in OAuth 2.0

Parameter	Presence	Value	Description
response_type	REQUIRED	*String* code	The value SHALL be "code".
client_id	REQUIRED	*String*	The unique "client ID" previously assigned to the signature application by the remote service.
redirect_uri	OPTIONAL	*String*	The URL where the user will be redirected after the authorization process has completed. 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* service | credential	The scope of the access request as described by Section 3.3 of RFC 6749 [11]. • "service": it SHALL be used to obtain an authorization code suitable for service authorization. • "credential": it SHALL be used to obtain an authorization code suitable for credentials authorization. The parameter is OPTIONAL. The defaults scope is "service" in case it is omitted.
state	OPTIONAL	*String*	Up to 255 bytes of arbitrary data from the signature application that will be passed back to the redirect URI. The use is RECOMMENDED for preventing cross-site request forgery.

Input parameters defined in this specification

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 otherwis
credentialID	REQUIRED Conditional	*String*	The identifier associated to the credential to authorize. It SHALL be used only if the scope of the OAuth 2.0 authorization request is "credential". Be aware that this parameter value may contain forbidden characters and SHALL be url-encoded.
numSignatures	REQUIRED Conditional	*Number*	The number of signatures to authorize. Multi-signature transactions can be obtained by using a combination of array of hash values and by calling multiple times the **signatures/signHash** method, as defined in section 11.9. It SHALL be used only if the scope of the OAuth 2.0 authorization request is "credential".
hash	REQUIRED Conditional	*String*	One or more base64url-encoded hash values to be signed. It allows the server to bind the *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** method, as defined in section 11.5, for the current *credentialID* is "2", otherwise it is OPTIONAL. Multiple hash values can be passed as comma separated values, e.g. oauth2/authorize?hash=dnN3ZX...ZmRm,ZjIxM3...Z2Zk,...

			The order of multiple values does not have to match the order of hashes passed to **signatures/signHash** method, as defined in section 11.9.
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 section 8.3.1. It MAY be required by a RSSP if their authorization server has a restricted access. The value is a JSON Web Token (JWT) according to RFC 7519 [16].
clientData	OPTIONAL	*String*	Arbitrary data from the signature application. It can be used to handle a transaction identifier or other application-specific data that may be useful for debugging purposes. WARNING: this parameter MAY expose sensitive data to the remote service. Therefore it SHOULD be used carefully.

Output: After a successful user authentication, 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_type": it SHALL be used if the server does not support the required response type. • "invalid_scope": it SHALL be used if the requested scope is invalid, unknown, or malformed. • "server_error": it SHALL be used if the server encountered an unexpected condition that prevented it from fulfilling the request. • "temporarily_unavailable": it SHALL be used if the server is currently unable to handle the request due to temporary overload or maintenance. It SHALL be returned only in case of an error.
error_description	OPTIONAL	*String*	Human-readable text providing additional error information. It MAY be returned only in case of an error.
error_uri	OPTIONAL	*String*	A URI identifying a human-readable web page with information about the error. It MAY be returned only in case of an error.

Sample Request (Service authorization)

```
GET https://www.domain.org/oauth2/authorize?
response_type=code&
client_id=<OAuth2_client_id>&
redirect_uri=<OAuth2_redirect_uri>&
scope=service&
lang=en-US&
state=12345678
```

Sample Response (Service authorization)

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

Sample Request (Credential authorization)

```
GET https://www.domain.org/oauth2/authorize?
response_type=code&
client_id=<OAuth2_client_id>&
redirect_uri=<OAuth2_redirect_uri>&
scope=credential&
credentialID=GX0112348&
numSignatures=1&
hash=MTIzNDU2Nzg5MHF3ZXJ0enVpb3Bhc2RmZ2hqa2zDtnl4&state=12345678
```

Sample Response (Credential authorization)

```
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
```

8.3.3 oauth2/token (OAuth 2.0 Token Endpoint)

Description: Obtain an OAuth 2.0 bearer access token from the authorization server by passing either the client credentials pre-assigned by the authorization server to the signature application, or the authorization code or refresh token returned by the authorization server after a successful user authentication, along with the client ID and client secret in possession of the signature application. This method SHALL be used only in case of an Authorization Code flow as described in Section 1.3.1 of RFC 6749 [11], in case of Client Credential flow as described in Section 1.3.4 of RFC 6749 [11] or in case of Refresh Token flow as described in Section 1.5 of RFC 6749 [11]. Notice that the Client Credential flow and Refresh Token flow can be used only for service authorization.

A confidential client SHALL authenticate with the authorization server by applying one of the following means:

- 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].

NOTE 1: **oauth2/token** does not specify a regular CSC API method, but rather the URI path component of the OAuth 2.0 Token endpoint. The complete URL to invoke this endpoint is obtained by adding **oauth2/token** to the base URI of the authorization server as returned in the oauth2 parameter by the info method, as defined in section 11.1, and it does not necessarily include the base URI of the remote service API.

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

NOTE 2: 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 section 8.3.2.
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 no authorization header and no client assertion is used.
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 no authorization header and no *client_secret* is used.
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.

Input parameters defined in this specification

Parameter	Presence	Value	Description
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in section 8.3.2.

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

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. When the scope is "service" then the authorization server returns a bearer token to be used as the value of the "Authorization: Bearer" in the HTTP header of the subsequent API requests within the same session. When the scope is "credential" then the authorization server returns a Signature Activation Data token to authorize the signature request. This value SHOULD be used as the value for the SAD parameter when invoking the **signatures/signHash** method, as defined in section 11.9.
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* Bearer | SAD	When the scope is "service", this specifies a "Bearer" token type as defined in RFC6750 [12]. When the scope is "credential", this specifies a "SAD" token type.
expires_in	OPTIONAL	*Number*	The lifetime in seconds of the service access token. If omitted, the default expiration time is 3600 sec. (1 hour).

NOTE 3: The lifetime of the refresh token is determined by the RSSP.

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-bA H3XlFQZ3ndFhkXf9P24/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": "SAD",
  "expires_in": 300
}
```

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
}
```

8.3.4 oauth2/revoke (OAuth 2.0 Revocation Endpoint)

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 SHALL 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 by applying one of the following means:

- 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].

NOTE 1: **oauth2/revoke** does not specify a regular CSC API method, but rather the URI path component of the OAuth 2.0 Revocation endpoint. The complete URL to invoke this endpoint is obtained by adding **oauth2/revoke** to the base URI of the authorization server as returned in the oauth2 parameter by the info method, as defined in section 11.1, and it does not necessarily include the base URI of the remote service API.

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

NOTE 2: 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	REEQUIRED Conditional	*String*	The *client_id* as defined in the Input parameter table in section 8.3.2. 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 section 8.3.3.
client_assertion	REQUIRED Conditional	*String*	The *client_assertion* as defined in the Input parameter table in section 8.3.3.
client_assertion_type	REQUIRED Conditional	*String*	The *client_assertion_type* as defined in the Input parameter table in section 8.3.3.

Input parameters defined in this specification

Parameter	Presence	Value	Description
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in section 8.3.2.

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
```

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 forbit multi-signature transactions. The *multisign* output value of the **credentials/info** method, as defined in section 11.5, 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 section 11.9, 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 section 13 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 section 11.1, 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/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 1: Although **oauth2/authorize** , **oauth2/token** and **oauth2/revoke**, as defined in section 8.3, do not specify regular CSC API methods but rather endpoints managed by the OAuth2 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-specific 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 value: 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. 1 for API v1) and Minor is a number identifying the version update, while x and y are subversion numbers. The value corresponding to this specification is “1.0.3.0”.
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.
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. The parameter SHALL be present in any of the following cases:

			• The *authType* parameter contains "oauth2code" or "oauth2client"; • The remote service supports the value "oauth2code" for the authMode parameter returned by **credentials/info**, as specified in section 11.5. This URI SHALL be combined with the OAuth 2.0 endpoints described in Section 8.3.
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.

NOTE 1: **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** and **oauth2/revoke**, as defined in section 8.3, do not specify regular API methods but rather endpoints managed by the OAuth2 authorization server, therefore they MAY be excluded from the list of API method names returned by the *methods* parameter.

Sample Request

```
POST /csc/v1/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/v1/info
```

Sample Response

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

{
  "specs": "1.0.3.0",
  "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"]
}
```

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 1: The RECOMMENDED mechanism for service authorization is OAuth 2.0 (see section 8.3). 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 section 8.3.3).
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] par. 1.5.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in section 8.3.2.

Output value: 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	*Number*	The lifetime in seconds of the service access token. If omitted, the default expiration time is 3600 (1 hour).

NOTE 2: 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 3: 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's up to the remote service to support a single or multiple refresh tokens per user.

NOTE 4: 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

Sample Request

```
POST /csc/v1/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/v1/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 section 11.2) and tokens issued as a result of an OAuth 2.0 flow (e.g. **oauth2/token** as defined in section 8.3.3). 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 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.

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 section 8.3.2.

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	The request is missing a REQUIRED parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
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

Sample Request

```
POST /csc/v1/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/v1/auth/revoke
```

Sample Response

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

11.4 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 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.
maxResults	OPTIONAL	*Number*	The maximum number of items to return from this call. In case this parameter is omitted or invalid (e.g. the value is too big) the remote service SHALL return its own predefined maximum number of items.
pageToken	REQUIRED Conditional	*String*	An opaque token to retrieve a new page of results. The parameter is only REQUIRED to retrieve results other than the first page, based on the value of *maxResult*.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in section 8.3.2.

NOTE 1: 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 value: 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*. No more than *maxResults* items SHALL be returned.
nextPageToken	OPTIONAL	*String*	The page token required to retrieve the next page of results. No value SHALL be returned if the remote service does not suports items pagination or theresponse relates to the last page of results.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
Invalid “pageToken”	400 (bad request)	invalid_request	Invalid parameter pageToken
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

Sample Request

```
POST /csc/v1/credentials/list HTTP/1.1
Host: service.domain.org
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json

{
  "maxResults": 10
}
```

cURL example

```
curl -i -X POST
     -H "Content-Type: application/json"
     -H "Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA"
     -d '{"maxResults": 10}'
https://service.domain.org/csc/v1/credentials/list
```

Sample Response

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

{
  "credentialIDs": [ "GX0112348", "HX0224685" ]
}
```

11.5 credentials/info

Description: Retrieve the credential and return the main identity information and the public key certificate or the certificate chain associated to it. If requested, it can also return an optional 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	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.
certInfo	OPTIONAL	*Boolean*	Request to return various parameters containing information from the end entity certificate. This is useful in case the signature application wants to retrieve some details of the certificate without having to decode it first. The default value is "false", so if the parameter is omitted then the information will not be returned.
authInfo	OPTIONAL	*Boolean*	Request to return various parameters containing information on the authorization mechanisms supported by this credential (PIN and OTP groups). The default value is "false", so if the parameter is omitted then the information will not be returned.
lang	OPTIONAL	*Strings*	The *lang* as defined in the Input parameter table in section 11.1.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in section 8.3.2.

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

Attribute	Presence	Value	Description
description	OPTIONAL	*String*	A free form description of the credential in the *lang* language. The maximum size of the string is 255 characters.
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	*Number*	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*	The status of validity of the end entity certificate. The value is OPTIONAL, so the remote service SHOULD only

		valid | expired | revoked | suspended	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 charcter 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 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 certificate as UTF-8-encoded charcter 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 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 certificate in printable string format, encoded as GeneralizedTime format (RFC 5280 [8]) (e.g. "YYYYMMDDHHMMSSZ"). This value SHALL be returned when *certInfo* is "true".
authMode	REQUIRED	*String* implicit | explicit | oauth2code	Specifies one of the authorization modes. For more information also see section 8.2: • "implicit": the authorization process is managed by the remote service autonomously. Authentication factors are managed by the RSSP by interacting directly with the user, and not by the signature application. • "explicit": the authorization process is managed by the signature application, which collects authentication factors like PIN or One-Time Passwords (OTP). • "oauth2code": the authorization process is managed by the remote service using an OAuth 2.0 mechanism based on authoritzation code as described in Section 1.3.1 of RFC 6749 [11].
SCAL	OPTIONAL	*String* 1 | 2	Specifies if the RSSP will generate for this credential a signature activation data (SAD) 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". NOTE: As decribed in section 8.2, one difference between SCAL1 and SCAL2, as described in CEN TS 119 241-1 [i.5], is that for SCAL2, the signature activation data needs to have a link to the data to-be-signed. The value "2" only gives information on the link between the hash and the SAD, it does not give information if a full SCAL2 as described in CEN TS 119 241-1 [i.5] is implemented.

PIN/presence	REQUIRED Conditional	*String* true | false | optional	Specifies if a text-based PIN is required, forbidden, or optional. This value SHALL be present only when *authMode* is "explicit".
PIN/format	REQUIRED Conditional	*String* A | N	The data format of the PIN string: • "A": the PIN string contains alphanumeric text and/or symbols like "-.%!$@#+". • "N": the PIN string only contains numeric text. This value SHALL be present only when *authMode* is "explicit" and *PIN/presence* is not "false". NOTE: The size of the expected PIN is not specified, since this information could help an attacker in performing guessing attacks.
PIN/label	OPTIONAL Conditional	*String*	A label for the data field used to collect the PIN in the user interface of the signature application, in the language specified in the *lang* parameter. This valuer MAY be present only when *authMode* is "explicit" and *PIN/presence* is not "false". The maximum size of the string is 255 characters.
PIN/description	OPTIONAL Conditional	*String*	A free form description of the PIN in the language specified in the *lang* parameter. This value MAY be present only when *authMode* is "explicit" and *PIN/presence* is not "false". The maximum size of the string is 255 characters.
OTP/presence	REQUIRED Conditional	*String* true | false | optional	Specifies if a text-based One-Time Password (OTP) is required, forbidden, or optional. This value SHALL be present only when *authMode* is "explicit".
OTP/type	REQUIRED Conditional	*String* offline | online	The type of the OTP: • "offline": The OTP is generated offline by a dedicated device and does not require the client to invoke the **credentials/sendOTP** method. • "online": The OTP is generated online by the remote service when the client invokes the **credentials/sendOTP** method. This value SHALL be present only when *authMode* is "explicit" and *OTP/presence* is not "false".
OTP/format	REQUIRED Conditional	*String* A | N	The data format of the OTP string: • "A": the OTP string contains alphanumeric text and/or symbols like "-.%!$@#+". • "N": the OTP string only contains numeric text. This value SHALL be present only when *authMode* is "explicit" and *OTP/presence* is not "false".
OTP/label	OPTIONAL Conditional	*String*	A label for the data field used to collect the OTP in the user interface of the signature application, in the language specified in the *lang* parameter. This value MAY be present only when *authMode* is "explicit" and *OTP/presence* is not "false". The maximum size of the string is 255 characters.
OTP/description	OPTIONAL Conditional	*String*	A free form description of the OTP mechanism in the language specified in the *lang* parameter. This value MAY be present only when *authMode* is "explicit" and *OTP/presence* is not "false". The maximum size of the string is 255 characters.
OTP/ID	REQUIRED Conditional	*String*	The identifier of the OTP device or application. This value SHALL be present only when *authMode* is "explicit" and *OTP/presence* is not "false".
OTP/provider	OPTIONAL Conditional	*String*	The provider of the OTP device or application. This value MAY be present only when *authMode* is "explicit" and *OTP/presence* is not "false".
multisign	REQUIRED	*Number* ≥ 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 section 11.9, 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 section 11.1.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
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

Sample Request

```
POST /csc/v1/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/v1/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"
  },
  "authMode": "explicit",
  "PIN":
  {
    "presence": "true",
    "format": "N",
    "label": "PIN",
    "description": "Please enter the signature PIN"
  },
  "OTP":
  {
    "presence": "true",
    "type": "offline",
    "ID": "MB01-K741200",
    "provider": "totp",
    "format": "N",
    "label": "Mobile OTP",
    "description": "Please enter the 6 digit code you received by SMS"
  },
  "multisign": 5,
  "lang": "en-US"
}
```

11.6 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 section 11.9.

PIN and/or OTP values collected from the user SHALL be included in the request according to the requirements specified by the **credentials/info** method, as defined in section 11.5.
This method SHALL be used in case of "explicit" authorization. This method SHALL also be used in case of "implicit" authorization, to trigger the 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 section 11.7, 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 the 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 section 11.5.
numSignatures	REQUIRED	*Number*	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 section 11.9, multiple times.
hash	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 section 11.5, 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.
PIN	REQUIRED Conditional	*String*	The PIN provided by the user. It SHALL be used only when *authMode* from **credentials/info** is "explicit" and *PIN/presence* is not "false".
OTP	REQUIRED Conditional	*String*	The OTP provided by the user. It SHALL be used only when *authMode* from **credentials/info** method, as defined in section 11.5, is "explicit" and *OTP/presence* is not "false".

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 when *authMode* from **credentials/info** method, as defined in section 11.5, is "implicit" to provide some hints about the occurring transaction.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in section 8.3.2.

Output value: This method returns the following values 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 section 11.9.
expiresIn	OPTIONAL	*Number*	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	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
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 equrest)	Invalid_request	Numbers of signatures is too high
Invalid OTP	400 (bad request)	invalid_otp	The OTP is invalid
Invalid PIN	400 (bad request)	invalid_pin	The PIN is invalid
PIN locked	400 (bad request)	invalid_request	PIN locked
OTP locked	400 (bad request)	invalid_request	OTP locked

NOTE 1: 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.

Sample Request

```
POST /csc/v1/credentials/authorize HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
  "credentialID": "GX0112348",
  "numSignatures": 2,
  "hash":
  [
    "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
    "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
  ],
  "PIN": "12345678",
  "OTP": "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,
           "hash": [ "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
                 "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
                 ],
           "PIN": "12345678",
           "OTP": "738496",
           "clientData": "12345678" }'
https://service.domain.org/csc/v1/credentials/authorize
```

Sample Response

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

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

11.7 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 section 11.9, 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 section 11.5.
hash	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 section 11.5, for the current *credentialID* is "2" , otherwise it is OPTIONAL.
SAD	REQUIRED	*String*	The current unexpired Signature Activation Data. This token is returned by the **credentials/authorize**, as defined in section 11.6, or by the previous call to **credentials/extendTransaction**.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in section 8.3.2.

NOTE 1: 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 section 13.8 shows this use case.

Output value: 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	*Number*	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	The request is missing a REQUIRED parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.

Sample Request

```
POST /csc/v1/credentials/extendTransaction HTTP/1.1
Host: service.domain.org
Content-Type: application/json
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

{
  "credentialID": "GX0112348",
  "hash": [ "WlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0=" ],
  "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",
           "hash": [ "WlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0=" ],
           "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
           "clientData": "12345678" }'
https://service.domain.org/csc/v1/credentials/extendTransaction
```

Sample Response

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

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

11.8 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 section 11.5.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in section 8.3.2.

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 authroization 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 1: 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.

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
```

11.9 signatures/signHash

Description: Calculate the remote digital signature of one or multiple hash values provided in input. This method requires credential authorization in the form of Signature Activation Data (SAD). The signature application SHALL first pass to this method the SAD obtained from either a **credential/authorize**, as defined in section 11.6, or a **oauth2/authorize** calls, as defined in section 8.3.2, depending on the type of supported authorization mechanisms associated with the credential.

In case of multi-signature transactions, the SAD SHALL be updated with **credentials/extendTransaction**, as defined in section 11.7, every time this method is invoked until the maximum number of authorized signatures has been generated.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
credentialID	REQUIRED	*String*	The credentialID as defined in the Input parameter table in section 11.5.
SAD	REQUIRED	*String*	The Signature Activation Data returned by the Credential Authorization methods.
hash	REQUIRED	*Array of String*	One or more hash values to be signed. This parameter SHALL contain the Base64-encoded raw message digest(s).
hashAlgo	REQUIRED Conditional	*String*	The OID of the algorithm used to calculate the hash value(s). This parameter SHALL be omitted or ignored if the hash algorithm is implicitly specified by the *signAlgo* algorithm. 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].
signAlgo	REQUIRED	*String*	The OID of the algorithm to use for signing. It SHALL be one of the values allowed by the credential as returned in *keyAlgo* by the **credentials/info** method, as defined in section 11.5.
signAlgoParams	REQUIRED Conditional	*String*	The Base64-encoded DER-encoded ASN.1 signature parameters, if required by the signature algorithm. Some algorithms like RSASSA-PSS, as defined in RFC 8917 [18], may require additional parameters.
clientData	OPTIONAL	*String*	The *clientData* as defined in the Input parameter table in section 8.3.2.

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

Attribute	Presence	Value	Description
signatures	REQUIRED	*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.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
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 “hash” parameter	400 (bad request)	invalid_request	Missing (or invalid type) array parameter hash
Empty hash parameter	400 (bad request)	invalid_request	Empty hash 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 the SAD.
Missing or not String “signAlgo” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter signAlgo
Missing or not String “signAlgoParams” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter signAlgoParams
Missing or not String “hashAlgo” parameter when “signAlgo” is equal to “1.2.840.113549.1.1.1”	400 (bad request)	invalid_request	Missing (or invalid type) string parameter hashAlgo
Invalid “hashAlgo” parameter	400 (bad request)	invalid_request	Invalid parameter hashAlgo
Invalid “signAlgo” parameter	400 (bad request)	invalid_request	Invalid parameter signAlgo
When present, invalid “clientData” format (not string)	400 (bad request)	invalid_request	Invalid parameter clientData
Invalid “hash” 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.

Sample Request

```
POST /csc/v1/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",
  "hash":
  [
    "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
    "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
  ],
  "hashAlgo": "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",
           "hash": [ "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
                 "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
            ],
           "hashAlgo": "2.16.840.1.101.3.4.2.1",
           "signAlgo": "1.2.840.113549.1.1.1",
           "clientData": "12345678"}'
https://service.domain.org/csc/v1/signatures/signHash
```

Sample Response

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

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

11.10 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 section 8.3.2.

NOTE 1: RFC 3161 [2] contains more detailed definitions of time stamp parameters that can be used in the context of this specification.

Output value: 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	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
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

Sample Request

```
POST /csc/v1/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/v1/signatures/timestamp
```

Sample Response

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

{
  "timestamp":
"MGwCAQEGCSsGAQQB7U8CATAxMA0GCWCGSAFlAwQCAQUABCCrCqnrjH0VxXyQQlfnFJRx1jjrviTs7/
GjKghr2AmluQIIVs5D8OUB4p4YDzIwMTQxMTE5MTEzMjM5WjADAgEBAgkAnWn2SSIWlXk="
}
```

12 JSON schema and OpenAPI description

A signature application may want to validate the JSON objects described in this specification, to ensure that required properties are present and that additional constraints are met. Validation of JSON data is typically performed by means of a specific JSON Schema.

A JSON Schema is a grammar language for defining the structure, content, and semantics of JSON data objects. It can specify metadata about the meaning of an object's properties and values that are valid for those properties. The JSON Schema is defined at https://json-schema.org .

The JSON schema of the API specification described in this specification is available from the website of the Cloud Signature Consortium at https://cloudsignatureconsortium.org/specifications .

The JSON Schema file contains the definition of all CSC API parameters and the definition of the input and output objects managed by the CSC API. The following objects are defined:

- *input-info:* input object for info method
- *output-info:* output object for info method
- *input-auth-login:* input object for auth/login method
- *output-auth-login:* output object for auth/login method
- *input-auth-revoke:* input object for auth/revoke method
- *input-credentials-list:* input object for credentials/list method
- *output-credentials-list:* output object for credentials/list method
- *input-credentials-info:* input object for credentials/info method
- *output-credentials-info:* output object for credentials/info method
- *input-credentials-authorize:* input object for credentials/authorize method
- *output-credentials-authorize:* output object for credentials/authorize method
- *input-credentials-extendTransaction:* input object for credentials/extendTransaction method
- *output-credentials-extendTrasaction:* output object for credentials/extendTransaction method
- *input-credentials-sendOTP:* input object for credentials/sendOTP method
- *input-signatures-signhash:* input object for signatures/signhash method
- *output-signatures-signhash:* output object for signatures/signhash method
- *input-signatures-timestamp:* input object for signatures/timestamp method
- *output-signatures-timestamp:* output object for signatures/timestamp method

In addition, an OpenAPI 3.0 description file is provided, as defined by the OpenAPI Initiative (OAI) https://www.openapis.org, containing these JSON Schema definitions together with other information to fully describe the CSC API protocol. The OpenAPI file contains:

1. A general information about the protocol like, for example, the APIs version, the Cloud Signature Consortium contact information and the license;
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.

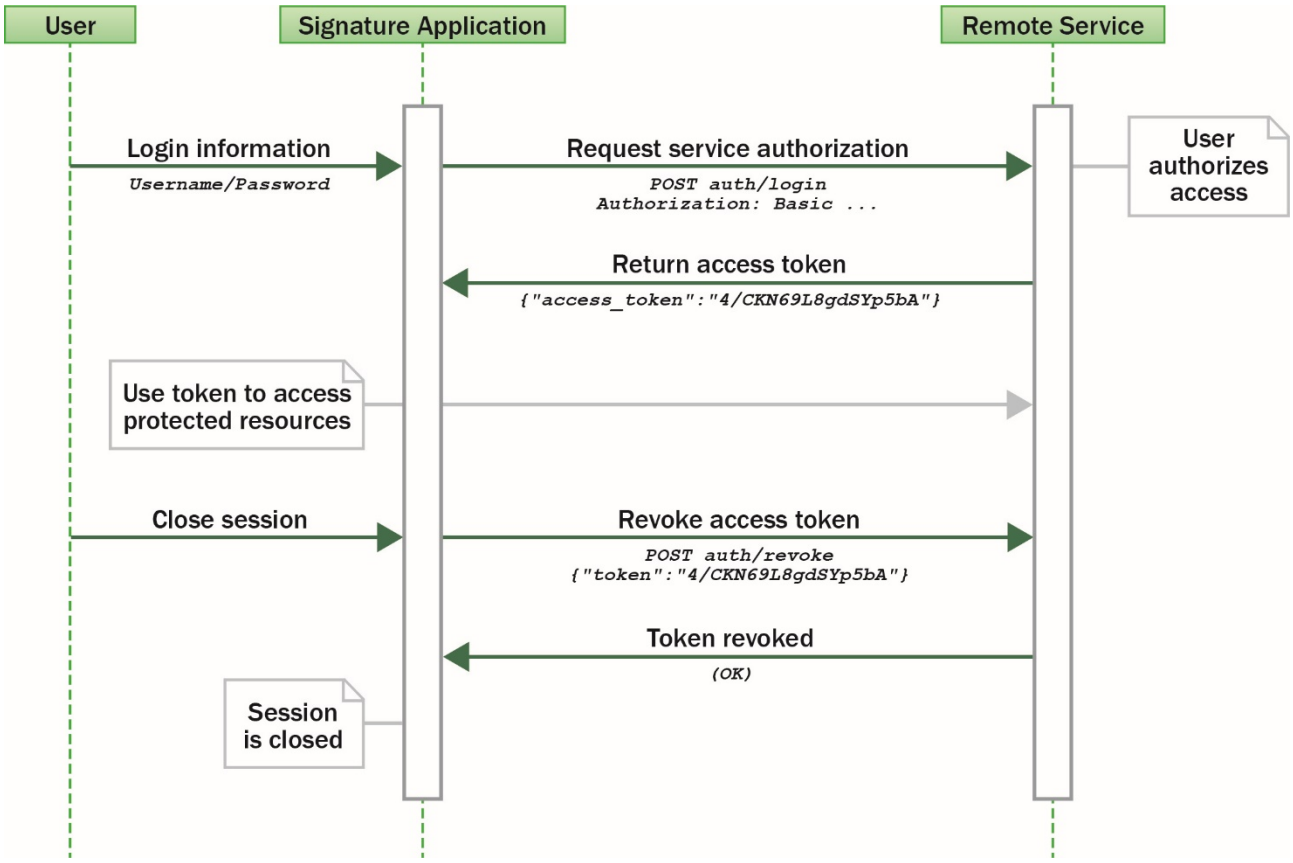
The OpenAPI description file can also be used by developers or testers to automatically generate a CSC compliant server interfaces or client stubs.

13 Interaction among elements and components

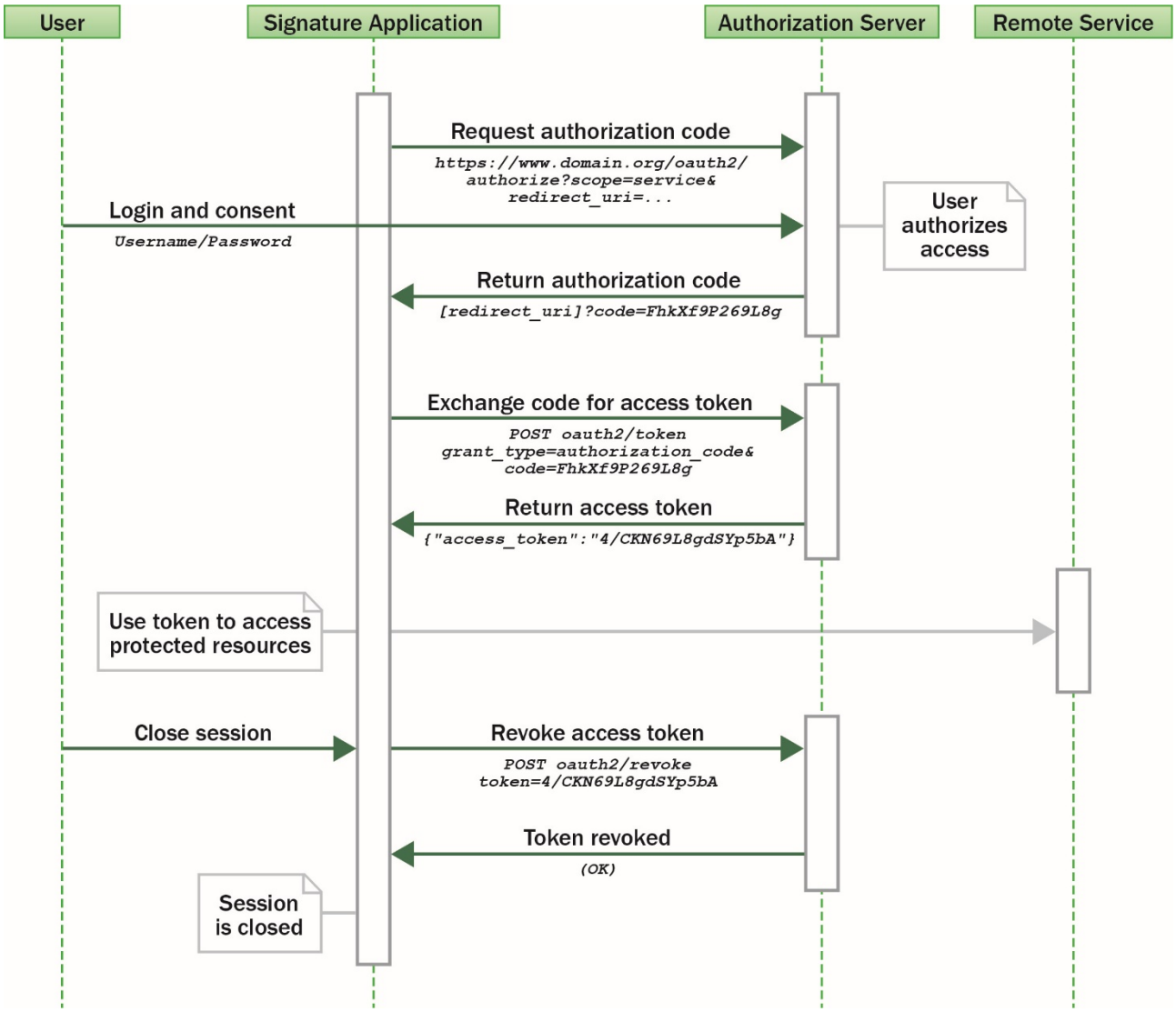
The building blocks of a remote signature solution interact with the API methods described in this specification. The following sections describe the sequence diagrams of some of the most common operations required to obtain a service authorization, credential authorization and to request a remote signature.

NOTE 1: 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.

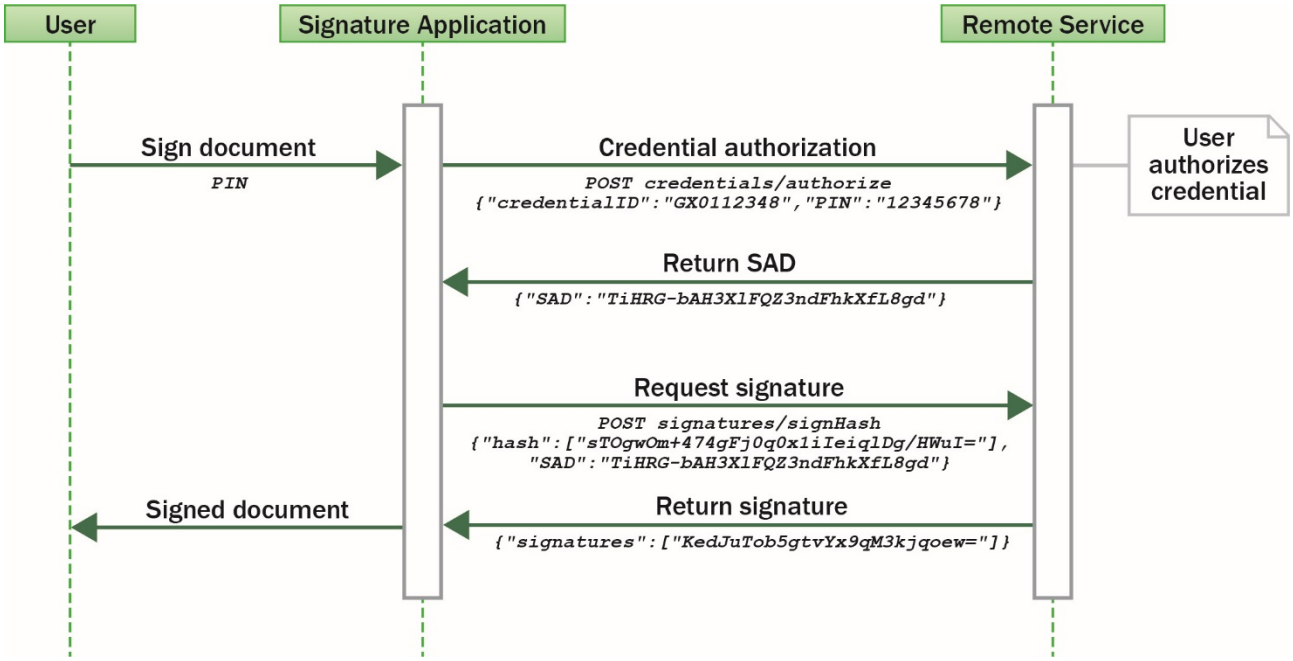
13.1 Remote signing service authorization using Basic Authentication



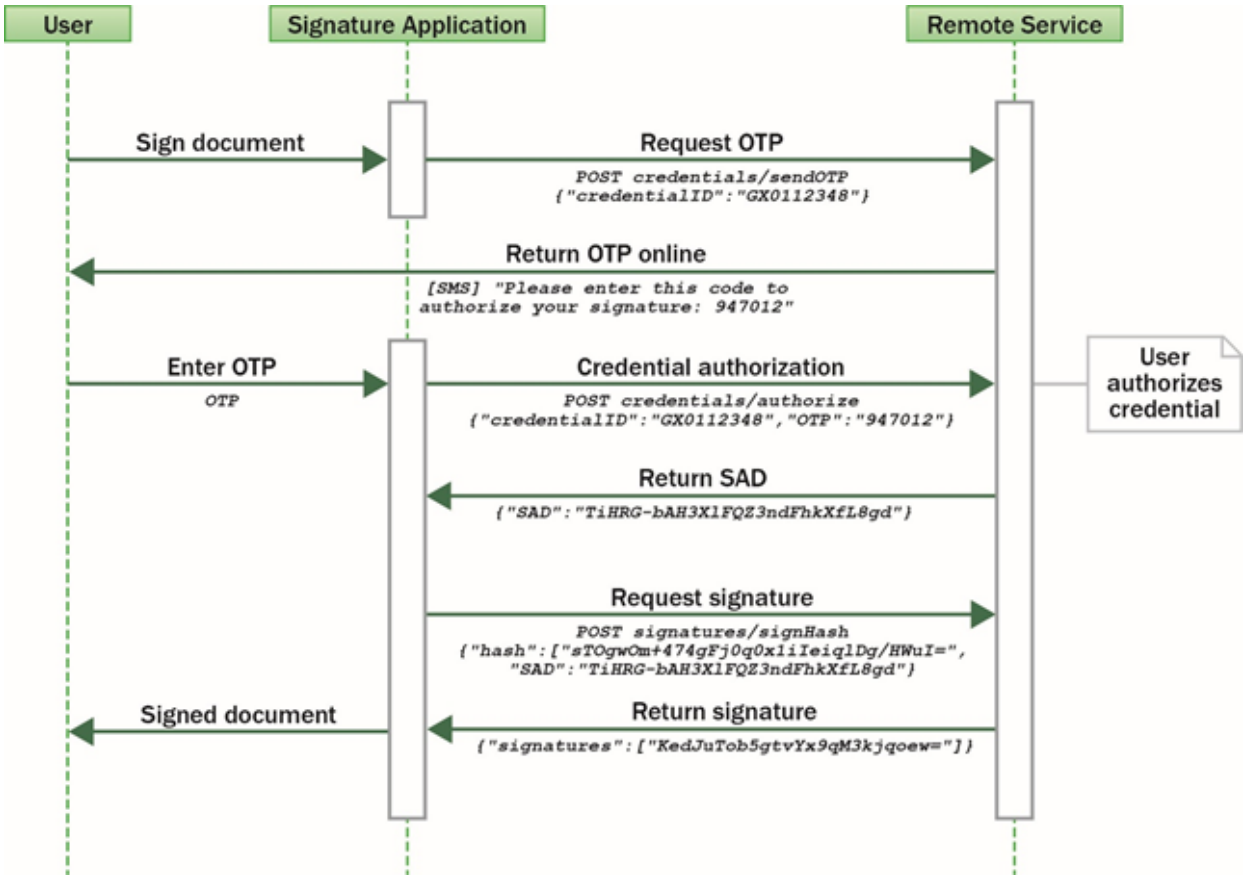
13.2 Remote signing service authorization using OAuth2 with Authorization Code flow



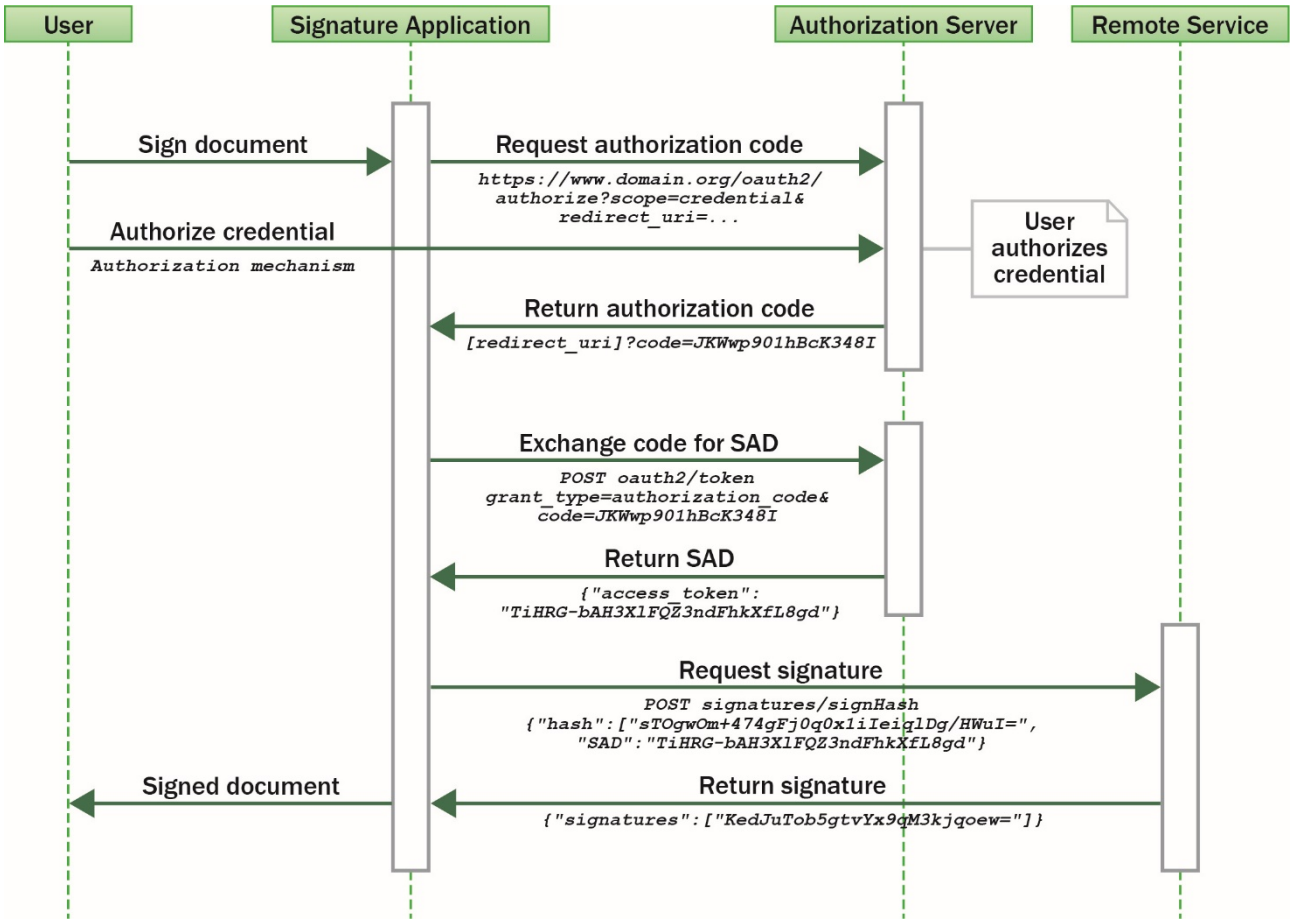
13.3 Create a remote signature with a credential protected by a PIN



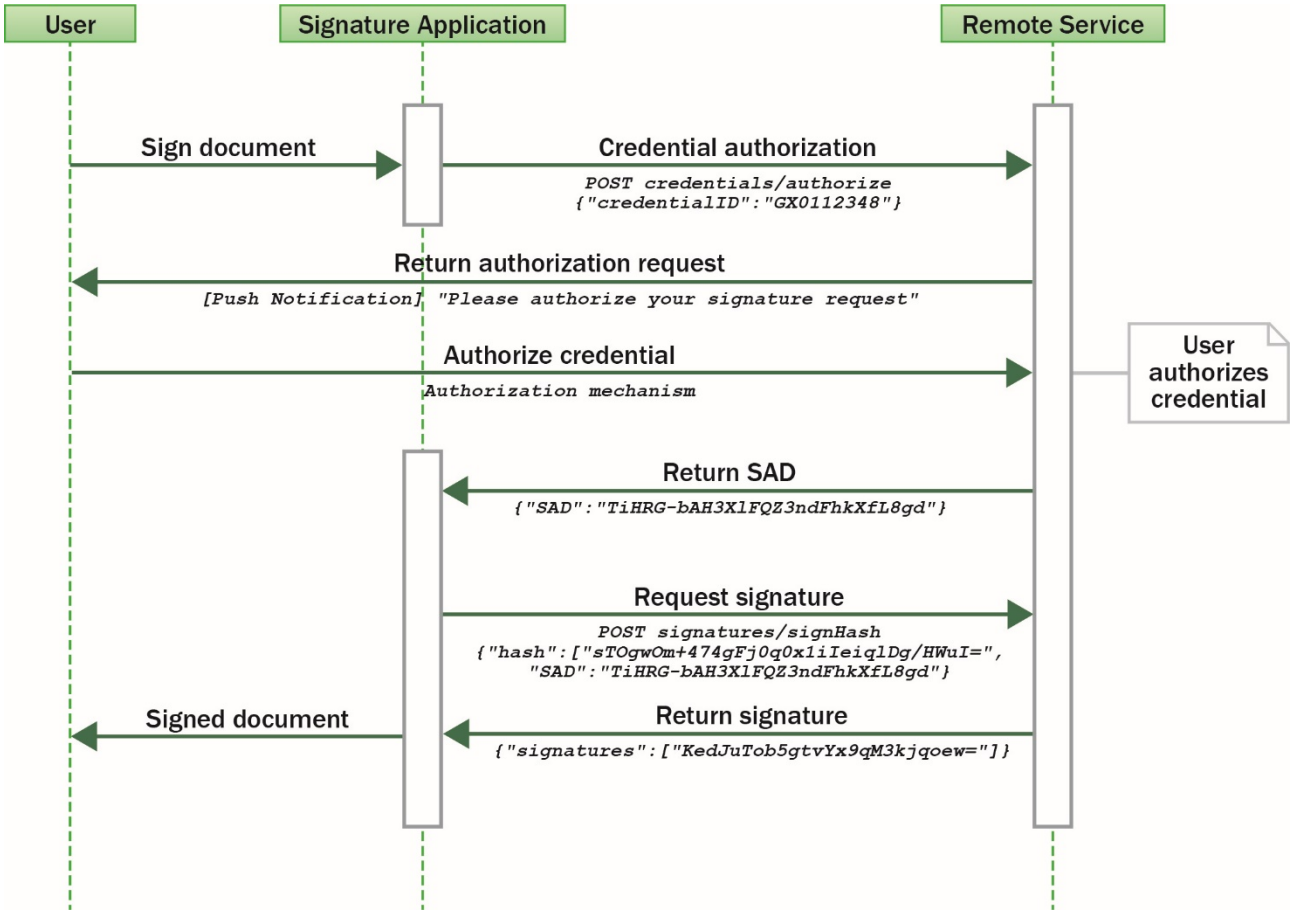
13.4 Create a remote signature with a credential protected by an "online" OTP (based on SMS)



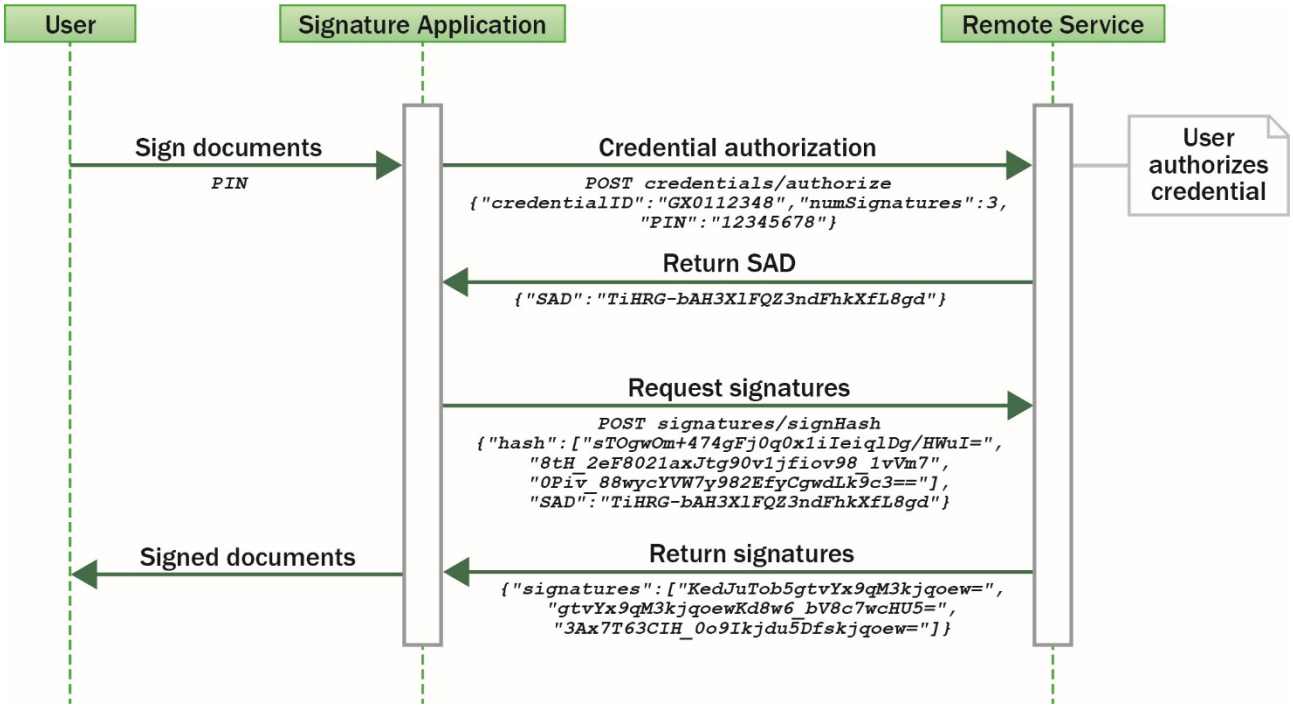
13.5 Create a remote signature with a credential protected by OAuth2 with Authorization Code flow



13.6 Create a remote signature with a credential protected by implicit authorization

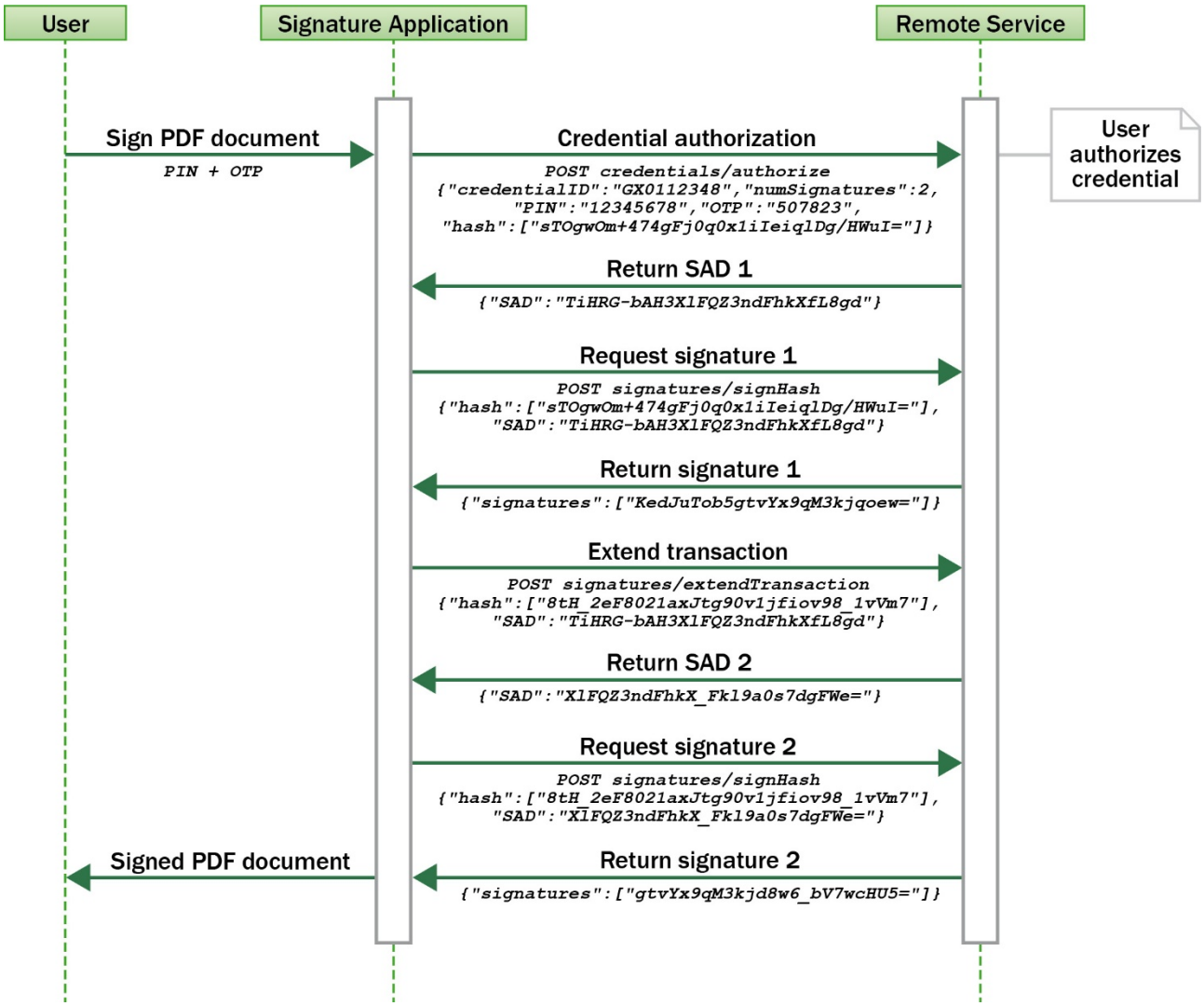


13.7 Create multiple remote signatures from a list of hash values



13.8 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 section 11.7 for more information.



CLOUD
SIGNATURE
CONSORTIUM

Cloud Signature Consortium | Rue du Luxembourg 22-24 | B1000 Brussels | Belgium, EU