



Electronic Signatures and Trust Infrastructures (ESI); Protocols for remote digital signature creation

Reference

RTS/ESI-0019432v131

Keywords

authentication, authorization, electronic signature,
protocol, remote, security, trust services

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Important notice

The present document can be downloaded from the
[ETSI Search & Browse Standards](#) application.

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the prevailing version of an ETSI deliverable is the one made publicly available in PDF format on [ETSI deliver](#) repository.

Users should be aware that the present document may be revised or have its status changed,
this information is available in the [Milestones listing](#).

If you find errors in the present document, please send your comments to
the relevant service listed under [Committee Support Staff](#).

If you find a security vulnerability in the present document, please report it through our
[Coordinated Vulnerability Disclosure \(CVD\)](#) program.

Notice of disclaimer & limitation of liability

The information provided in the present deliverable is directed solely to professionals who have the appropriate degree of experience to understand and interpret its content in accordance with generally accepted engineering or other professional standard and applicable regulations.

No recommendation as to products and services or vendors is made or should be implied.

No representation or warranty is made that this deliverable is technically accurate or sufficient or conforms to any law and/or governmental rule and/or regulation and further, no representation or warranty is made of merchantability or fitness for any particular purpose or against infringement of intellectual property rights.

In no event shall ETSI be held liable for loss of profits or any other incidental or consequential damages.

Any software contained in this deliverable is provided "AS IS" with no warranties, express or implied, including but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights and ETSI shall not be held liable in any event for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption, loss of information, or any other pecuniary loss) arising out of or related to the use of or inability to use the software.

Copyright Notification

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.

The content of the PDF version shall not be modified without the written authorization of ETSI.

The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2026.
All rights reserved.

Contents

List of figures.....	7
Intellectual Property Rights	8
Foreword.....	8
Modal verbs terminology.....	8
Executive summary	8
Introduction	9
1 Scope	10
2 References	10
2.1 Normative references	10
2.2 Informative references.....	11
3 Definition of terms, symbols and abbreviations.....	12
3.1 Terms.....	12
3.2 Symbols.....	14
3.3 Abbreviations	14
4 Signature creation process, service decomposition.....	16
4.1 Signature creation process steps and data elements	16
4.2 Service main components and interfaces.....	17
4.3 Signature Creation Application	17
4.3.1 Signer's document and hashing.....	17
4.3.2 DTBS composition and formatting.....	17
4.3.3 DTBS preparation.....	18
4.3.4 SDO composer.....	18
4.4 Server Signing Application	18
4.4.1 Signature creation	18
4.4.1.1 Introduction.....	18
4.4.1.2 Signature activation.....	18
4.4.1.3 Signature creation by SCDev	19
5 Authentication and authorization	19
5.1 Introduction	19
5.2 Service authorization.....	19
5.2.1 Overview	19
5.2.2 HTTP Basic Authentication and HTTP Digest Authentication	19
5.2.3 OAuth 2.0	21
5.2.4 Other authentication mechanisms.....	23
5.3 Signatures creation authorization	24
5.3.1 Overview	24
5.3.2 Creating a remote signature with a credential protected by signature creation service-managed authorization	24
5.3.3 Create a remote signature with a credential protected by OAuth2.....	26
5.4 Credential creation authorization	27
5.5 Credential deletion authorization	29
6 Architectures and use cases for server signing.....	29
6.1 Introduction	29
6.2 Architectures for creating remote signatures with a credential protected by signature creation service-managed authorization	30
6.3 Architectures for creating remote signatures with a credential protected by OAuth2.....	32
6.4 Architectures for creating remote signatures using the European Digital Identity Wallet	33
6.4.1 Overview	33
6.4.2 EUDIW as authentication and authorization layer for creating remote signatures	34
6.4.3 EUDIW participating in the signature creation process.....	42
6.4.4 EUDIW registration and authentication to authorization servers	44

6.4.4.1	Introduction.....	44
6.4.4.2	EUDIW registration	44
6.4.4.3	EUDIW authentication.....	45
6.4.4.4	Security requirements.....	46
6.4.4.5	Examples.....	46
6.4.4.5.1	Dynamic Client Registration request payload (JSON)	46
6.4.4.5.2	Authorization server response	46
6.4.4.5.3	EUDIW authorization request payload (JSON).....	47
6.4.4.5.4	PAR Response (from AS to EUDIW)	48
6.4.4.5.5	Authorization Request (from EUDIW to authorization endpoint).....	48
6.4.4.5.6	Authorization server redirects back with authorization code.....	48
6.4.5	DA / SCA registration and authentication to authorization servers	48
6.4.5.1	Introduction.....	48
6.4.5.2	client registration.....	48
7	Remote signatures creation service API.....	50
7.1	Introduction	50
7.2	Service information API.....	50
7.2.1	Description.....	50
7.2.2	Request message	50
7.2.2.1	Component for requesting service information.....	50
7.2.3	Response message	50
7.2.3.1	Component for responding to service information requests.....	50
7.3	Credentials list API	51
7.3.1	Description.....	51
7.3.2	Request message.....	51
7.3.2.1	Component for requesting credentials list.....	51
7.3.3	Response message	51
7.3.3.1	Component for responding to credentials list requests.....	51
7.4	Credentials info API.....	51
7.4.1	Description.....	51
7.4.2	Request message.....	51
7.4.2.1	Component for requesting credential information	51
7.4.3	Response message	52
7.4.3.1	Component for responding to credential information requests	52
7.5	Hash(es) signing API.....	52
7.5.1	Description.....	52
7.5.2	Request message.....	52
7.5.2.1	Component for requesting hash(es) signature creation	52
7.5.3	Response message	52
7.5.3.1	Component for responding to hash(es) signature creation requests	52
7.6	Document(s) signing API.....	52
7.6.1	Description.....	52
7.6.2	Request message.....	52
7.6.2.1	Component for requesting documents signature creation	52
7.6.3	Response message	53
7.6.3.1	Component for responding to documents signature creation requests	53
7.7	Signature(s) creation polling API.....	53
7.7.1	Description.....	53
7.7.2	Request message.....	53
7.7.2.1	Component for polling previous signature creation requests	53
7.7.3	Response message	53
7.7.3.1	Component for responding to polling previous signature creation requests	53
7.8	Credentials creation API	53
7.8.1	Description.....	53
7.8.2	Request message.....	53
7.8.2.1	Component for requesting credential creation	53
7.8.3	Response message	54
7.8.3.1	Component for responding to credential creation requests	54
7.8.4	Subject distinguished name of the new credential certificate	54
7.9	Credentials deletion API	54
7.9.1	Description.....	54

7.9.2	Request message	54
7.9.2.1	Component for requesting credential creation	54
7.9.3	Response message	54
7.9.3.1	Component for responding to credential deletion requests	54
8	Remote signatures creation service API based on OASIS DSS-X.....	55
8.1	Introduction and General Provisions	55
8.2	Service information API.....	55
8.2.1	Description.....	55
8.3	Credentials list API	56
8.3.1	Description.....	56
8.4	Credentials info API.....	56
8.4.1	Description.....	56
8.5	Hash(es) signing API.....	56
8.5.1	Description.....	56
8.6	Document(s) signing API.....	56
8.6.1	Description.....	56
8.7	Signature(s) creation polling API.....	57
8.7.1	Description.....	57
8.8	Credentials creation API	57
8.8.1	Description.....	57

Annex A (normative):	OpenID4VP EUDIW-centric signing flow profile	58
A.1	Overview	58
A.2	Roles & Standards	58
A.3	Profile scope & transport.....	59
A.4	Profile identifiers.....	59
A.5	High-level interaction models	59
A.5.1	EUDIW: RP ↔ EUDIW, EUDIW ↔ QTSP	59
A.5.2	QTSP: QTSP ↔ EUDIW	59
A.6	Request construction (RP → EUDIW).....	59
A.6.1	Introduction.....	59
A.6.2	Required parameters (OpenID4VP layer).....	60
A.6.3	dcql_query (selecting an acceptable signing certificate).....	60
A.6.4	transaction_data (the QES transaction).....	61
A.7	Response handling (EUDIW → RP)	62
A.7.1	Inline (VP Token) - x509PresentationResponse	62
A.7.2	Out-of-band (HTTP POST to responseURI)	62
A.7.3	EUDIW processing.....	62
A.8	Transaction data processing & UX rendering (EUDIW).....	63
A.9	Complementary approval flow	63
A.10	Security & privacy requirements.....	63
A.11	Conformance checklist (RP).....	64
A.12	Examples	64
A.12.1	Authorization request (core parameters).....	64
A.12.2	Example 'qesRequest' (transaction_data before base64url encoding).....	64
A.12.3	VP Token (inline) - PAdES result	65
A.12.4	Out-of-band POST to responseURI	65

Annex B (normative):	EUDIW QES creation approval profile	67
B.1	Overview	67
B.2	Roles and architecture	67
B.3	Signature authorization attestation	67
B.4	Profile scope & transport.....	68
B.5	Profile identifiers.....	68
B.6	Data model for approval.....	68
B.6.1	Introduction	68
B.6.2	qesApprovalRequest.....	68
B.6.3	Computing qesApproval (EUDIW).....	69

B.7	OpenID4VP request (QTSP → Wallet)	70
B.8	EUDIW response (EUDIW → QTSP).....	70
B.8.1	VP Token / DC API response.....	70
B.8.2	Using qesApproval at the AS/SCSP	71
B.9	Processing & rendering requirements (EUDIW)	71
B.10	Security & privacy	71
B.11	Conformance checklist (SCSP).....	71
B.12	Working sequence	71
Annex C (normative):	Specific recommendations for qualified signatures and/or seals creation	72
C.1	Introduction	72
C.2	Separation of Responsibilities.....	72
C.3	Authentication and Authorization Layer	72
C.4	Signature Creation Service	72
C.5	Strong Binding Between Consent and DTBS	72
C.6	Use of FAPI 2.0 for EUDIW-Based Clients	72
C.7	SCS API Security Recommendations	72
Annex D (informative):	Change history	73
History		74

List of figures

Figure 1: Process Steps and Data Elements in Signature Creation.....	17
Figure 2: Messages between a driving application and a signature creation service in a common authentication scheme	21
Figure 3: Authorization Code Grant between a driving application and a signature creation service.....	23
Figure 4: A general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by SCSP-managed authorization.....	25
Figure 5: A general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server	27
Figure 6: A general flow involving a driving application requesting signing credential creation to a signature creation service with an authorization process managed by an OAuth2 authorization server	28
Figure 7: Remote hashes signature creation by using a signing credential whose usage is protected by authorization mechanisms managed by the signature creation service	31
Figure 8: Remote hashes signature creation by using a signing credential whose usage is protected by a PIN and an OTP provided through an agreed communication channel managed by the signature creation service (i.e. SMS, app, email).....	32
Figure 9: Multiple remote signatures creation from a list of hash values by using a signing credential whose usage is protected by PIN.....	32
Figure 10: Sequence diagram illustrating an authorization code flow between a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server	33
Figure 11: Sequence diagram illustrating an authorization code flow between a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server by a pushed and rich authorization request	34
Figure 12: Architecture with SCA at QTSP and DA at RP	35
Figure 13: Architecture with SCA and DA at different parties and with EUDIW and user agent in the same user device	36
Figure 14: Architecture with SCA and DA at different parties	37
Figure 15: Architecture with DA at RP and SCA at independent party	38
Figure 16: Architecture with DA and SCA at RP	39
Figure 17: Architecture with authorization server at QTSP and with SCA and DA at different parties	40
Figure 18: Generic simplified flow with EUDIW as authentication and authorization layer.....	41
Figure 19: Remote signature creation with OAuth2 Authorization Code flow and Pushed and Rich Authorization Request with EUDIW as authentication and authorization layer	42
Figure 20: Architecture with SCA and DA at different independent intermediaries and with EUDIW integrating SCA and SIC.....	43
Figure 21: Architecture with SCA and DA at different independent intermediaries and with EUDIW and SCA in the same user device.....	44
Figure 22: Generic simplified signature model flow with EUDIW integrating SCA and SIC.....	45

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the [ETSI IPR online database](#).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM®** and the GSM logo are trademarks registered and owned by the GSM Association.

BLUETOOTH® is a trademark registered and owned by Bluetooth SIG, Inc.

Foreword

This Technical Specification (TS) has been produced by ETSI Technical Committee Electronic Signatures and Trust Infrastructures (ESI).

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the [ETSI Drafting Rules](#) (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Executive summary

The present document specifies protocols and interfaces for components providing specific functionalities as part of a process for remote digital signatures creation and construction of AdES formats.

Introduction

Standards for digital signatures have generally been developed for a long-time considering solutions tailored to the characteristics of devices such as desktop computers and laptops where all signature processing was done in one system locally to the user. These traditional signature solutions assume that the signer uses smart cards or tokens to create any required digital signatures. Given developments in distributed systems, cloud computing, mobile equipment and related technologies, solutions have been emerging in the last few years where the process of digital signature creation and construction of AdES format is done in a distributed way with different steps of the process carried out by different systems/services that may be controlled by different actors.

The present document aims at supporting electronic signatures and electronic seals, including qualified electronic signatures and qualified electronic seals according to the 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.1] (the eIDAS Regulation).

1 Scope

The present document specifies protocols and interfaces applicable when the process of creating AdES digital signatures as defined by ETSI EN 319 102-1 [i.12] and/or digital signature values, as result of Data To Be Signed Representations signatures, is carried out by a distributed solution comprised of two or more systems/services/components.

The present document is limited to remote server signing, i.e. the signing key is held in a remote shared service.

NOTE: Remote signature creation with local signing, i.e. the signing key is held with the signer's personal device but other steps in the signature creation are carried out by means of networked services, is a possible solution but protocols for such architecture are not covered in the present document.

Finally, the present document does not specify any mandatory binding.

As far as it has been possible and suitable, references to constructs of CSC API [1] JSON specifications and OASIS DSS-X [4] XML specifications are specified in the present document. When this has not been possible the present document specifies new components semantically and also syntactically.

The authorized signer's use of its key for signing requires users to provide multiple proofs of their claimed identity before being granted access to the needed set of resources. The present document considers some ways in which the user identity verification process and the user signatures authorization process can be carried out by the service provider.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found in the [ETSI docbox](#).

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents are necessary for the application of the present document.

- [1] [Cloud Signature Consortium Standard \(Version 2.2.0.0\)](#): "Architectures and protocols for remote signature applications".
- [2] [Cloud Signature Consortium Standard \(Version 1.0.0\)](#): "Data model for remote signature applications".
- [3] [Cloud Signature Consortium Standard \(Version 1.0.0\)](#): "CSC data model bindings".
- [4] [OASIS](#): "Digital Signature Service Core Protocols, Elements, and Bindings Version 2.0", comprises JSON and XML Bindings.
- [5] [OASIS](#): "Digital Signature Service Metadata Version 1.0", comprises JSON and XML Bindings.
- [6] [EN 419 241-1](#): "Trustworthy Systems Supporting Server Signing - Part 1: General System Security Requirements", produced by CEN.
- [7] [IETF RFC 8446](#): "The Transport Layer Security (TLS) Protocol Version 1.3".
- [8] [IETF RFC 7515](#): "JSON Web Signature (JWS)".
- [9] W3C®: [XML Signature Syntax and Processing Version 1](#)".

- [10] OpenID Foundation: "[FAPI 2.0 Security Profile](#)".
- [11] [IETF RFC 7591](#): "OAuth 2.0 Dynamic Client Registration Protocol".
- [12] [IETF RFC 7517](#): "JSON Web Key (JWK)".
- [13] [OpenID Connect Relying Party Metadata Choices 1.0](#).
- [14] [IETF RFC 7636](#): "Proof Key for Code Exchange by OAuth Public Clients".
- [15] [IETF RFC 8414](#): "OAuth 2.0 Authorization Server Metadata".
- [16] [IETF RFC 9110](#): " HTTP Semantics".
- [17] [IETF RFC 6749](#): "The OAuth 2.0 Authorization Framework".
- [18] [IETF RFC 9126](#): "OAuth 2.0 Pushed Authorization Requests".
- [19] [IETF RFC 6750](#): "The OAuth 2.0 Authorization Framework: Bearer Token Usage".
- [20] FIDO Alliance: "[Client to Authenticator Protocol \(CTAP\)](#)".
- [21] "[OpenID Connect Core 1.0 incorporating errata set 2](#)".
- [22] [OpenID for Verifiable Presentations 1.0](#).
- [23] W3C®: "[Digital Credentials, W3C Working Draft](#)", 18 December 2025.
- [24] [IETF RFC 9207](#): "OAuth 2.0 Authorization Server Issuer Identification".
- [25] [IETF RFC 9449](#): "OAuth 2.0 Demonstrating Proof of Possession (DPoP)".
- [26] [IETF RFC 8725](#): "JSON Web Token Best Current Practices".
- [27] OASIS: "[Assertions and Protocols for the OASIS Security Assertion Markup Language \(SAML\) 2.0](#)".
- [28] OASIS: "[Profiles for the OASIS Security Assertion Markup Language \(SAML\) 2.0](#)".
- [29] [IETF RFC 7522](#): "Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants".
- [30] W3C®: "[Web Authentication - An API for accessing Public Key Credentials Level 3](#)".
- [31] [IETF RFC 9101](#): "The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding but are not required for conformance to the present document.

- [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, amended by Regulation (EU) 2024/1183.
- [i.2] ETSI TS 119 312: "Electronic Signatures and Trust Infrastructures (ESI); Cryptographic Suites".

- [i.3] ETSI TR 119 001: "Electronic Signatures and Infrastructures (ESI); The framework for standardization of signatures; Definitions and abbreviations".
- [i.4] ETSI TS 103 171: "Electronic Signatures and Infrastructures (ESI); XAdES Baseline Profile".
- [i.5] ETSI TS 103 172: "Electronic Signatures and Infrastructures (ESI); PAdES Baseline Profile".
- [i.6] ETSI TS 119 431-1: "Electronic Signatures and Trust Infrastructures (ESI); Policy and security requirements for trust service providers; Part 1: TSP services operating a remote QSCD / SCDev".
- [i.7] ETSI TS 119 431-2: "Electronic signatures and infrastructures (ESI); Policy and security requirements for trust service providers; Part 2: TSP service components supporting AdES digital signature creation".
- [i.8] IETF RFC 7662: "OAuth 2.0 Token Introspection".
- [i.9] IETF RFC 8259: "The JavaScript Object Notation (JSON) Data Interchange Format".
- [i.10] IETF RFC 7617: "The 'Basic' HTTP Authentication Scheme".
- [i.11] [Architecture and Reference Framework](#).
- [i.12] ETSI EN 319 102-1: "Electronic Signatures and Trust Infrastructures (ESI); Procedures for Creation and Validation of AdES Digital Signatures; Part 1: Creation and Validation".
- [i.13] IETF draft-ietf-oauth-sd-jwt-vc-15: "SD-JWT-based Verifiable Digital Credentials (SD-JWT VC)".
- [i.14] [OpenID for Verifiable Credential Issuance 1.0](#).
- [i.15] IETF RFC 7616: "HTTP Digest Access Authentication".
- [i.16] IETF RFC 9068: "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens".
- [i.17] IETF RFC 7519: "JSON Web Token (JWT)".
- [i.18] IETF RFC 7523: "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants".
- [i.19] IETF RFC 8017: "PKCS #1: RSA Cryptography Specifications Version 2.2".
- [i.20] ETSI TS 103 173: "Electronic Signatures and Infrastructures (ESI); CAdES Baseline Profile".
- [i.21] ISO/IEC 18013-5: "Personal identification — ISO-compliant driving licence — Part 5: Mobile driving licence (mDL) application".

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI TR 119 001 [i.3] and the following apply:

AdES (digital) signature: digital signature that is either a CAdES signature, or a PAdES signature or a XAdES signature

client application: application that requests authorization from an Authorization Server on behalf of a resource owner (signer in the present document), in order to access protected resources at a Resource Server (signer's private key usage in the present document)

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

digital signature value: result of the cryptographic transformation of a data unit that allows a recipient of the data unit to prove the source and integrity of the data unit and protect against forgery (e.g. by the recipient)

driving application: application that uses a SCASC or a SSASC service to create a signature

FAPI: an OpenID Foundation working group, where FAPI was previously known as the Financial-grade API but where consensus was reached within the working group to update the name to just FAPI to reflect that the specification was appropriate for many high-value use-cases requiring a more secure model beyond just financial services

intermediary: entity acting on behalf of a relying party for the purpose of transmitting or facilitating requests to the EUDIW

relying party: entity that relies upon the output of a trust service, in particular upon a certificate, an electronic signature/seal, or the result of its validation; in a remote-signature architecture, it can be the entity that requests remote signature creation and then relies on the result and/or the recipient that verifies and then relies upon the AdES/QES generated by the remote signing service

remote signature creation device: signature creation device used remotely from signer perspective and providing control of signing operation on the signer's behalf

remote signing server: server application using a signature creation device to create a digital signature value on behalf of a signer

server signing application: application using a remote signature creation device to create a digital signature value on behalf of a signer

server signing application service component: TSP service component employing a server signing application

server signing application service provider: TSP operating a server signing application service component

service authorization: process of verifying and enforcing that a client (such as an application, user, or system component) has the necessary permissions and rights to access and invoke specific operations or endpoints exposed by a service API, according to predefined policies or access control rules

signature activation data: set of data used to control with a high level of confidence a given signature operation, performed by a cryptographic module on behalf of the signer, that is under sole control of the signer

signature activation module: configured software that uses the SAD in order to guarantee with a high level of confidence that the signing keys are used under sole control of the signer

NOTE: As defined in EN 419 241-1 [6].

signature creation application: application within the signature creation system that creates the AdES digital signature and relies on the SCDev to create a digital signature value

NOTE: The SCDev can be managed by the SSASC.

signature creation application service component: TSP service component employing a signature creation application

signature creation application service provider: TSP operating a signature creation application service component

signature creation constraint: criteria used when creating a digital signature

signature creation device: configured software or hardware used to implement the signature creation data and to create a digital signature value

signature creation policy: set of signature creation constraints processed or to be processed by the SCASC or the SSASC

signature creation service: TSP service implementing a signature creation application and/or a server signing application

signature creation service provider: service provider offering a signature creation service

NOTE: As defined in EN 419 241-1 [6].

signing credential: set of the signing key and the corresponding signing certificate

user agent: software component acting on behalf of the resource owner (the signer in the context of the present document) that interacts with the authorization server and the driving application

NOTE: Typically, this is the web browser or application through which the user accesses the driving application. The user agent is responsible for presenting the authorization interface to the user and forwarding requests and responses between the driving application and the authorization server.

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

API	Application Program Interface
AS	Authorization Server
AS/SAM	Authorization Server / Signature Activation Module
CA	Certification Authority
CBOR	Concise Binary Object Representation
CEN	European Committee for Standardization
CMS	Cryptographic Message Syntax
CR	Change Request
CRL	Certificate Revocation List
CSC	Cloud Signature Consortium
CSC API	CSC Standard: "Architectures and protocols for remote signature applications"
CSC DM	CSC Standard: "Data model for remote signature applications"
DA	Driving Application
DC	Digital Credential
DCQL	Digital Credentials Query Language
DPoP	Demonstrating Proof-of-Possession
DSS-X	Digital Signature Services eXtended
DSV	Digital Signature Value
DTBS	Data To Be Signed
DTBSF	Data To Be Signed Formatted
DTBSR	Data To Be Signed Representation
ECDSA	Elliptic Curve Digital Signature Algorithm
eIDAS	electronic Identification And trust Services
EN	European Norm
EUDI	European Digital Identity
EUDIW	European Digital Identity Wallet
FIDO2	Fast IDentity Online 2
HMAC	Hashed Message Authentication Code
HTTP	Hyper Text Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ISO	International Organization for Standardization
JAR	JWT-Secured Authorization Request
JSON	Java Script Object Notation
JWS	JSON Web Signature
JWT	JSON Web Token
LT	Long Term
LTA	Long Term Archival
mTLS	mutual Transport Layer Security
NFC	Near Field Communication
OASIS	Organization for the Advancement of Structured Information Standards
OCSP	Online Certificate Status Protocol
OID	Object IDentifier
OIDC	OpenID Connect

OpenID	OpenID foundation
OpenID4VP	OpenID for Verifiable Presentations
OTP	One-Time Password
PAR	Pushed Authorization Requests
PID	Person Identification Data
PIN	Personal Identification Number
PKCE	Proof Key for Code Exchange
PKI	Public Key Infrastructure
QES	Qualified Electronic Signature
QR code	Quick Response code

NOTE: Two-dimensional machine-readable optical barcode that contains information about the item to which it is attached.

QSCD	Qualified electronic Signature Creation Device
QTSP	Qualified Trust Service Provider
RA	Registration Authority
RAR	Rich Authorization Request
RP	Relying Party
RSA	Rivest, Shamir, & Adleman
RSSP	Remote Signing Server Provider
SAD	Signature Activation Data
SAM	Signature Activation Module
SAML	Security Access Markup Language
SCA	Signature Creation Application
SCAL	Sole Control Assurance Level
SCAL1	Sole Control Assurance Level 1

NOTE: As defined in EN 419 241-1 [6].

SCAL2 Sole Control Assurance Level 2

NOTE: As defined in EN 419 241-1 [6].

SCASC	Signature Creation Application Service Component
SCDev	Signature Creation Device
SCS	Signature Creation Service
SCSP	Signature Creation Service Provider
SD	Signer's Document
SD-JWT VC	Selective Disclosure based JSON Web Token Verifiable Credentials
SDO	Signed Data Object
SDOC	Signed Data Object Composer
SDR	Signer's Document Representation
SHA	Secure Hash Algorithm
SIC	Signature Interaction Component

NOTE: As defined in EN 419 241-1 [6].

SMS	Short Message Service
SRI	SubResource Integrity
SSA	Server Signing Application
SSASC	Server Signing Application Service Component
SSH	Secure Shell Protocol
TLS	Transport Layer Security
TR	Technical Report
TSP	Trust Service Provider
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
UTF-8	Unicode Transformation Format - 8 bit
UX	User eXperience
VP	Verifiable Presentation
WRP	Wallet-Relying Party

WUA
XML
XSD

Wallet Unit Attestation
eXtended Markup Language
XML Schema Definition

4 Signature creation process, service decomposition

4.1 Signature creation process steps and data elements

Figure 1 (derived from ETSI EN 319 102-1 [i.12], clause 4.2.1) shows the various steps and the related data elements for a signature creation process. For remote signature creation, different steps of this process are carried out according to a decomposition into several components, which will have access to or make available the corresponding data elements. The process illustrated in the figure below is limited to the buildings blocks and information needed for creating a signature without specifying possible solutions for tasks such as signer authentication, authorization to the signing key usage or signing certificate availability. The signature activation module in the tamper protected area is needed only when the Signature Creation Service (SCS) complies to the Sole Control Assurance Level 2 (SCAL2) signature activation mechanism as defined in EN 419 241-1 [6].

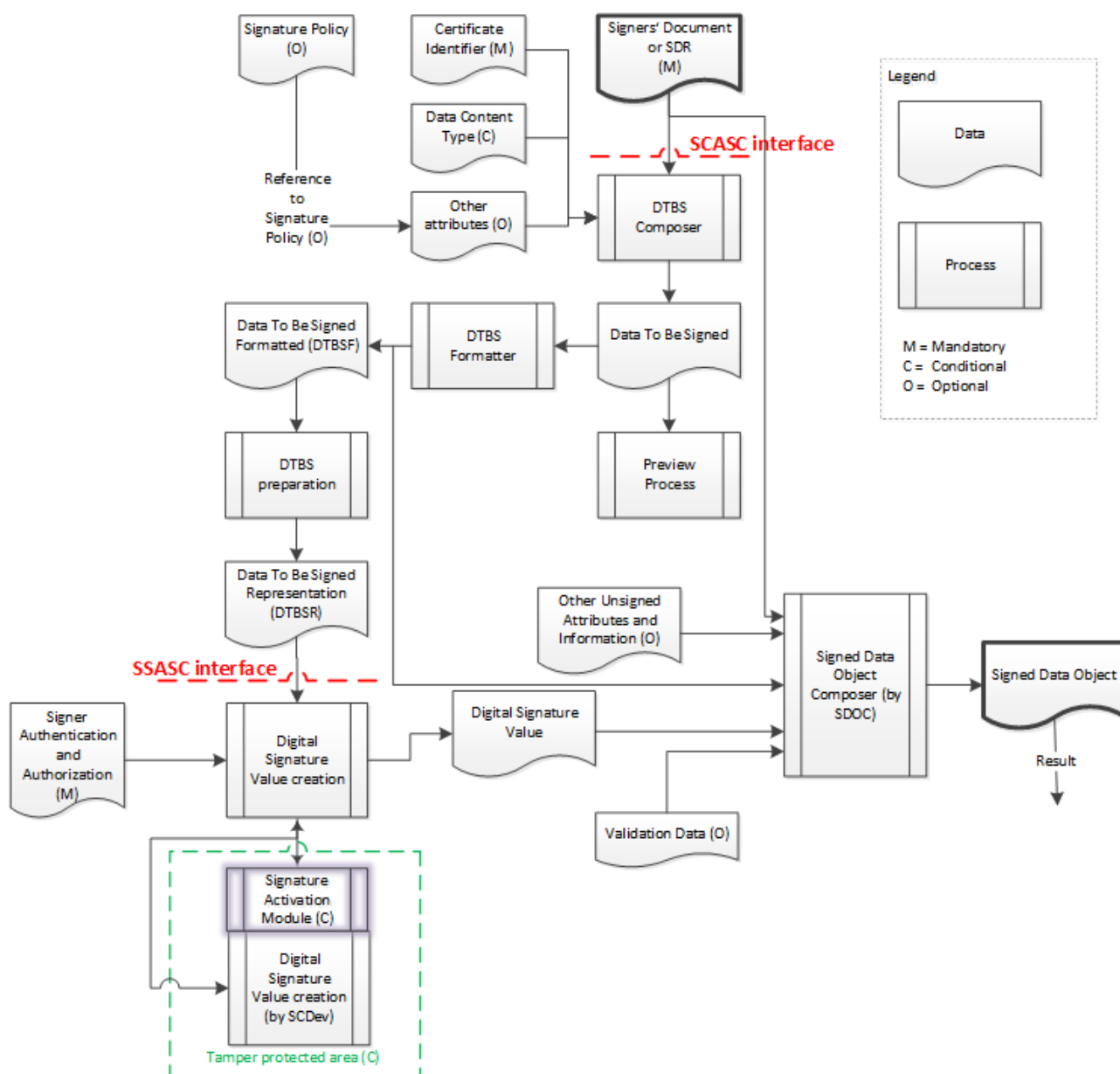


Figure 1: Process Steps and Data Elements in Signature Creation

4.2 Service main components and interfaces

The above process points out scenarios where the AdES and/or Digital Signature Value (DSV) are created using a signing key held within a cryptographic security module named Signature Creation Device (SCDev) operated by a Signature Creation Service Provider (SCSP).

Based on the different types of data managed in requests and responses, two main components can be identified in the above schema providing different interfaces for signing management: the Server Signing Application Service Component (SSASC) and the Signature Creation Application Service Component (SCASC) defined below.

The SSASC is the component supporting digital signature values creation. The SSASC can interact with the SCDev holding the signer's private key. When the SSASC uses the SCDev, the authorized signer controls the signing key with a certain level of confidence.

The SSASC interface has the Data To Be Signed Representation (DTBSR) and other parameters as main input and the digital signature value as main output.

The SCASC is the component supporting AdES digital signature creation and carries out several specific parts of the signature creation process. The SCASC interacts with the SSASC for requesting digital signature values creation.

The SCASC interface has the document(s) to be signed (SD) or its (their) representation (SDR) and other parameters as main input and the signed document(s) or the digital signature(s) as main output.

SCS denotes a TSP service implementing a Signature Creation Application (SCA) and/or a Server Signing Application (SSA).

Some variants of these interfaces are possible depending on the functional split between the SCS and the signer's local system.

The following clauses specify main information objects and processes in SCASC and SSASC.

4.3 Signature Creation Application

4.3.1 Signer's document and hashing

The signature creation process starts with the signer's document (SD), which is to be signed. The SD is represented (SDR) by a hash value in the Data To Be Signed (DTBS). The following observations are made:

- The creation of the SDR (the hashing) can be done where the SD is stored or by the SCASC. In the former case, the SDR is transferred to the SCASC while in the latter case, the SD is transferred to the SCASC.
- The SD is part of the final Signed Data Object (SDO). Part of the Signed Data Object Composer (SDOC) function (building of the final AdES format) is to relate the digital signature value to the SD.

An important design decision for remote signature creation services is where the SD, and thus its content, needs to be available. Making available only the SDR limits threats to confidentiality but may result in limitations in the functionality of the remote signature creation solution (i.e. when enveloping or enveloped signatures need to be created, or when visual representation of the signature needs to be included).

4.3.2 DTBS composition and formatting

In the two processes of DTBS composition and formatting, which in the context of the present document are considered together, the SDR (hash of the document to be signed or hash of a formatted version of the document to be signed) and hashes of signed attributes/properties are assembled into the Data To Be Signed Formatted (DTBSF). In addition to a certificate identifier (hash of signing certificate, possibly also of further certificates in a certificate chain) as indicated in the figure, further signed attributes are required or allowed by the ETSI standard signature formats (C/X/J/PAdES). For example all baseline CAdES and XAdES variants require the presence of the signed attributes "document type" (of SD) and "claimed signing time".

The signed attributes or their hash values, whose presence is needed in the DTBS, are available to the SCASC when the DTBSF is created by the SCASC.

4.3.3 DTBS preparation

This step consists of creating the DTBSR from the DTBSF. The SCASC prepares the entire DTBSF, calculates the hash, and sends the hash value (DTBSR) as input to an SSASC.

4.3.4 SDO composer

As the final step, the SDO (the AdES format) is constructed. This consists of combining the digital signature value with other parameters into the requested format. Depending on the format, the digital signature made available for the SD is named:

- Enveloped: The signature is added to the SD (e.g. PAdES signature).
- Enveloping: The signature wraps the SD (e.g. certain CAdES formats).
- Detached: The signature is a separate object linked to the SD.

The SDO composing is done by a separate service instance or integrated with other functions in the SCASC.

4.4 Server Signing Application

4.4.1 Signature creation

4.4.1.1 Introduction

The purpose of the signature creation process is to take the DTBSR and create a digital signature value under the control of the signer. In the context of the present document, the creation of the digital signature value is managed by an SSASC that uses a signing key, held within a cryptographic security module (SCDev), that the signers can activate by means of a secure authorization and activation process (i.e. by applying the signature activation protocol specified in EN 419 241-1 [6]).

4.4.1.2 Signature activation

The SSASC uses a remote SCDev to generate, maintain and use the signing keys under the control of their authorized signers. The authorized signer remotely controls the signing key with a certain level of confidence possibly by means of the Signature Activation Module (SAM). The SAM is a software component using the Signature Activation Data (SAD) for enforcing sole control in remote signatures creation from the signer confirming its authorization to activate its signing key for the purpose of signing the DTBSR. This process ensures confidence that the signing keys are under the sole control of the signer.

Two different levels of confidence of the control of the signing key, as defined in EN 419 241-1 [6], are considered in the present document:

- Sole Control Assurance Level 1 (SCAL1):
 - The signing keys are used, with a low level of confidence, under the sole control of the signer.
 - The authorized signer's use of its key for signing is enforced by the SSASC which authenticates the signer. The activation of the signing key can remain for a given period and/or for a given number of signatures.

NOTE: It is not expected that such implementations would meet the requirements of sole control as it would be expected for a stand-alone QSCD as defined in the eIDAS [i.1] Regulation.

- Sole Control Assurance Level 2 (SCAL2):
 - The signing keys are used, with a high level of confidence, under the sole control of the signer.
 - The authorized signer's use of its key for signing is enforced by the signature activation module by means of signature activation data provided, by the signer, using a signature activation protocol, in order to enable the use of the corresponding signing key to sign specific documents.

4.4.1.3 Signature creation by SCDev

The signature creation process is performed by the SCDev. In the context of the present document, only architectures where the signature creation process is carried out by a remote SCDev are considered. According to the above sole control assurance levels the signing key can be used to generate the digital signature value creation after a successful signer authentication by the SSASC (SCAL1) or after a successful SAD verification by the SAM.

5 Authentication and authorization

5.1 Introduction

The present document recommends protecting remote signature creation services from unauthorized access. Any driving application shall be authenticated when invoking a signature creation service. In order to create signatures an authorization from the user who owns and controls the signing key is required. The user may authorize the creation of one or more signatures with a signing certificate of a certain type possibly using short-lived signing keys and certificates to be created on-the-fly. The driving application, responsible for requesting signature creation, in the authentication and authorization flows described in the following clauses assumes also the client role implementing the necessary client authentication and authorization procedures.

5.2 Service authorization

5.2.1 Overview

Various types of authorization mechanisms may be supported, and the driving application shall adopt any of those available from the signature creation service.

5.2.2 HTTP Basic Authentication and HTTP Digest Authentication

NOTE: HTTP Basic Authentication [i.10] is an unsafe mechanism, and therefore it should not be used, especially by driving application running as a service. It should only be used when there is a high degree of trust between the user and driving application and when other authorization types are not available. This mechanism may also be deprecated in future releases of the present document. HTTP Basic Authentication should not be employed in processes implementing qualified electronic signature or seal creation, as it does not meet the required security assurance level.

HTTP defines a general challenge-response authentication framework. When a client requests a protected resource, the server replies with the HTTP status code 401 "Unauthorized", defined in clause 15.5.2 of IETF RFC 9110 [16], and a WWW-Authenticate header advertising one or more authentication challenges; the client retries the request including credentials in the Authorization header. Proxy variants use the HTTP status code 407 "Proxy Authentication Required", defined in clause 15.5.8 of IETF RFC 9110 [16].

The driving application may obtain an access token for service authorization directly from the signature creation service using HTTP Basic Authentication, as defined in IETF RFC 9110 [16], using authentication factors (like user and password) assigned to the signer. These authentication factors will be passed directly in the HTTP header as an authorization grant to obtain an access token to use for the subsequent requests within the same session. HTTP Basic Authentication is a simple scheme that sends credentials as base64 (username:password) in the Authorization header. Base64 is an encoding, not encryption, therefore TLS (HTTPS) should be used to protect credentials in transit.

The challenge and response flow works like this:

- The signature creation service responds to the driving application with the HTTP status code 401 "Unauthorized", defined in clause 15.5.2 of IETF RFC 9110 [16], and provides information on how to authorize with a WWW-Authenticate response header containing at least one challenge.
- A client that wants to authenticate itself with the signature creation service can do so by including an Authorization request header with the authentication factors.

- Usually, a driving application will present a password prompt to the signer and will then issue the request including the correct Authorization header.

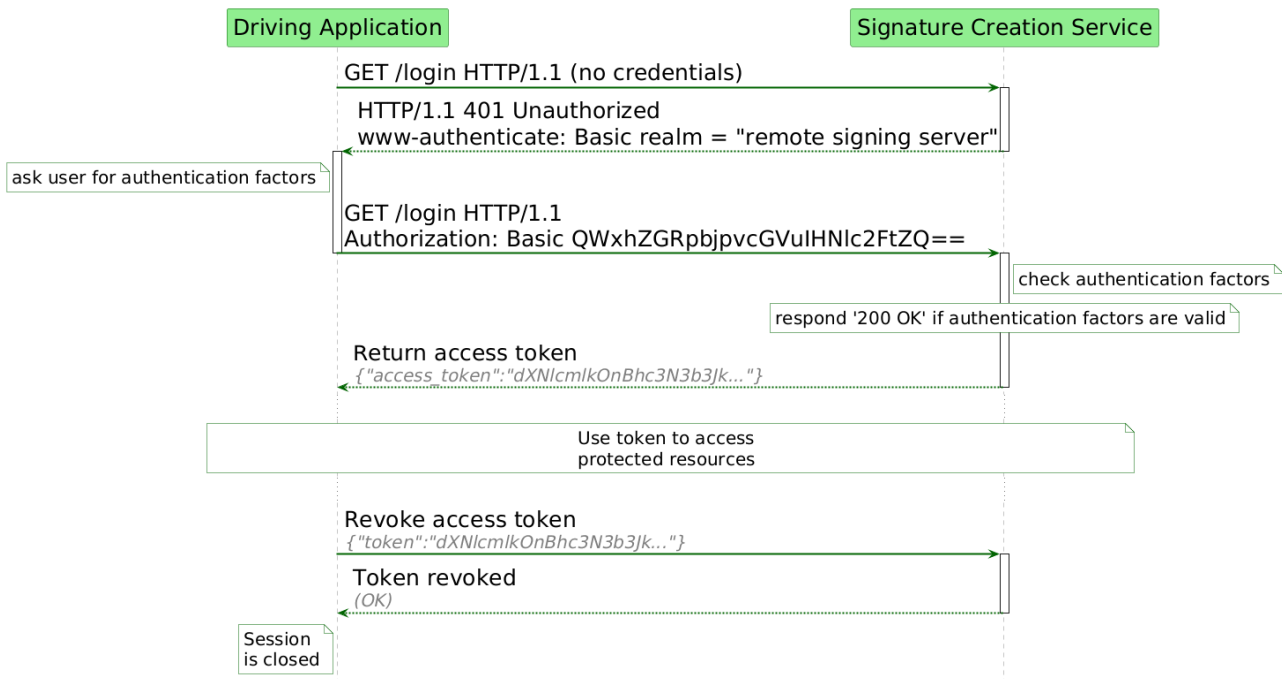


Figure 2: Messages between a driving application and a signature creation service in a common authentication scheme

The "Basic" authentication scheme used in the diagram above sends the authentication factors encoded but not encrypted. This would be completely insecure unless the exchange is over a secure connection (HTTPS/TLS).

The general message flow above is the same for most authentication schemes. The actual information in the headers and the way it is encoded may change. HTTP Digest authentication [i.15] improves on HTTP Basic Authentication by avoiding transmission of plaintext passwords. The server issues a challenge containing parameters such as realm, nonce, algorithm, and qop (quality of protection values supported by the server). The client responds with a hash that proves knowledge of the password, incorporating method and URI, and freshness parameters (nonce, cnonce, nonce count).

The above HTTP Basic Authentication scheme is implemented in CSC API [1] by invoking the API:

- *auth/login* passing the username and password assigned to the signer directly in the HTTP header as an authorization grant.
- *auth/revoke* passing the token, that was obtained from the remote service, to be revoked.

By invoking the *auth/login* API the driving application can obtain an access token for service authorization from the remote service using HTTP Basic Authentication or HTTP Digest authentication, as defined in IETF RFC 9110 [16].

The *auth/login* API identified in CSC API [1], clause 11.2 shall apply.

The message for requesting the access token for service authorization shall contain the components defined in CSC API [1], clause 11.2 section Input.

The message for returning the access token for service authorization shall contain the components defined in CSC API [1], clause 11.2 section Output.

By invoking the *auth/revoke* API the driving application can invalidate access or refresh token obtained from the remote service or an associated authorization server so that any further access by reusing the token itself is prevented.

The *auth/revoke* API identified in CSC API [1], clause 11.3 shall apply.

The message for invalidating an access or refresh token for service authorization shall contain the components defined in CSC API [1], clause 11.3 section Input.

The *auth/revoke* API has no output values, the HTTP status code 204 "No Content", defined in clause 15.3.5 of IETF RFC 9110 [16], is provided in response.

5.2.3 OAuth 2.0

When the authentication is under the control of the signature creation service the signature creation service may adopt an indirect way of authorizing access to the API by using OAuth 2.0 [17] that is primarily an authorization framework, but that can also be leveraged for authentication when interacting with a signature creation service API. OAuth 2.0 ensures that only authorized applications and users can access sensitive signing operations without exposing credentials directly. A common approach is to combine OAuth 2.0 with OpenID Connect (OIDC) [21] to handle user authentication and, optionally, authorization delegation. OAuth 2.0 defines a framework for token-based authorization, while OIDC adds an identity layer for authentication. In this flow, the user authenticates through an Identity Provider, which issues an access token for authorization and, in the case of OIDC, an identity token (JWT [i.4] may be used) that conveys identity information. These tokens can then be presented to the signature creation service. The Identity Provider can be an internal authentication service or an external one. A driving application will include a token in its API calls to the signature creation service, representing the user's identity and permissions. The signature creation service validates this token to ensure the driving application is authorized and allowed to access the requested resource.

Main steps:

1) Client Registration

The driving application (client) registers with the remote signing server's Authorization Server to obtain a *client_id* and possibly a *client_secret*.

2) Authorization Flow

The user (signatory) is redirected to the signature creation service's Authorization Server.

The user authenticates using their identity credentials.

The user grants consent for the driving application to access signature creation service API.

3) Token Issuance

After successful authentication and consent, the Authorization Server issues:

- Access Token, used by the driving application to invoke the signature creation service API.
- ID Token (optional, if OpenID Connect is used), that provides user identity information for authentication purposes.

4) API Access

The driving application includes the Access Token in the Authorization header:

- Authorization: Bearer <access_token>

The signature creation service validates the token before allowing operations like listing credentials or retrieving certificates.

A common token format is JSON Web Tokens (JWT), which are digitally signed tokens that can carry signer identity (subject), issued-at time, expiry, and roles or scopes as claims. They are convenient in services invocation because they are stateless - any service can verify the token with a public key without needing to call back to the issuer for each request. It shall be ensured that the JWT is signed with strong algorithms and that remote signature creation services trust the issuer's public key securely. Using JWT scopes or claims, a fine-grained authorization can be implemented.

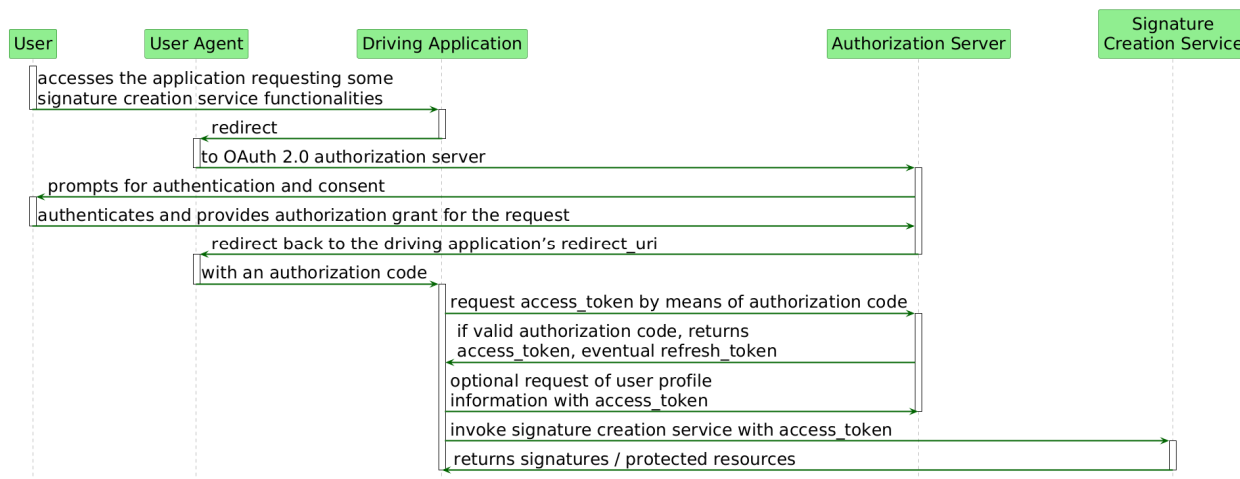


Figure 3: Authorization Code Grant between a driving application and a signature creation service

In the above diagram, if the user grants authorization, the driving application will obtain from the authorization server an access token that can be used for authorizing requests to signature creation service by using an Authorization header with *Bearer* type followed by that access token.

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

NOTE: In order to ensure security and integrity of authorization requests, IETF RFC 9126 [18] recommends invoking first *pushed_authorize* to securely transmit the authorization request, immutable, and protected from tampering or leakage and then the *authorize* endpoint that will only need the *request_uri* lightweight reference, minimizing risk and mitigating injection attacks.

The above scheme is implemented in CSC API [1] by invoking the API:

- *oauth2/authorize* (and *oauth2/pushed_authorize* if needed), that manages authorization requests using the Authorization Code flow as described in Section 4.1 of IETF RFC 6749 [17].
- *oauth2/token* to obtain an OAuth 2.0 bearer access token [19] from the authorization server by passing either the client credentials (id and secret) pre-assigned by the authorization server to the driving application, or the authorization code or refresh token returned by the authorization server after a successful user authentication.
- *oauth2/revoke* passing the access or refresh token, that was obtained from the authorization server, to be revoked.

By invoking the *oauth2/authorize* API the driving application can obtain an authorization code for service authorization from the authorization server of the remote signing service.

The *oauth2/authorize* API identified in CSC API [1], clause 8.2.2 shall apply.

The message for requesting the authorization code for service authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Input considering what specified in section Service scope.

The message for returning the authorization code for service authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Output.

By invoking the *oauth2/pushed_authorize* API the driving application can push the payload of an OAuth 2.0 authorization request to the authorization server via a direct request and obtain a request URI to be used as reference to the data in a subsequent call to the authorization endpoint (*oauth2/authorize*) of the authorization server.

The *oauth2/pushed_authorize* API identified in CSC API [1], clause 8.2.3 shall apply.

The message for requesting the request URI that will be used as reference to the data in a subsequent call to the authorization endpoint shall contain the components defined in CSC API [1], clause 8.2.2 section Input apart from the *request_uri* parameter.

The message for returning the request URI shall contain the components defined in IETF RFC 9126 [18], clause 2.2.

By invoking the *oauth2/token* API the driving application can obtain an access token for service authorization from the authorization server of the signature creation service.

The *oauth2/token* API identified in CSC API [1], clause 8.2.4 shall apply.

The message for requesting the access token for service authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Input.

The message for returning the access token for service authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Output.

By invoking the *oauth2/revoke* API the driving application can invalidate access or refresh token obtained from the authorization server of the signature creation service so that any further access by reusing the token itself is prevented.

The *oauth2/revoke* API identified in CSC API [1], clause 8.2.5 shall apply.

The message for invalidating an access or refresh token for service authorization shall contain the components defined in CSC API [1], clause 8.2.5 section Input.

The *oauth2/revoke* API has no output values, the HTTP status code 204 "No Content", defined in clause 15.3.5 of IETF RFC 9110 [16], is provided in response.

5.2.4 Other authentication mechanisms

The underlying communication between the driving application and the signature creation service may ensure access control in some other different ways than OAuth 2.0 or HTTP Basic or Digest Authentication. The following alternatives can be considered for authenticating clients to a signature creation service:

- Mutual TLS (mTLS) with client certificates. The driving application presents an X.509 certificate during the TLS handshake. The signature creation service validates it (chain, revocation, policy) and authorizes based on the certificate's identity/attributes.
- HMAC-Signed Requests. Each driving application has an API key + secret. It computes HMAC (secret, canonical_request) and sends signature + metadata (timestamp, nonce).
- Signed JWT Assertions. Driving applications self-issue a signed assertion (JWT) proving identity and authorization claims. The signature creation service validates signature (against client public key or its own CA) and standard claims (aud, exp, nbf, jti).
- SAML 2.0 Assertions used within the SAML authentication framework (see [27] and [28]) or alternatively within the OAuth2.0 framework (see [17]) like Bearer (or Holder-of-Key) (see [29]). Driving applications obtain a SAML assertion from an Identity Provider and present it to the signature creation service, with Holder-of-Key, the driving applications prove possession of a key (stronger than bearer).
- SSH Mutual Authentication via Reverse Tunnel. Establishing an SSH tunnel with mutual public-key auth.
- Challenge-Response with Client's Qualified Key (Nonce Signing). The signature creation service issues a nonce (and optional context hash). The driving application signs the nonce with a QSCD/qualified certificate private key (CAAdES/XAdES/JAdES compatible) and returns signature. The signature creation service validates the signature before allowing the requested API invocation.
- FIDO2/Passkey phishing-resistant strong multi-factor authentication (see [30] and [20]).

The above list is not exhaustive, further different solutions can be operational too.

5.3 Signatures creation authorization

5.3.1 Overview

Requesting signatures creation to a signature creation service requires an authorization from the user who owns and controls the signing key to be used for signatures creation. The user might authorize signatures creation with an already existing signing key or with a signing key to be created on-the-fly. Two different levels of confidence of the control of the signing key, as defined in EN 419 241-1 [6], are considered in the present document:

- SCAL1 - Sole Control Assurance Level 1 that ensures that the signature operation is performed on behalf of the signer, but with limited confidence. Typically relies on basic authentication mechanisms (e.g. username/password or PIN). There is no cryptographic binding between the Signature Activation Data (SAD) and the data to be signed.
- SCAL2 - Sole Control Assurance Level 2 that requires stronger authentication (e.g. multi-factor, cryptographic challenge-response) and cryptographic binding between the SAD and the data to be signed. This means the SAD is linked to the specific document or hash being signed, reducing the risk of misuse.

The signing credential use by the driving application may be protected directly by the Signature Creation Service Provider (SCSP) or by means of a framework, like OAuth2, that allows the driving applications to obtain access to the user's signing credential without needing the user's authentication data.

In the following clauses the sequence diagrams of some of the most common operations needed to request and authorize a remote signature creation are described, with particular reference to the possible use of the European Digital Identity Wallet (EUDIW), designed to operate at a Level of Assurance High (LoA), as a signer authentication factor.

5.3.2 Creating a remote signature with a credential protected by signature creation service-managed authorization

NOTE: Explicit authorization requires the driving application to collect authentication factors (e.g. PIN, OTP) within its own environment. This introduces complexity in ensuring secure factor collection and storage. This mechanism should not be suitable to satisfy SCAL2 requirements.

Below is a sequence diagram illustrating a general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by SCSP-managed authorization.

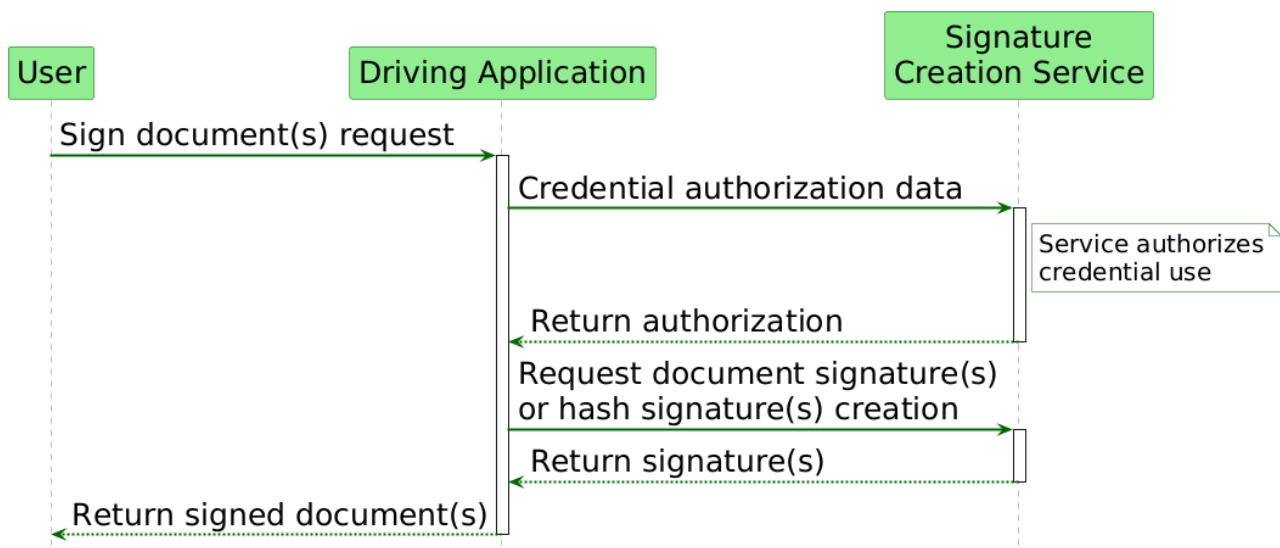


Figure 4: A general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by SCSP-managed authorization

The above scheme is implemented in CSC API by invoking the API

- *credentials/authorize*, passing, in addition to the values identifying the signing credential and the DTBSR, the authentication objects needed to complete signatures creation authorization; CSC API specification [1] in section Authentication objects of clause 11.8 defines a generic way to manage access control to signing credentials associating each signing credential with a set of authentication object types and an access rule describing the precondition to authorize the signing credential access and usage; *credentials/authorizeCheck* and *credentials/extendTransaction* may be invoked for retrieving authorization data and for extending the validity of a multi signature transaction authorization data; when the authorization process is based on a challenge response protocol, *credentials/getChallenge* may be invoked to request a challenge from the signature creation service
- *signatures/signHash* or *signatures/signDoc* to obtain the requested signature objects or signed documents as specified in clause 11.13 and 11.14 of CSC API.

By invoking the *credentials/authorize* API the driving application can obtain the authorization data for making use of a signing credential to create signatures, according to the authorization mechanisms associated to the credential itself.

The *credentials/authorize* API identified in CSC API [1], clause 11.8 shall apply.

The authentication objects and corresponding values collected from the user shall be included in the components defined in CSC API [1], clause 11.8 section Input for creating the request message.

The message for returning the authorization data for signing credential usage shall contain the components defined in CSC API [1], clause 11.8 section Output.

By invoking the *credentials/authorizeCheck* API the driving application can poll the authorization state of a previous *credentials/authorize* request terminated with HTTP status code 202 "Accepted", defined in clause 15.3.3 of IETF RFC 9110 [16], representing that some authorization process was still underway.

The *credentials/authorizeCheck* API identified in CSC API [1], clause 11.9 shall apply.

The message for polling the authorization state of a previous *credentials/authorize* request terminated with HTTP status code 202 "Accepted", defined in clause 15.3.3 of IETF RFC 9110 [16], shall contain the components defined in CSC API [1], clause 11.9 section Input.

The message for returning the authorization data for signing credential usage shall contain the components defined in CSC API [1], clause 11.9 section Output.

By invoking the *credentials/extendTransaction* API the driving application can extend the validity of a multi-signature transaction authorization by obtaining new authorization data so that, i.e. the API method *signatures/signHash* can be invoked multiple times with a single credential authorization event.

The *credentials/extendTransaction* API identified in CSC API [1], clause 11.11 shall apply.

The message for extending the validity of a multi-signature transaction authorization shall contain the components defined in CSC API [1], clause 11.11 section Input.

The message for returning the new authorization data for signing credential usage shall contain the components defined in CSC API [1], clause 11.11 section Output.

By invoking the *credentials/getChallenge* API the driving application can request a challenge needed to perform a challenge response protocol for the access to a credential for remote signing.

The *credentials/getChallenge* API identified in CSC API [1], clause 11.10 shall apply.

The message for requesting a challenge for an authentication object shall contain the components defined in CSC API [1], clause 11.10 section Input.

The message for returning the requested challenge or for indicating that a challenge has been sent by out-of-band means shall contain the components defined in CSC API [1], clause 11.10 section Output.

5.3.3 Create a remote signature with a credential protected by OAuth2

OAuth 2.0 authorization is a widely used framework that allows applications to obtain limited access to a user's resources on a service, without needing the user's authentication data (i.e. the user's password). OAuth 2.0 can provide client applications a secure delegated access to server resources on behalf of a resource owner. In the context of remote signatures creation, the driving application is the client application. This allows signing credentials owners to authorize third-party usage of their signing credentials without sharing their authentication and/or authorization data.

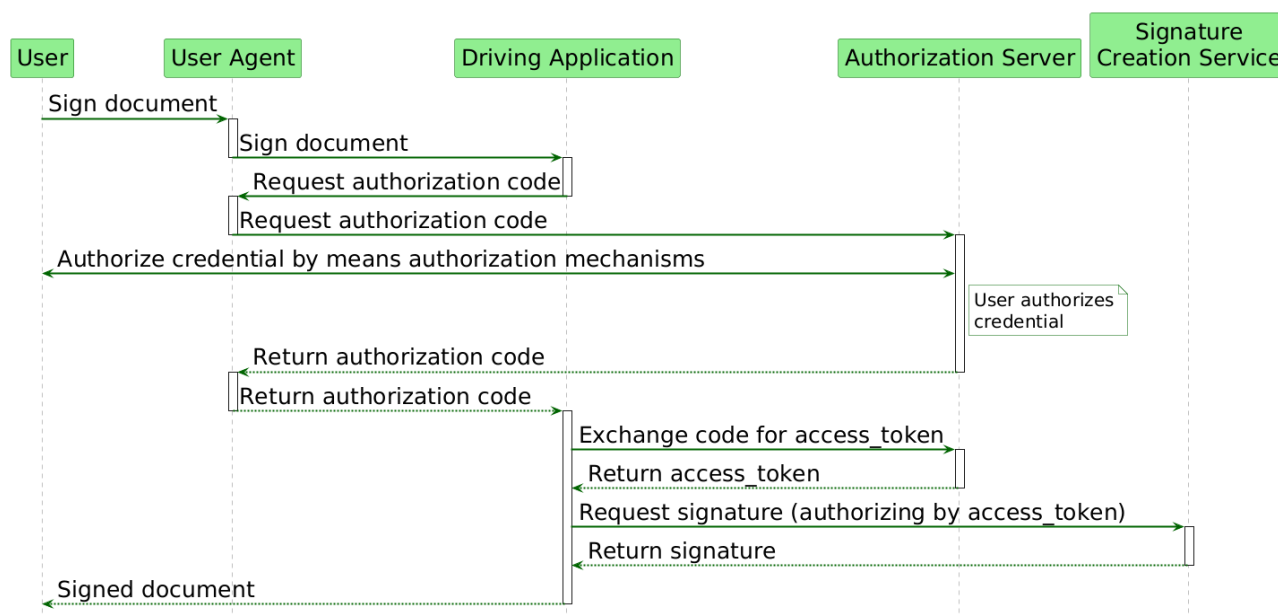


Figure 5: A general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server

The above scheme is implemented in CSC API [1] by invoking the API:

- *oauth2/authorize* (and *oauth2/pushed_authorize* if needed), passing the needed information about the data to be signed and the signing credential to be used according to the parameters defined in clause 8.2.2 and 8.2.3 of CSC API [1], that manages authorization requests using the Authorization Code flow as described in Section 4.1 of IETF RFC 6749 [17].
- *oauth2/token* to obtain an OAuth 2.0 bearer access token from the authorization server by passing the authorization code or refresh token returned by the authorization server after a successful signatures creation authorization.
- *oauth2/revoke* passing the access or refresh token, that was obtained from the authorization server, to be revoked.
- *signatures/signHash* or *signatures/signDoc* to obtain the requested signature objects or signed documents as specified in clause 11.13 and 11.14 of CSC API [1].

By invoking the *oauth2/authorize* API, via the user agent, the driving application can obtain an authorization code for signatures creation authorization from the authorization server of the signature creation service.

The *oauth2/authorize* API identified in CSC API [1], clause 8.2.2 shall apply.

The message for requesting the authorization code for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Input considering what specified in sections Service scope and Credential scope and authorization details.

The message for returning the authorization code for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Output.

By invoking the *oauth2/pushed_authorize* API the driving application can push the payload of an OAuth 2.0 authorization request to the authorization server via a direct request and obtain a request URI to be used as reference to the data in a subsequent call to the authorization endpoint (*oauth2/authorize*) of the remote service.

The *oauth2/pushed_authorize* API identified in CSC API [1], clause 8.2.3 shall apply.

The message for requesting the request URI that will be used as reference to the data in a subsequent call to the authorization endpoint shall contain the components defined in CSC API [1], clause 8.2.2 section Input apart from the *request_uri* parameter.

The message for returning the request URI shall contain the components defined in IETF RFC 9126 [18], clause 2.2.

By invoking the *oauth2/token* API the driving application can obtain an access token for signatures creation authorization from the authorization server of the remote service.

The *oauth2/token* API identified in CSC API [1], clause 8.2.4 shall apply.

The message for requesting the access token for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Input.

The message for returning the access token for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Output.

By invoking the *oauth2/revoke* API the driving application can invalidate access or refresh token obtained from the authorization server of the signature creation service so that any further access by reusing the token itself is prevented.

The *oauth2/revoke* API identified in CSC API [1], clause 8.2.5 shall apply.

The message for invalidating an access or refresh token for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.5 section Input.

The *oauth2/revoke* API has no output values, the HTTP status code 204 "No Content", defined in clause 15.3.5 of IETF RFC 9110 [16], is provided in response.

5.4 Credential creation authorization

Requesting credential creation to the signature creation service requires an authorization from the user to whom the credential to be created will be bound.

CSC API [1] only supports OAuth 2.0 authorization to grant credential creation authorization.

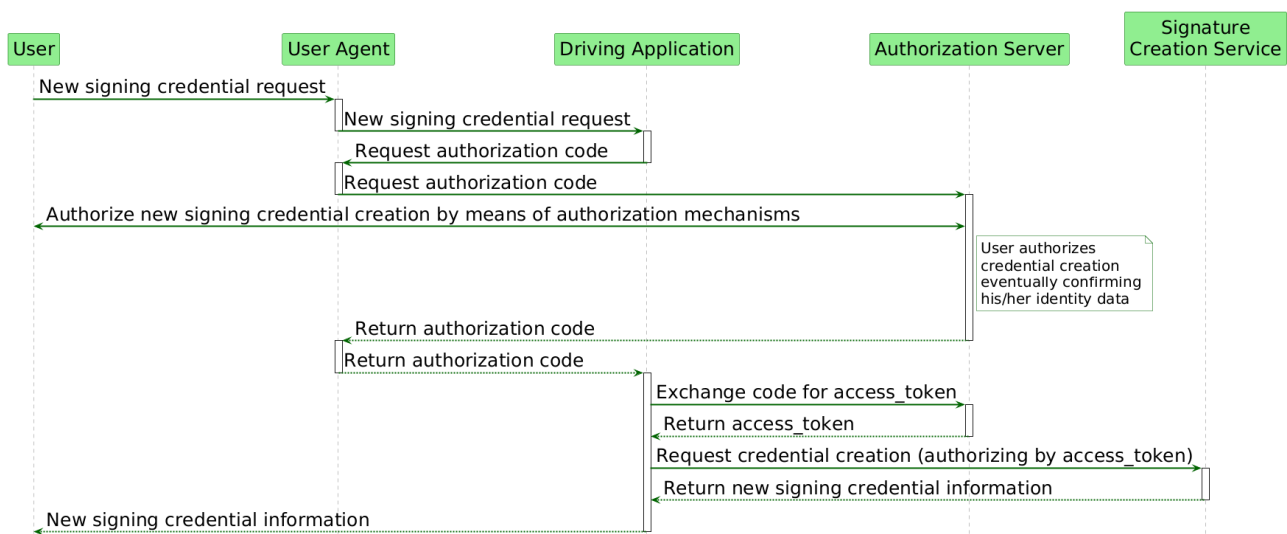


Figure 6: A general flow involving a driving application requesting signing credential creation to a signature creation service with an authorization process managed by an OAuth2 authorization server

The above scheme is implemented in CSC API [1] by invoking the API:

- *oauth2/authorize* (and *oauth2/pushed_authorize* if needed), possibly passing the subject data needed for the certificate signing request according to the parameters defined in clauses 8.2.2 and 8.2.3 of CSC API [1], that manages authorization requests using the Authorization Code flow as described in Section 4.1 of IETF RFC 6749 [17].
- *oauth2/token* to obtain an OAuth 2.0 bearer access token from the authorization server by passing the authorization code or refresh token returned by the authorization server after a successful credential creation authorization.
- *credentials/create* to requests credential creation possibly requesting information about the new signing certificate and/or about the authorization mechanisms required to authorize the usage of the new credential for signing as specified in clause 11.4 of CSC API [1].

By invoking the *oauth2/authorize* API, via the user agent, the driving application can obtain an authorization code for credential creation authorization from the authorization server of the signature creation service.

The *oauth2/authorize* API identified in CSC API [1], in clause 8.2.2 shall apply.

The message for requesting the authorization code for credential creation authorization shall contain the components defined in CSC API [1], clause 8.2.2 considering what specified in section Credential creation scope and authorization details.

The message for returning the authorization code for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Output.

By invoking the *oauth2/pushed_authorize* API the driving application can push the payload of an OAuth 2.0 authorization request to the authorization server via a direct request and obtain a request URI to be used as reference to the data in a subsequent call to the authorization endpoint (*oauth2/authorize*) of the remote service.

The *oauth2/pushed_authorize* API identified in CSC API [1], clause 8.2.3 shall apply.

The message for requesting the request URI that will be used as reference to the data in a subsequent call to the authorization endpoint shall contain the components defined in CSC API [1], clause 8.2.2 sections Input and Credential creation scope and authorization details apart from the *request_uri* parameter.

The message for returning the request URI shall contain the components defined in IETF RFC 9126 [18], clause 2.2.

By invoking the *oauth2/token* API the driving application can obtain an access token for credential creation authorization from the authorization server of the remote service.

The *oauth2/token* API identified in CSC API [1], clause 8.2.4 shall apply.

The message for requesting the access token for credential creation authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Input.

The message for returning the access token for credential creation authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Output.

The signature creation service may collect the subject data needed for the certificate signing request in different ways. The following is a non-exhaustive list of possible ways:

- when Rich Authorization Requests (RAR) are supported, the subject data may be provided in the authorization request;
- the subject data may be collected by the authorization server and made available to the signature creation service remote service:
 - in the authorization details of the access token;
 - as custom claims in the access token [i.16];
 - at a token introspection endpoint [i.8].

5.5 Credential deletion authorization

A credential may be deleted for various reasons:

- End of lifecycle: the credential is no longer needed or has expired.
- Security reasons: the credential may be compromised or the user requests revocation.
- Regulatory compliance: certain frameworks require deletion because of cryptographic algorithms obsolescence

Requesting credential deletion to the signature creation service requires an authorization from the user who owns and controls the credential or from a trusted driving application or trusted third-party. In cases where credential deletion authorization is not requested to the credential owner, the SCSP should inform the credential owner when a credential is deleted.

CSC API [1] only supports OAuth 2.0 authorization to grant credential deletion authorization.

The deletion process is implemented in CSC API [1] by invoking the API:

- *oauth2/authorize* (and *oauth2/pushed_authorize* if needed) according to the parameters defined in clause 8.2.2, specifically in section Credential deletion scope and authorization details, and clause 8.2.3 of CSC API [1], that manages authorization requests using the Authorization Code flow or the Client Credential flow, in case of trusted driving applications, as described respectively in Section 4.1 and 4.4 of IETF RFC 6749 [17].
- *oauth2/token*, possibly, to obtain an OAuth 2.0 bearer access token from the authorization server by passing the authorization code or refresh token returned by the authorization server after a successful credential deletion authorization.
- *credentials/delete* to requests credential deletion as specified in clause 11.5 of CSC API [1].

6 Architectures and use cases for server signing

6.1 Introduction

The following clauses describe a set of architectural models for the creation of remote electronic signatures, each distinguished by the mechanisms used to protect the signer's credential and to manage the authorization required for signature creation. The models differ in the interaction between components and in the authentication and authorization flows, but the governance and final responsibility of authentication and authorization mechanisms remain with the signature creation service provider.

The first clause presents architectures in which the signature creation credential is protected by an authorization mechanism fully managed by the signature creation service. The second clause introduces architectures leveraging OAuth 2.0-based protection of the signature creation credential, focusing on standardized authorization delegation and interoperability. The third clause describes architectures that employ the EUDIW as a central component for user authentication, authorization, and orchestration of signature-creation operations.

Together, these clauses provide a comprehensive overview of the main architectural models currently applicable to remote signature creation, supporting diverse deployment scenarios and regulatory requirements but other architectural models can be implemented achieving comparable levels of security.

Where CSC is referenced in the subsequent sub-clauses, the same considerations generally apply to OASIS DSS-X.

6.2 Architectures for creating remote signatures with a credential protected by signature creation service-managed authorization

NOTE: Explicit authorization requires the driving application to collect authentication factors (e.g. PIN, OTP) within its own environment. This introduces complexity in ensuring secure factor collection and storage. This mechanism should not be suitable to satisfy SCAL2 requirements.

Creating a remote signature using a signing credential protected by a signature creation service can be implemented in some different ways and involving the usage of different authorization data. Moreover, remote signatures creation under the eIDAS [i.1] Regulation involves a well-defined rationale rooted in security, compliance, and usability.

Below some examples of sequence diagrams illustrating specific different flows between a driving application and a signature creation service in the case of remote signatures creation with a signing credential protected by signature creation service-managed authorization mainly derived from CSC API [1].



Figure 7: Remote hashes signature creation by using a signing credential whose usage is protected by authorization mechanisms managed by the signature creation service

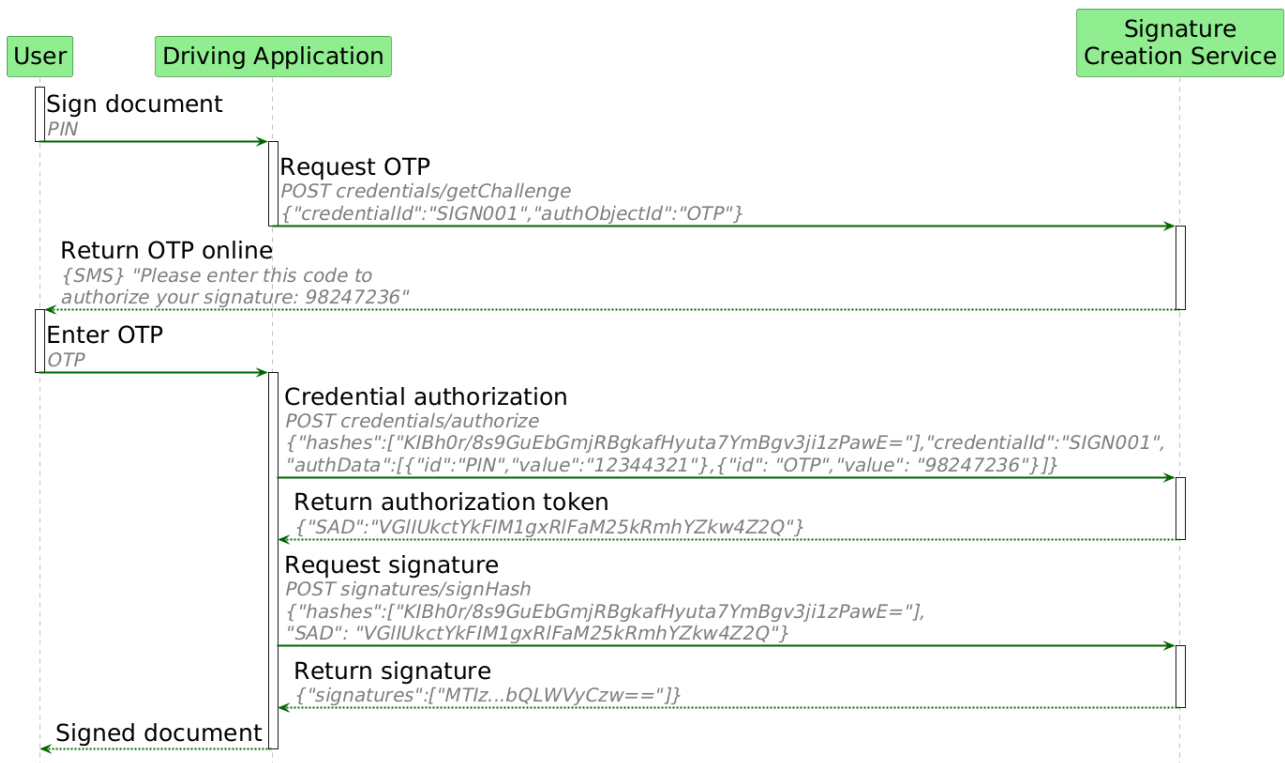


Figure 8: Remote hashes signature creation by using a signing credential whose usage is protected by a PIN and an OTP provided through an agreed communication channel managed by the signature creation service (i.e. SMS, app, email)

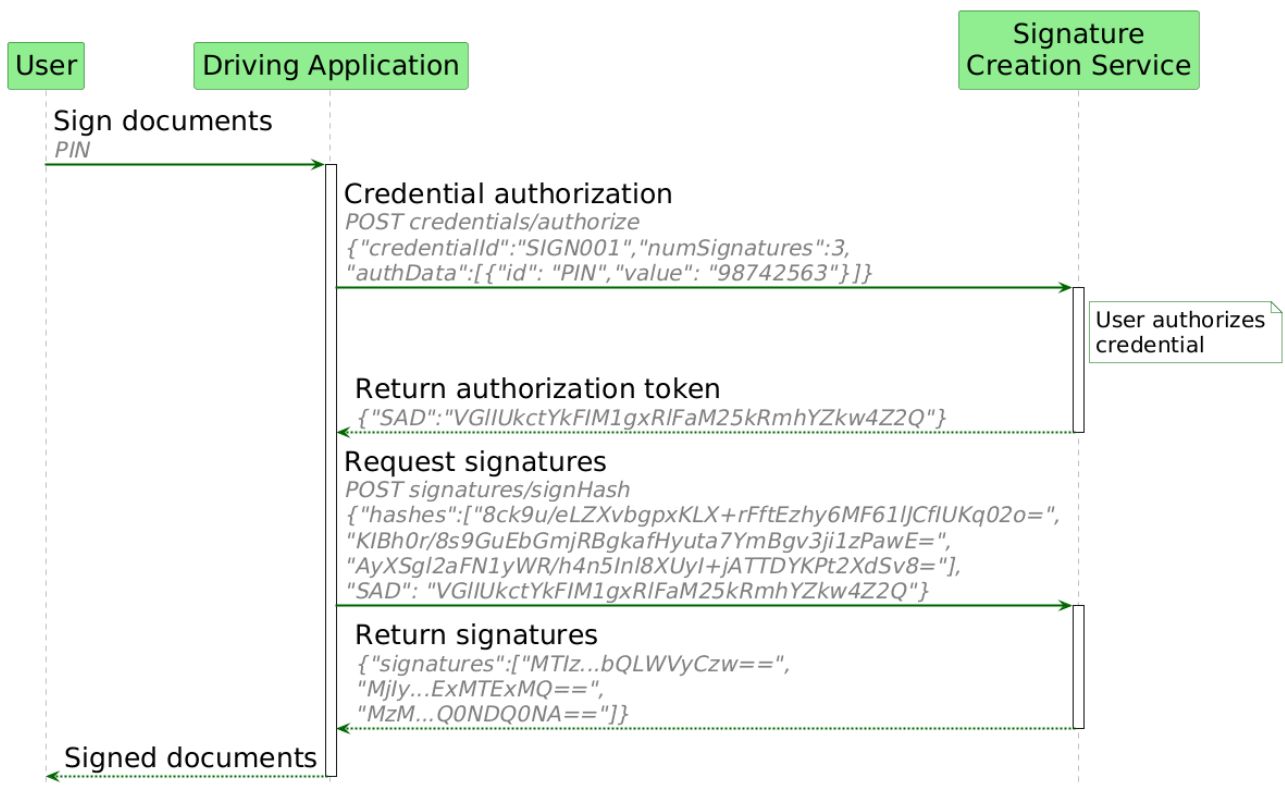


Figure 9: Multiple remote signatures creation from a list of hash values by using a signing credential whose usage is protected by PIN

6.3 Architectures for creating remote signatures with a credential protected by OAuth2

Using the OAuth 2.0 authorization scheme, the driving application will use the remote signature creation service's authorization server for user authentication and signing credential usage authorization. After a successful authentication and authorization, the authorization server of the signature creation service will provide the driving application with an access token that the driving application will use to authorize access to the signature creation service's resources (that's the users signing credentials).

There are some different ways in which requesting a remote signature creation authorization by means of an OAuth2 authorization server. In clauses 8.2.2 and 8.2.3 of CSC API [1] possible choices for selecting the signing credential and providing information about the data to be signed are specified.

The signing credential may be indicated in the credentialID parameter or selected according to the signatureQualifier parameter value that specifies the kind of signature to be created.

The data to be signed may be identified as data to be signed representation or as signer's document or signer's document representation (i.e. hash of SD) together with a label containing a human-readable description of the respective document.

Moreover, the number of signatures to authorize and a list of signed attributes to be included in the signature may be provided too.

Below some examples of sequence diagrams illustrating specific different flows between a driving application and a signature creation service in the case of remote signatures creation with a signing credential protected by an OAuth2 authorization server mainly derived from CSC API [1].

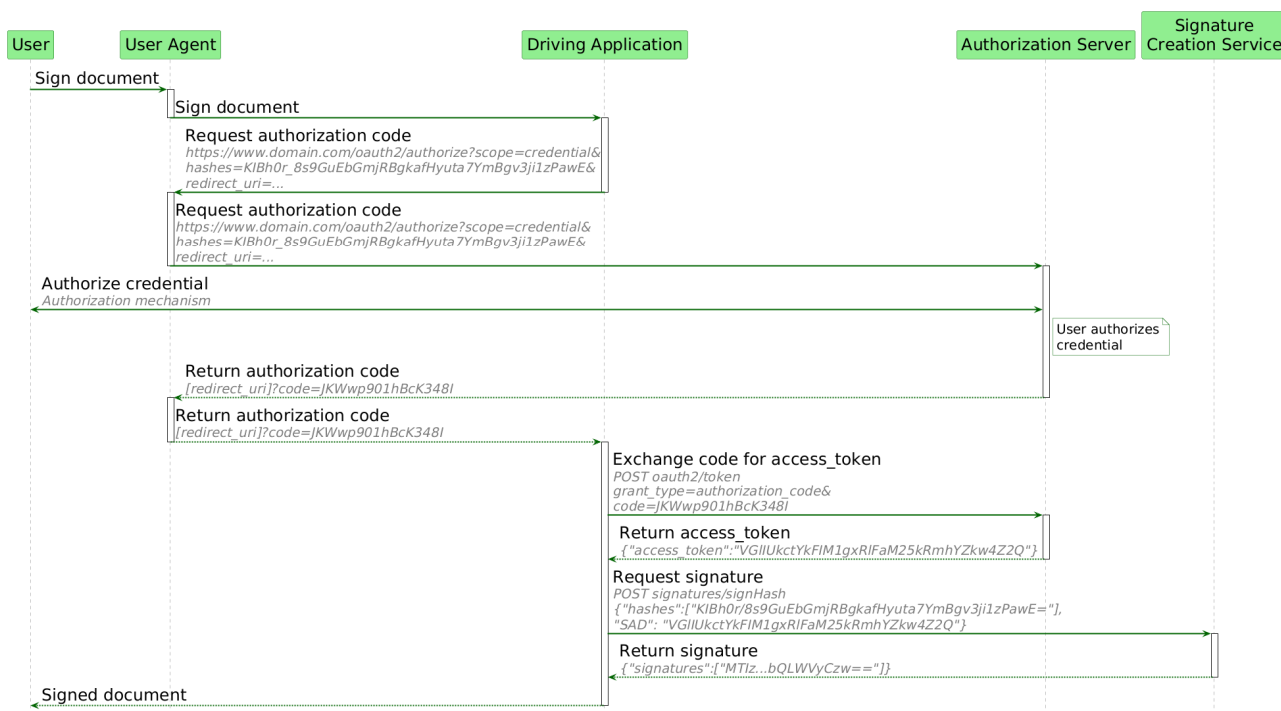


Figure 10: Sequence diagram illustrating an authorization code flow between a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server

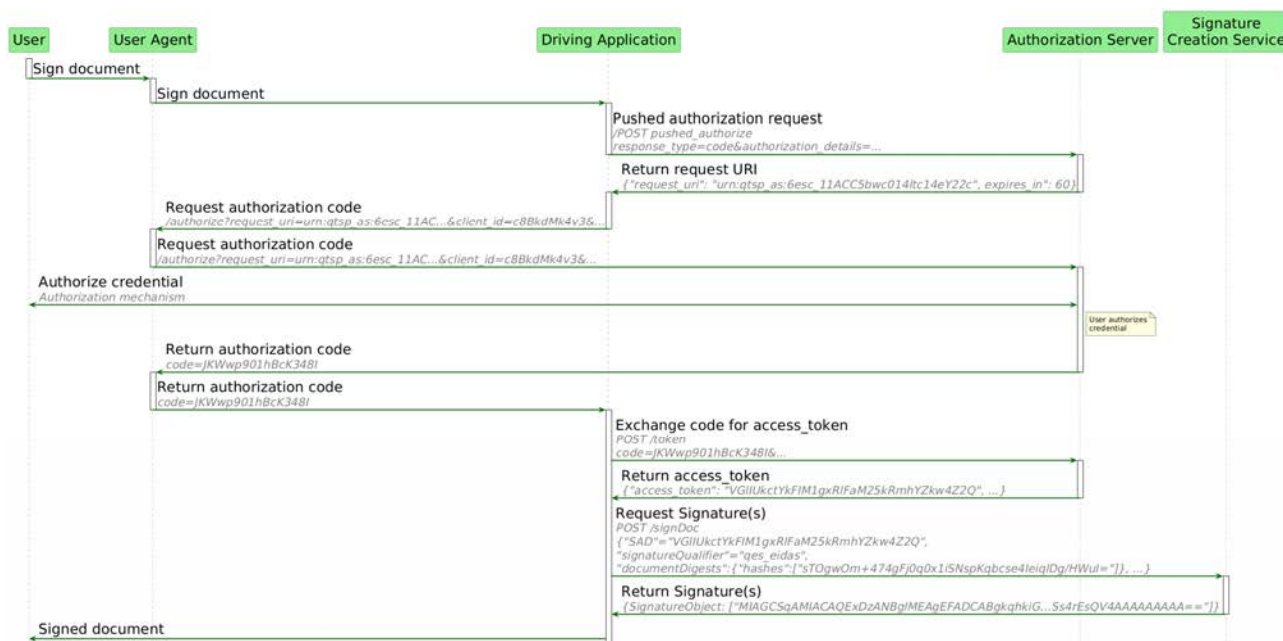


Figure 11: Sequence diagram illustrating an authorization code flow between a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server by a pushed and rich authorization request

6.4 Architectures for creating remote signatures using the European Digital Identity Wallet

6.4.1 Overview

In the European Union, Qualified Electronic Signatures (QES) are governed by the eIDAS Regulation [i.1]. This regulation defines the European Digital Identity Wallet (EUDIW) as an electronic identification means that enables users to securely store, manage, and validate personal identification data and electronic attestations of attributes. The EUDIW serves the purpose of presenting these credentials to relying parties and other EUDIW users, and of performing qualified electronic signatures or qualified electronic seals.

EUDIWs shall allow users to sign using qualified electronic signatures and to seal using qualified electronic seals. In particular, they shall support common protocols and standardized interfaces for creating qualified electronic signatures or seals through qualified electronic signature or seal creation devices.

Several models may be implemented for remote signature creation using the EUDIW, depending on the signature creation flow and the authorization process. In these models two main roles can be played by the EUDIW:

- EUDIW serves as the authentication and authorization layer restricted to user authentication and consent confirmation without performing signature creation.
- EUDIW serves as a mediator for the data flow, acting as a secure party that relays the Data To Be Signed (DTBS) or its hash to the signature creation service, while also assuming a primary role as the requestor initiating the signature creation process when implementing the DA or SCA.

6.4.2 EUDIW as authentication and authorization layer for creating remote signatures

In this architecture, as outlined in Figure 12, the signer uses the EUDIW to authorize a QTSP to sign a document on his/her behalf, the QTSP and the RP play the central roles in managing the signature creation process, while the EUDIW is primarily responsible for authorizing the signature creation. The Signer Interaction Component (SIC) is represented in two distinct parts, SIC Part 1 and SIC Part 2, to reflect the separation of responsibilities between the browser-based user interaction layer and the EUDIW-resident trusted execution layer. This split aligns with the functional decomposition defined in the EUDIW reference architectures, where different parts of the signature orchestration process may be executed in different security domains. The driving application runs in the RP environment and interacts with the authorization server via the user agent. The driving application requests AdES signatures creation to the QTSP which is integrating the SCA, i.e. by invoking CSC signDoc API.

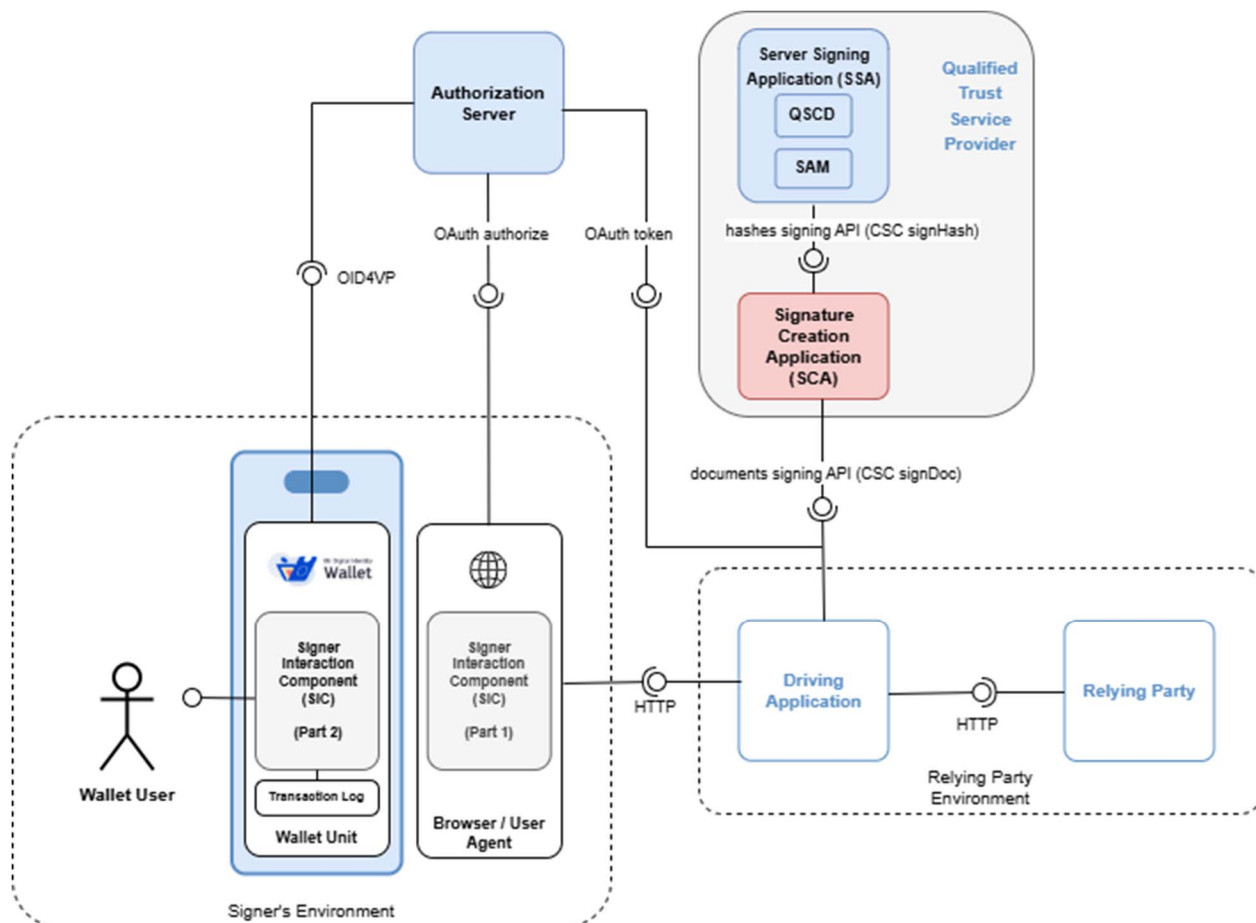


Figure 12: Architecture with SCA at QTSP and DA at RP

Different variations of the above architecture may be implemented according to the following alternatives:

- The user agent and the EUDIW may be in the same user device.
- The SCA may be implemented by an independent party or intermediary or by the RP that will operate both the driving application and the SCA.
- The driving application may be implemented by an independent party.
- The authorization server may be implemented by the QTSP operating the SSA.

Figure 13 shows an architecture in which the user agent and the EUDIW are in the same user device and the SCA and the driving application are implemented by different independent parties.

Figure 14 shows a slightly different architecture in which the user agent and the EUDIW are in different user devices and the SCA and the driving application are implemented by different independent parties.

Figure 15 shows an architecture in which the user agent and the EUDIW are in different user devices, the driving application is implemented at RP and the SCA is implemented by an independent party.

Figure 16 shows an architecture in which the user agent and the EUDIW are in different user devices, the driving application and the SCA are implemented at RP.

Figure 17 shows an architecture in which the user agent and the EUDIW are in different user devices, the SCA and the driving application are implemented by different independent parties, and the authorization server is implemented by the QTSP operating the SSA.

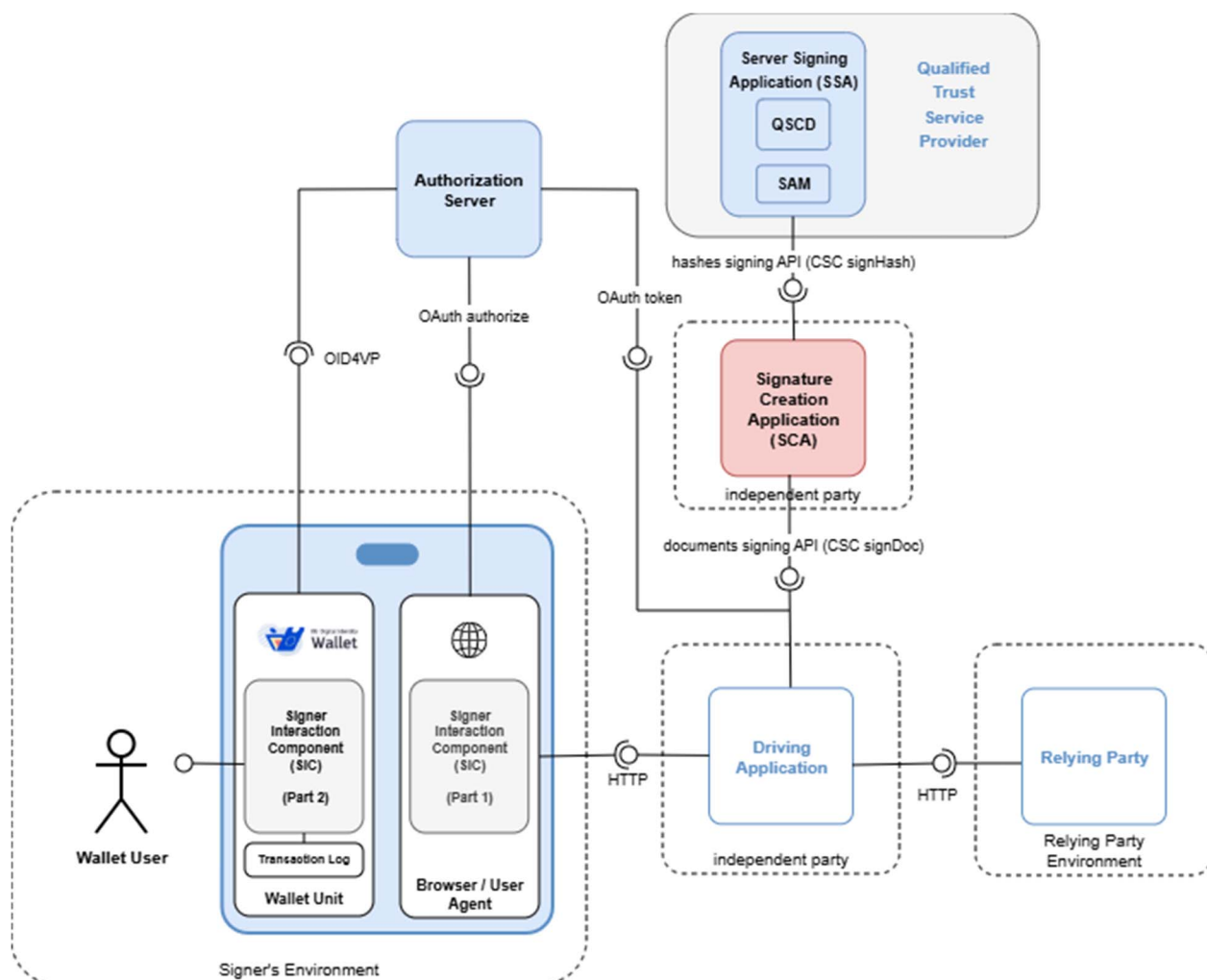


Figure 13: Architecture with SCA and DA at different parties and with EUDIW and user agent in the same user device

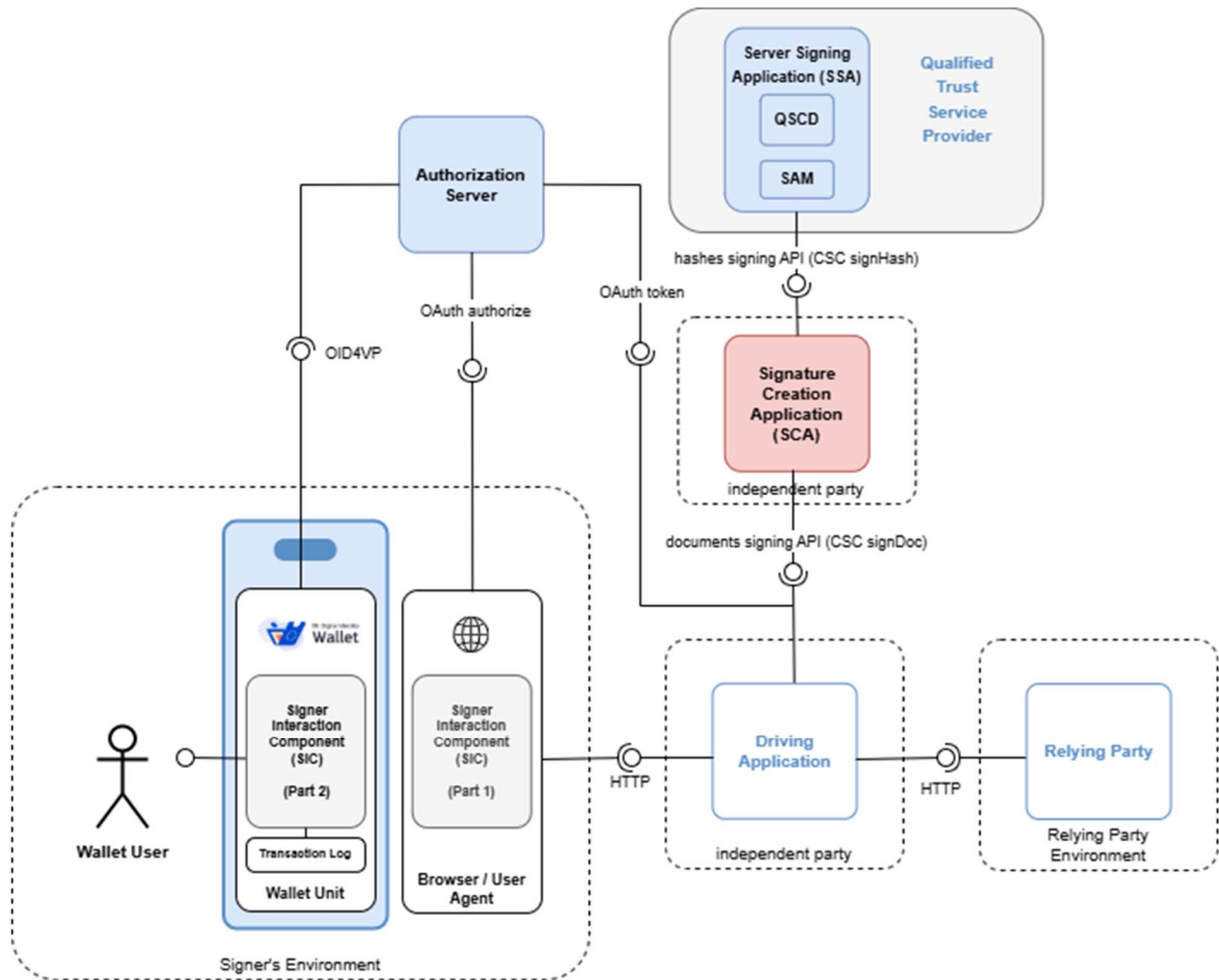


Figure 14: Architecture with SCA and DA at different parties

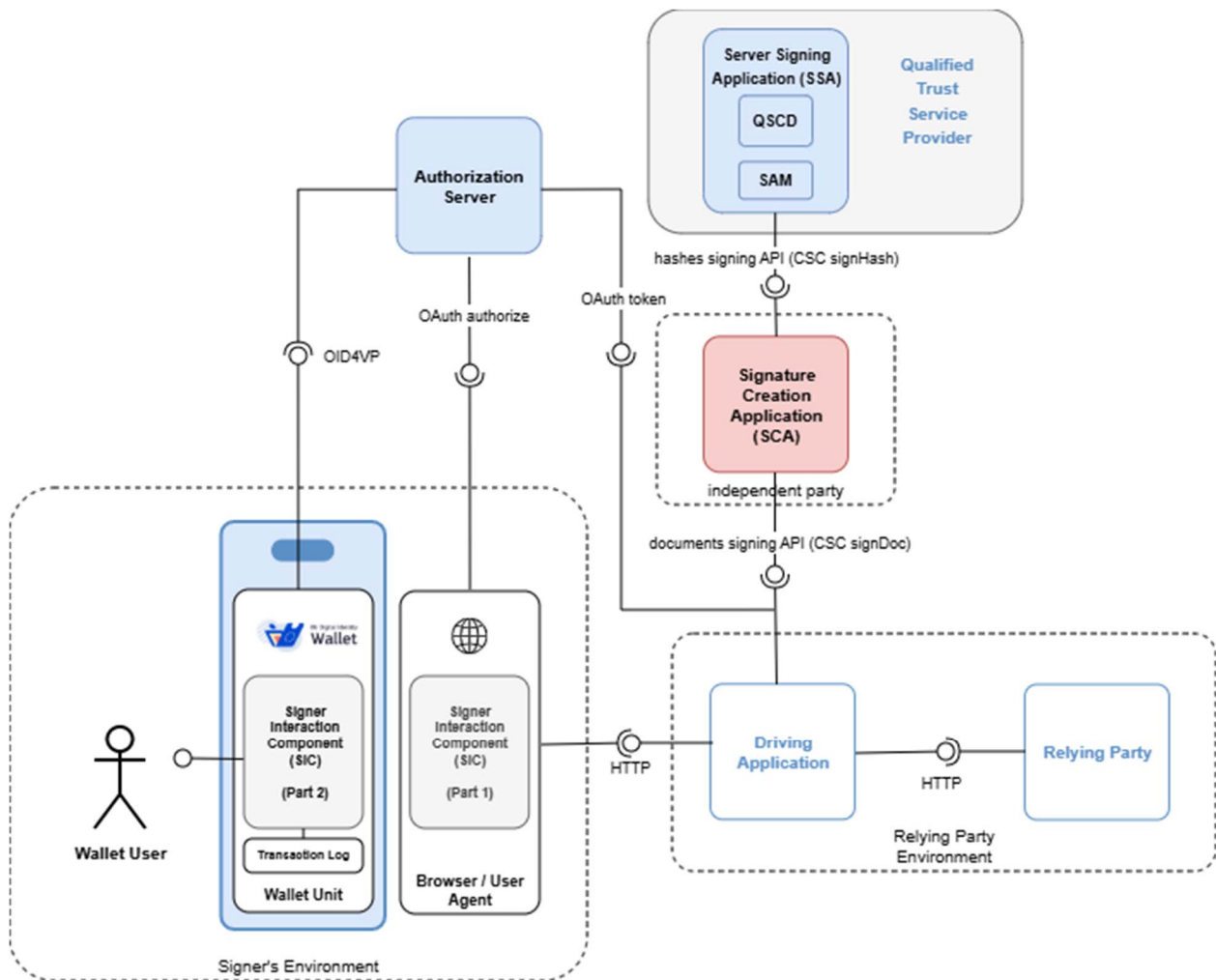


Figure 15: Architecture with DA at RP and SCA at independent party

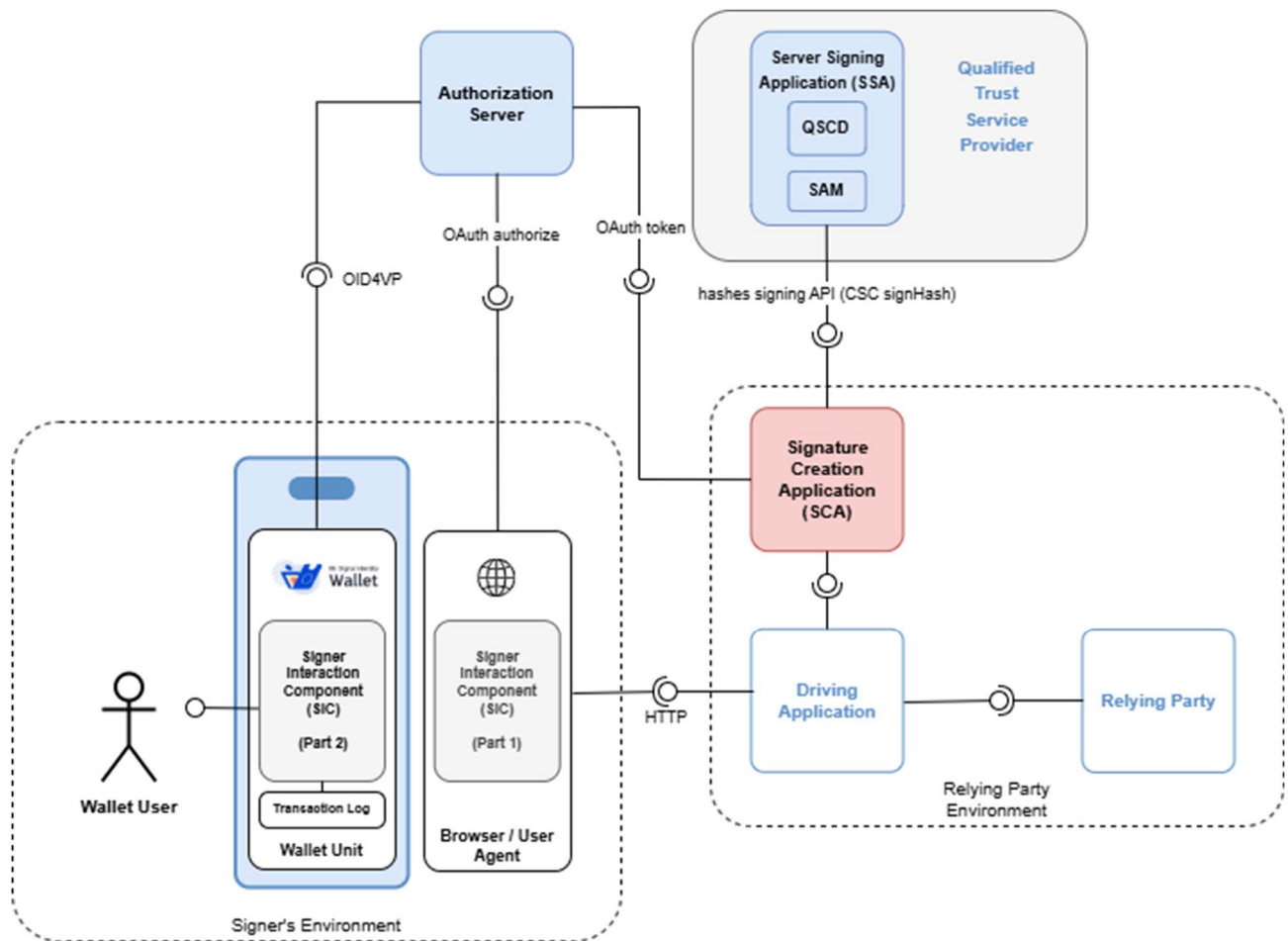


Figure 16: Architecture with DA and SCA at RP

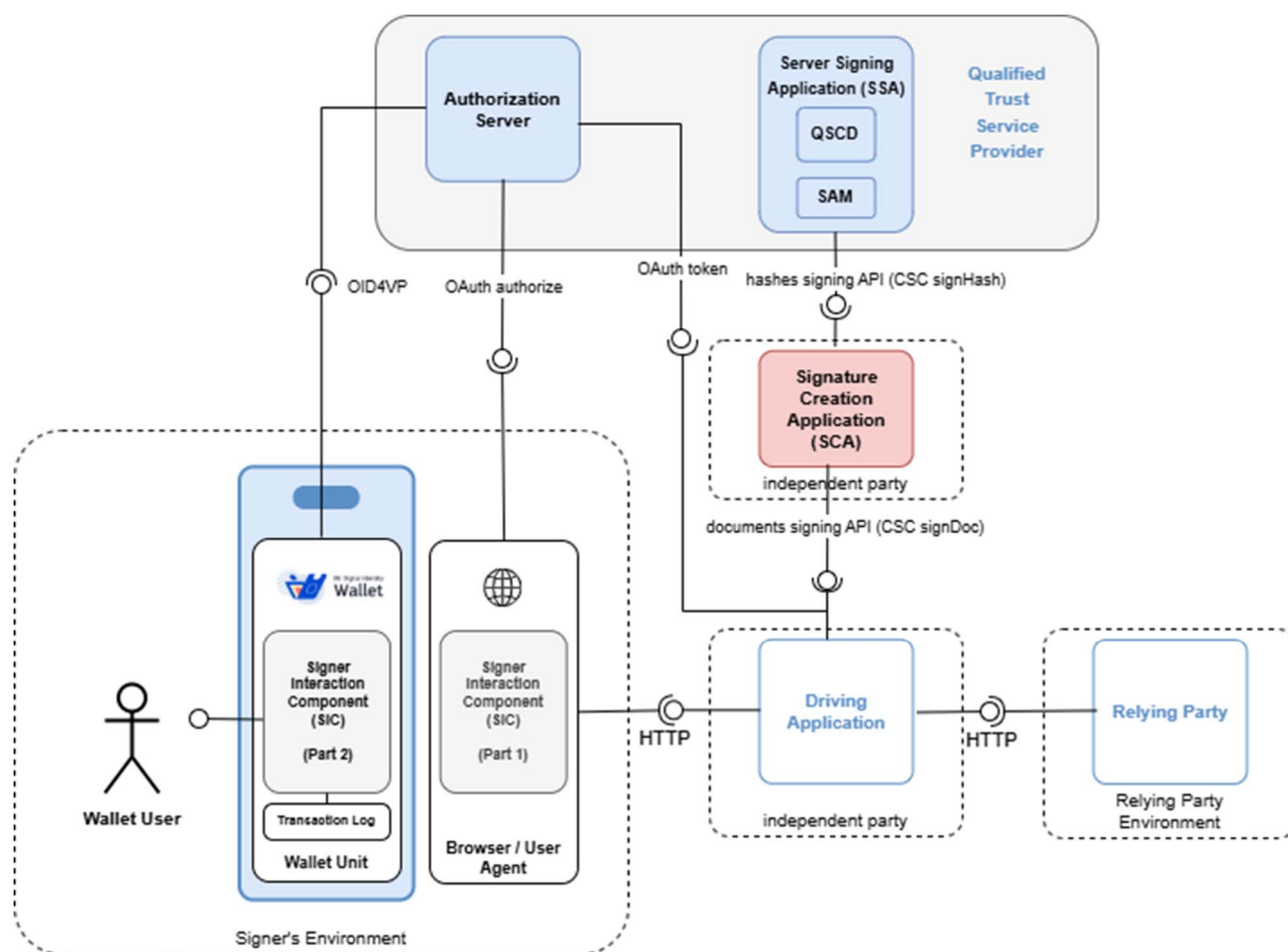


Figure 17: Architecture with authorization server at QTSP and with SCA and DA at different parties

In the above scenarios, the Authorization Server may transmit an OpenID4VP request to the EUDIW SIC component using the `openid4vp://` custom URL scheme, as specified in Section 12.1.2 of OpenID for Verifiable Presentations 1.0 [22], or via Digital Credentials API methods [23]. Additional delivery mechanisms may be supported if implemented by the EUDIW SIC component. The EUDIW processes the OpenID4VP request for example using PID presentation or transaction approval. The driving application can run in the local environment of the signer or in the environment of a relying party or in the environment of an independent intermediary. The driving application may request AdES signatures or digital signature values creation to the QTSP, i.e. by invoking CSC API [1].

Key Steps:

- **Document or Hash Transmission.** The driving application (it can be managed by a relying party, e.g. a bank, government agency, or business) sends the document or its hash to the authorization server requesting signatures creation authorization.
- **Signature Authorization.** The EUDIW is used by the user (signer) to authorize the signature operation. This typically involves user authentication and consent via the EUDIW.
- **Signature creation request.** The driving application requests AdES signatures or digital signature values creation to the QTSP.
- **Signature Creation.** If authorized, the QTSP uses a Qualified Signature Creation Device (QSCD) to generate the Qualified Electronic Signature (QES).
- **Return of Signed Document or Signed Data.** The signed document or signature is returned to the driving application.

Below a sequence diagram illustrating a generic signatures creation flow in an architecture where EUDIW serves as the authentication and authorization layer.

The driving application can be hosted at a relying party, at a TSP, or even an intermediary.

The signature creation application can be implemented at the driving application or at the signature creation service.

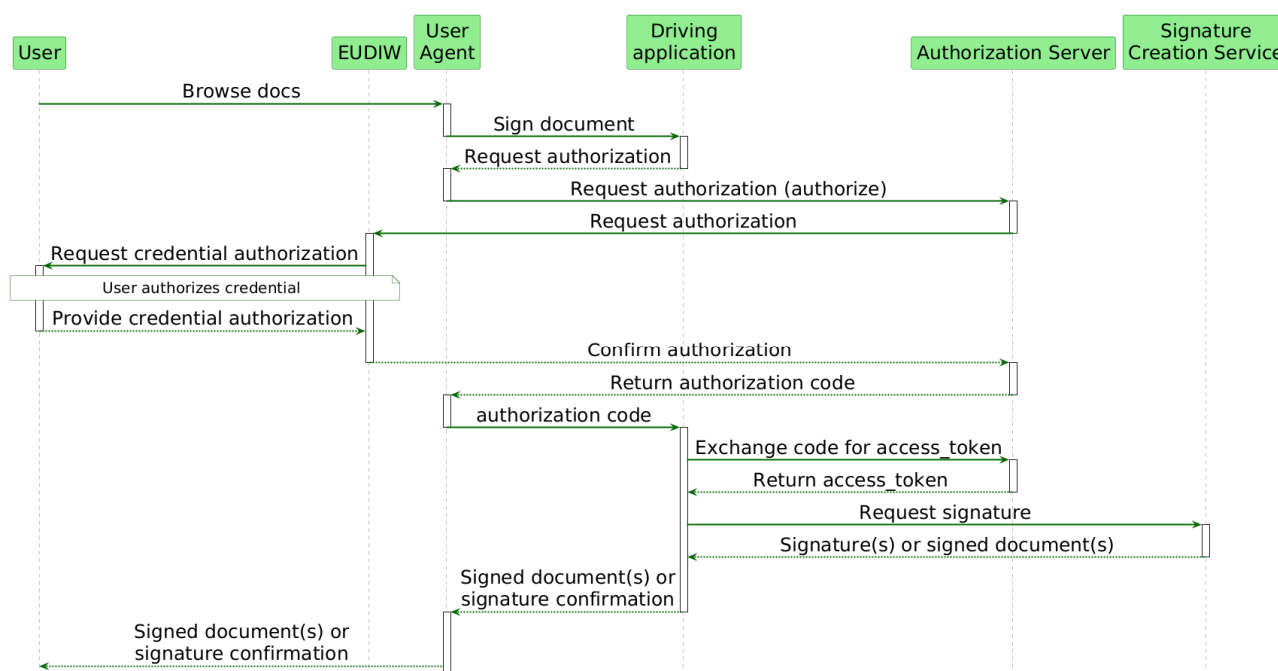


Figure 18: Generic simplified flow with EUDIW as authentication and authorization layer

Below another example of sequence diagrams illustrating a different flow in an architecture where EUDIW serves as the authentication and authorization layer.

The driving application can be hosted at a relying party, at a TSP, or even an intermediary.

The signature creation application can be implemented at the driving application or at the signature creation service.

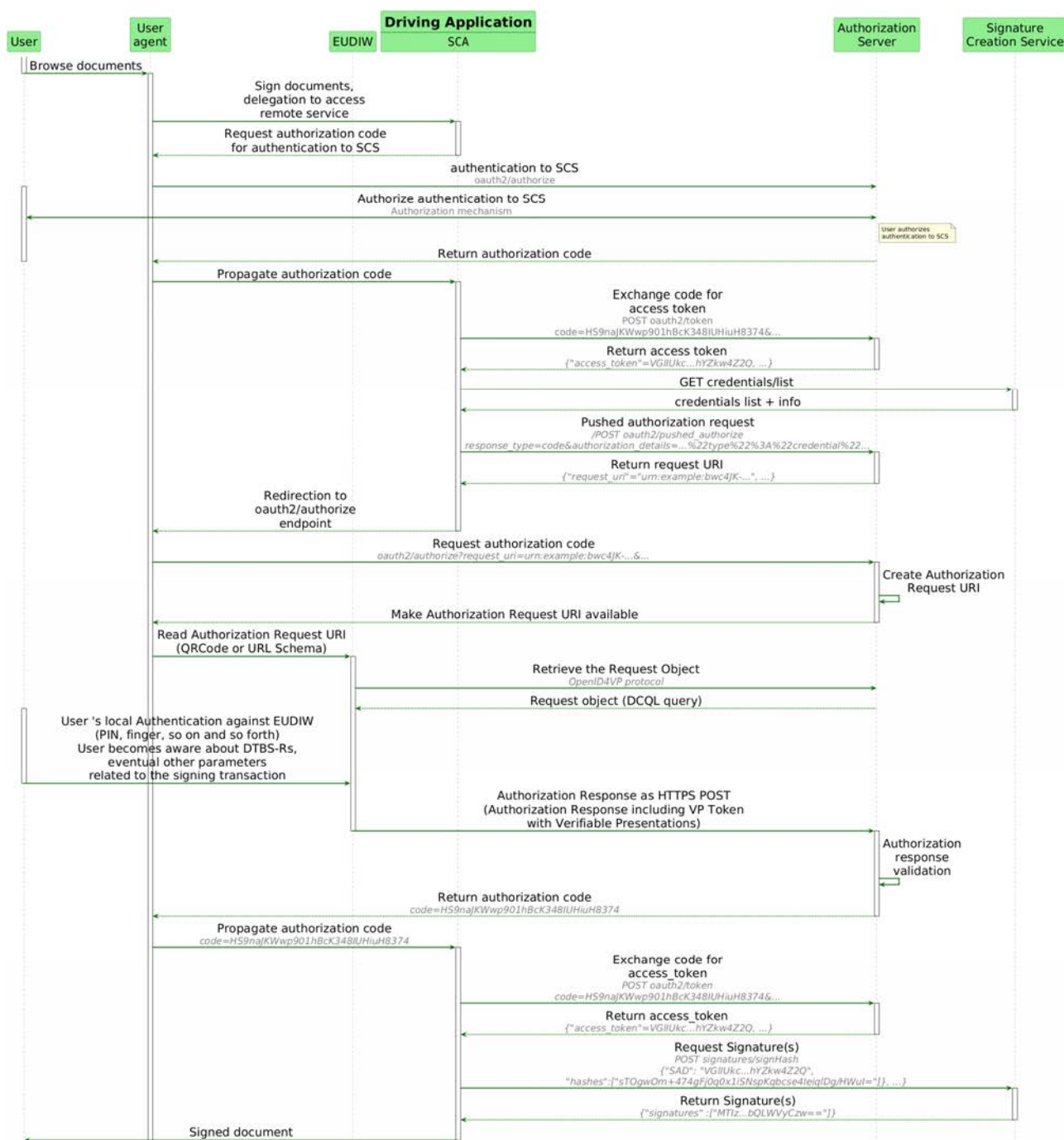


Figure 19: Remote signature creation with OAuth2 Authorization Code flow and Pushed and Rich Authorization Request with EUDIW as authentication and authorization layer

Service authentication and retrieval of the signing credential information may be omitted if the driving application knows in advance the user signing credential. The content of the request object passed to the EUDIW and of its response are specified in clause 7 of CSC data model bindings [3]. The transaction data type identifier for approving Qualified Electronic Signatures or seals (QES) creation is: "https://cloudsignatureconsortium.org/2025/qes-approval". The remote signatures creation may be performed by invoking the SSA CSC *signatures/signHash* or *signatures/signDoc* endpoints.

6.4.3 EUDIW participating in the signature creation process

In the model outlined in Figure 20, the EUDIW coordinates AdES signature creation through a SCASC, that may be part of the EUDIW or of the EUDIW backend, interacting with the QTSP-operated SSASC. The Wallet Relying Party (WRP) can pass data to be signed by the EUDIW through a structured and regulated process that ensures trust, transparency, and user control. The user's EUDIW plays the central role in managing the signature process to sign documents and to control and approve necessary transactions. The EUDIW collects consent and documents or DTBS/R to be signed, packages AdES, and requests the cryptographic signature value from the SSASC.

In the model outlined in Figure 21, the SCA is implemented in the same user device where the EUDIW is implemented too but is not part of the EUDIW itself.

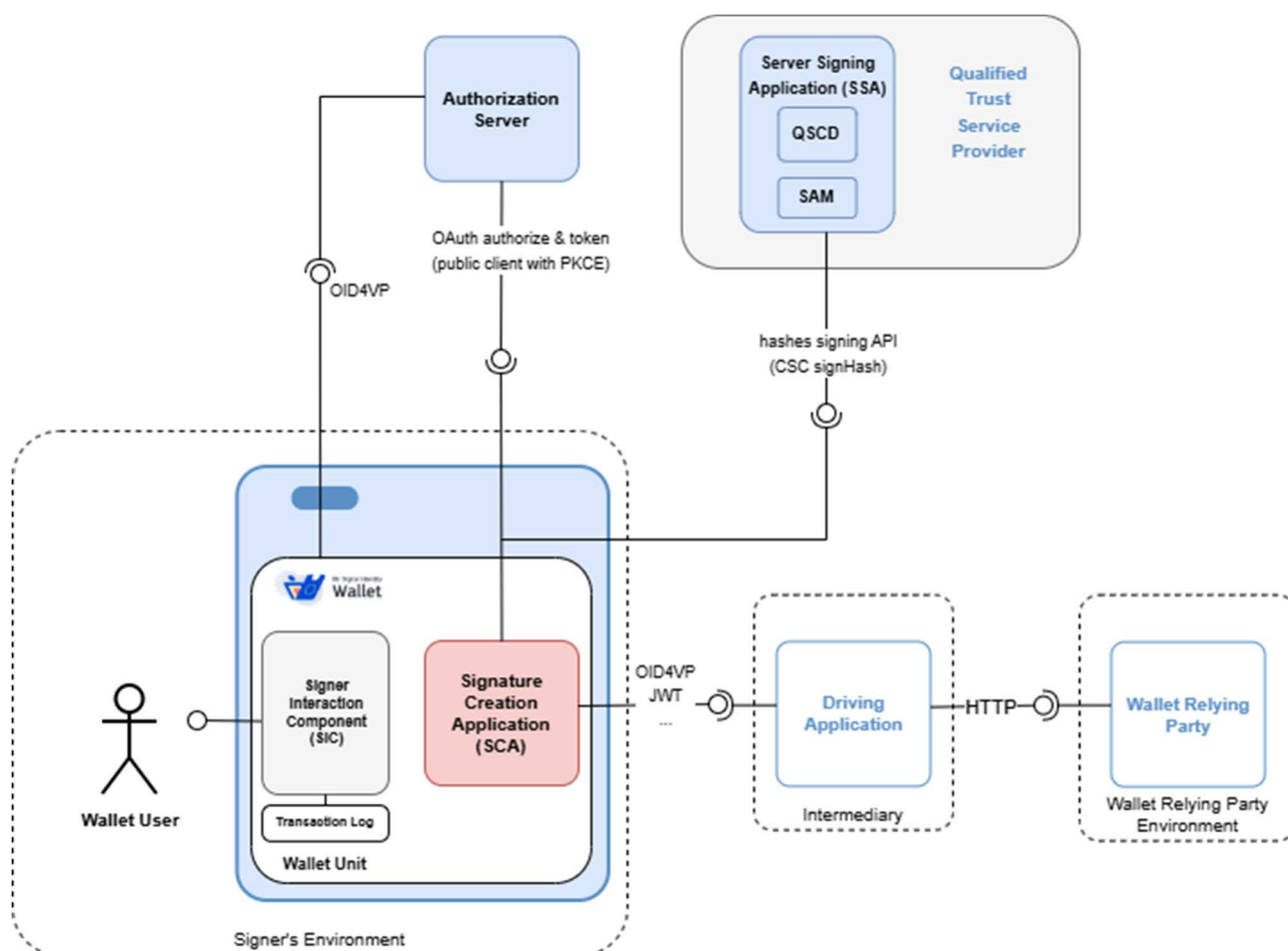


Figure 20: Architecture with SCA and DA at different independent intermediaries and with EUDIW integrating SCA and SIC

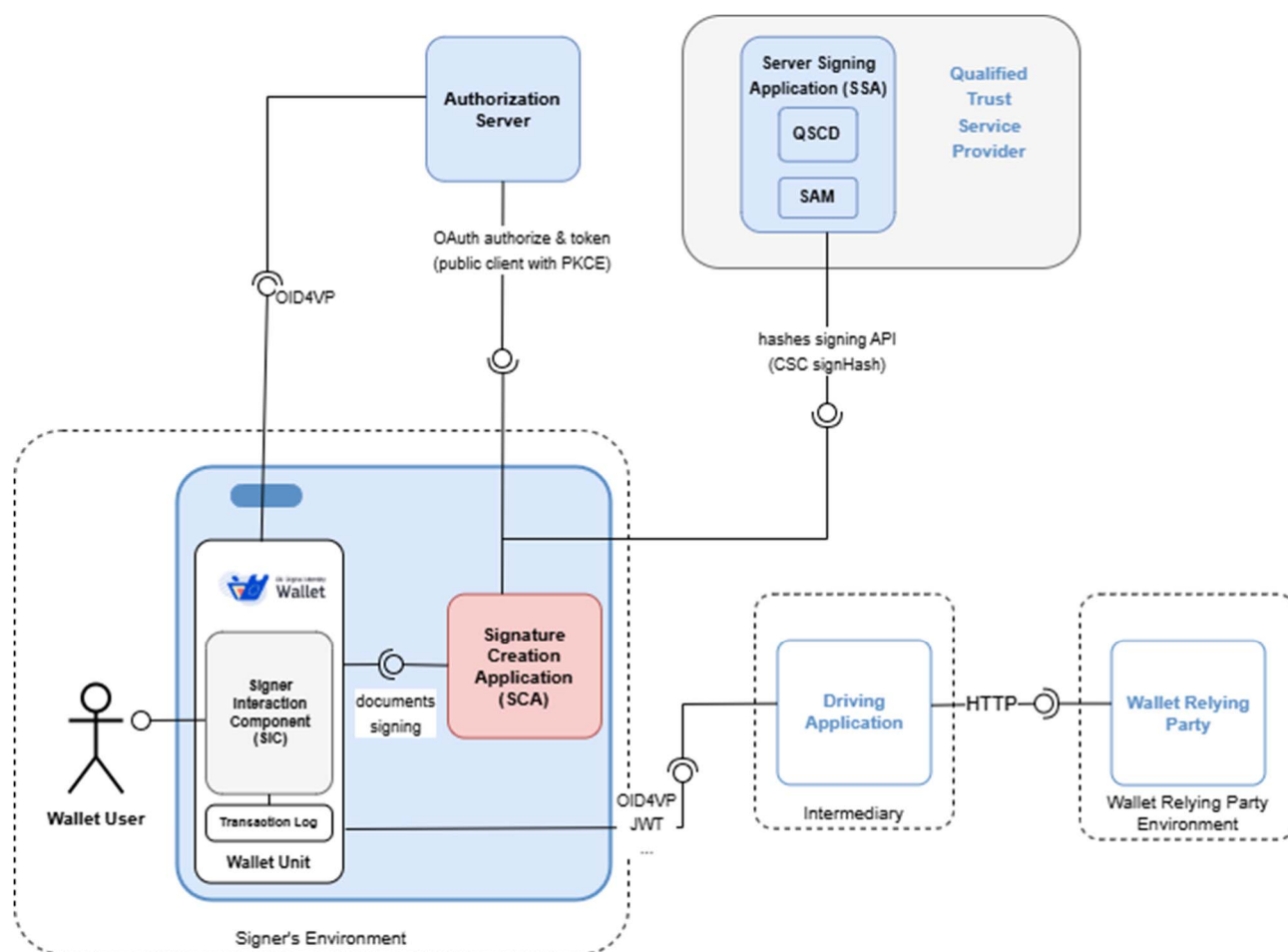


Figure 21: Architecture with SCA and DA at different independent intermediaries and with EUDIW and SCA in the same user device

Key Steps:

- **WRP Registration and Trust Establishment.** Before interacting with the EUDIW, the WRP shall:
 - register with a Wallet-Relying Party Registrar in its member state;
 - provide legal identifiers and contact details, a Registration Certificate specifying the intended use of the data and an Access Certificate proving its identity and authorization.
- **Presentation Request with Data to be Signed.** To pass data to be signed by the EUDIW, the WRP constructs a Presentation Request that includes:
 - the data to be signed (e.g. a challenge, transaction details);
 - the purpose of the request, aligned with the declared intended use;
 - the Access Certificate to authenticate itself.
- **The Presentation Request is sent to the EUDIW** either in a Same-Device flow (where WRP service and EUDIW run on the same device, e.g. using a deep link) or in a Cross-Device flow (where WRP service runs on one device, e.g. PC browser, and EUDIW runs on another, e.g. mobile app, often using QR codes) or in a Proximity-Based flow (e.g. using NFC or Bluetooth®).
- **Wallet Validation and User Consent.** The EUDIW validates the WRP's certificate and the trust chain, checks that the data request aligns with the RP's declared purpose, presents the request to the user for consent, if the user approves, the EUDIW signs the data using the appropriate credential or key and returns the signed data to the WRP.

In Figure 22 representing a simplified signature model with EUDIW integrating SCA and SIC, the data flows between different components are outlined.

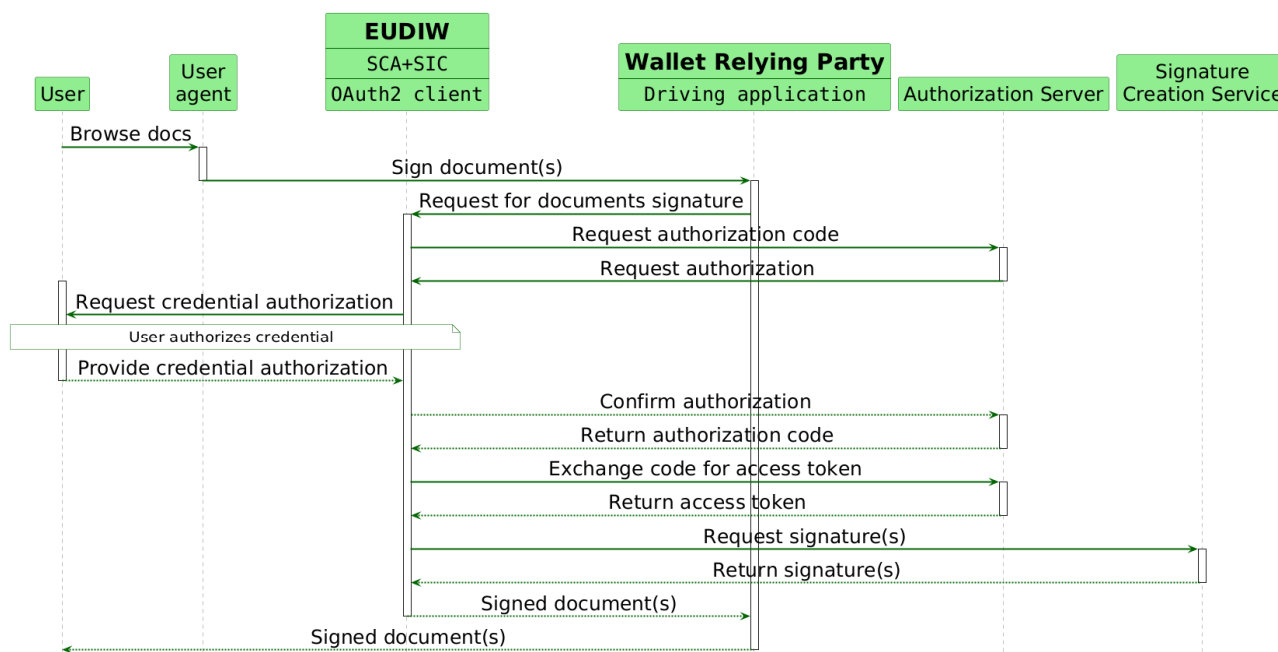


Figure 22: Generic simplified signature model flow with EUDIW integrating SCA and SIC

The content of the signature request object passed to the EUDIW and of its response are specified in clause 8 of CSC data model bindings [3]. The transaction data type identifier is <https://cloudsignatureconsortium.org/2025/qes> for requesting qualified electronic signatures or seals creation to EUDIW. Other more suitable request-response protocols for data other than verifiable presentations can arise in the future and could be used for this purpose. In annex A, a profile designing such a request-response protocol is defined.

6.4.4 EUDIW registration and authentication to authorization servers

6.4.4.1 Introduction

This clause defines the requirements for an EUDIW acting as an OAuth 2.0 client to register and authenticate to an authorization server using the OpenID Foundation FAPI 2.0 Security Profile [10], with explicit integration of Wallet Unit Attestation public key during dynamic client registration.

6.4.4.2 EUDIW registration

- The authorization servers shall implement the FAPI 2.0 Security Profile and IETF RFC 7591 [11], OAuth 2.0 Dynamic Client Registration Protocol.
- The authorization servers shall accept registration requests including client metadata values specified as claims in a software statement.
- The EUDIW shall register dynamically implementing requirements specified in IETF RFC OAuth 2.0 Dynamic Client Registration Protocol providing:
 - `redirect_uris`: the redirection URI strings for use in redirect-based flows;
 - `jwtks`: EUDIW's JSON Web Key Set as specified in IETF RFC 7517 [12], which contains the EUDIW's public keys, the WUA public key used by the EUDIW shall be included;
 - `token_endpoint_auth_method`: containing the value `private_key_jwt`;

- grant_types: the grant type strings that the EUDIW can use at the token endpoint, the value "authorization_code", representing the authorization code grant type defined in OAuth 2.0 clause 4.1, shall be included;
- response_types: the OAuth 2.0 response type strings that the EUDIW can use at the authorization endpoint, the value "code", representing the authorization code response type defined in OAuth 2.0 clause 4.1, shall be included;
- software_statement: metadata values about the EUDIW software including the following claims:
 - software_id: a unique identifier string used by registration endpoints to identify the client software to be dynamically registered, it shall include the attribute refNum, that specifies the reference number of the wallet solution, as defined in class WalletSolution as defined in the specification of systems enabling the notification and subsequent publication of EUDIW provider information;
 - client_name: human-readable string name of the EUDIW to be presented to the end-user during authorization, it shall include the string "European Digital Identity Wallet";
 - client_attestation (optional): an attestation that the authorization server may use to evaluate whether accepting or not the registration request depending on the trust relationship the authorization server has with the issuer of the software statement, if provided it shall include the EUDIW's Wallet Unit Attestation (WUA), allowing authorization servers to trust the integrity of digital identities and attributes stored within the EUDIW, in particular the public key included in the claim jwks, the authorization servers may ignore this claim.
- other optional client metadata parameters defined in OpenID Connect Relying Party Metadata Choices 1.0 - draft 04 [13] that the authorization server may ignore if not supported.
- The authorization servers may validate WUA evidence and reject registration if the attestation is invalid, expired, or revoked.
- The SCS shall provide the information stating if the client_attestation claim is mandatory or optional or not supported.
- Upon a successful registration request, the authorization server shall respond with an HTTP 201 "Created status" code and a body of type "application/json" with the following fields:
 - client_id: the OAuth 2.0 client identifier string that should not be valid for any other registered client, though an authorization server may issue the same client identifier to multiple instances of a registered client at its discretion;
 - all registered metadata about the EUDIW, including the software statement used as part of the registration request whose value shall be returned unmodified in the response.
- Upon an unsuccessful registration request, the authorization server shall return an HTTP 400 status code with content type "application/json" consisting of a JSON object (IETF RFC 8259 [i.9]), describing the error in the response body, with two members:
 - error: single ascii error code string.
 - error_description: human-readable ASCII text description of the error (optional).

6.4.4.3 EUDIW authentication

- The EUDIW shall implement OAuth 2.0 client capabilities.
- The EUDIW shall use Authorization Code Flow with PKCE (IETF RFC 7636 [14]).
- The EUDIW shall use PAR (IETF RFC 9126 [18]) with a signed JAR (IETF RFC 9101 [31]) for request integrity.
- The EUDIW authentication at PAR and token endpoints shall be supported via private_key_jwt as specified in Section 9 of OIDC.

- The EUDIW shall use state and nonce claims and validate issuer metadata (IETF RFC 8414 [15]) as specified in IETF RFC 9207 [24].

6.4.4.4 Security requirements

- The authorization server shall support issuance of sender-constrained access and refresh tokens by implementing the application-level mechanism Demonstrating Proof of Possession (DPoP), as specified in IETF RFC 9449 [25] as per FAPI.
- The EUDIW shall sign the DPoP according to JWT syntax with the private key whose public part is included in the WUA.
- The DPoP shall include the fields specified in clause 4.2 of IETF RFC 9449 [25].
- When creating the DPoP according to JWT syntax the recommendations provided in IETF RFC 8725 [26] should be followed.
- SCS should accept access tokens with DPoP proofs.
- If SCS accepts access tokens with DPoP proofs it shall validate any DPoP proof JWT.

6.4.4.5 Examples

6.4.4.5.1 Dynamic Client Registration request payload (JSON)

POST /oauth/register HTTP/1.1

Host: as.example.eu

Content-Type: application/json

```
{
  "client_name": "European Digital Identity Wallet (MemberState-X)",
  "application_type": "native",
  "redirect_uris": [
    "eudiwallet://cb",
    "https://wallet.example.ms/callback"
  ],
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "private_key_jwt",
  "token_endpoint_auth_signing_alg": "ES256",
  "jwks": {
    "keys": [
      {
        "kty": "EC",
        "crv": "P-256",
        "use": "sig",
        "alg": "ES256",
        "kid": "wua-key-2026-01",
        "x": "TCAER19Zvu30HF4j4W4vfSVoHIP1ILiLDls7vCeGemc",
        "y": "ZxjiWbZMQGHVWKVQ4hbSIirsvfuecCE6t4jt9F2HZQ"
      }
    ]
  }
}
{
  "tls_client_certificate_bound_access_tokens": true,
  "require_pushed_authorization_requests": true,
  "request_object_signing_alg": "ES256",
  "authorization_details_types": ["openid_credential_issuance"],
  "software_statement": "<signed-JWT>",
  "scope": "openid profile"
}
```

6.4.4.5.2 Authorization server response

HTTP/1.1 201 Created

Content-Type: application/json

Cache-Control: no-store

```
{
```

ETSI

Client authentication at PAR will use `private_key_jwt` or `DPoP`.

JWS header uses the WUA-attested key (`kid`).

6.4.4.5.4 PAR Response (from AS to EUDIW)

```
HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-store
{
  "request_uri": "urn:as:par:cx298dhasd98hasd9",
  "expires_in": 90
}
```

`request_uri` can only be used once, an authorization server ignores all authorization request parameters except `client_id` and `request_uri`.

6.4.4.5.5 Authorization Request (from EUDIW to authorization endpoint)

```
GET /authorize?
  client_id=client-12345&
  request_uri=urn:as:par:cx298dhasd98hasd9
HTTP/1.1
Host: as.example.eu
```

This request contains almost nothing except `client_id` and `request_uri`. All security parameters are inside the signed JAR that was previously pushed via PAR. `request URI` points to the previously registered JAR as stated in FAPI.

6.4.4.5.6 Authorization server redirects back with authorization code

If the user authenticates successfully and consents, the authorization server redirects:

```
HTTP/1.1 302 Found
Location: eudiwallet://cb?
  code=SplxIOBeZQYbYS6WxSbIA&
  state=af0ifjsldkj
```

`state` is returned exactly as provided by the client, `code` is bound to the client through sender-constrained token issuance with `DPoP`. Response mode is the default (query mode) because `response_type=code` only.

6.4.5 DA / SCA registration and authentication to authorization servers

6.4.5.1 Introduction

This clause defines the requirements for a DA or a SCA acting as an OAuth 2.0 client to register and authenticate to an authorization server. In the following clauses the generic term "client" will be used to refer to DA or SCA.

6.4.5.2 client registration

- The authorization server shall implement the FAPI 2.0 Security Profile and IETF RFC 7591 [11] OAuth 2.0 Dynamic Client Registration Protocol.
- The authorization server shall accept registration requests including client metadata values specified as claims in a software statement.
- The client may register dynamically implementing requirements specified in IETF RFC OAuth 2.0 Dynamic Client Registration Protocol providing:
 - `redirect_uris`: the redirection URI strings for use in redirect-based flows;

- `jwtks`: client's JSON Web Key Set as specified in IETF RFC 7517 [12], which contains the client's public keys, the client's registration certificate public key shall be included;
- `jwtks_uri`: URL string referencing the client's JSON Web Key Set as specified in IETF RFC 7517 [12], which contains the client's public keys, the client's registration certificate public key shall be included;
- `token_endpoint_auth_method`: containing the value `private_key_jwt`;
- `grant_types`: the grant type strings that the client can use at the token endpoint, the value `"authorization_code"`, representing the authorization code grant type defined in OAuth 2.0 clause 4.1, shall be included;
- `response_types`: the OAuth 2.0 response type strings that the client can use at the authorization endpoint, the value `"code"`, representing the authorization code response type defined in OAuth 2.0 clause 4.1, shall be included;
- `software_statement`: metadata values about the client software including the following claims:
 - `software_id`: a unique identifier string used by registration endpoints to identify the client software to be dynamically registered;
 - `client_name`: human-readable string name of the client to be presented to the end-user during authorization;
 - `client_attestation`: an attestation that the authorization server uses to evaluate whether accepting or not the registration request depending on the trust relationship the authorization server has with the issuer of the software statement, it shall include the client's registration certificate, allowing authorization servers to trust the integrity of the client's digital identities and attributes, in particular the public key included in the claim `jwtks`;
- other optional client metadata parameters defined in OpenID Connect Relying Party Metadata Choices 1.0 - draft 04 [13] that the authorization server may ignore if not supported.
- The authorization server shall validate WUA evidence and reject registration if the attestation is invalid, expired, or revoked.
- Upon a successful registration request, the authorization server shall respond with an HTTP 201 "Created status" code and a body of type "application/json" with the following fields:
 - `client_id`: the OAuth 2.0 client identifier string that should not be valid for any other registered client, though an authorization server may issue the same client identifier to multiple instances of a registered client at its discretion;
 - all registered metadata about the EUDIW, including the software statement used as part of the registration request whose value shall be returned unmodified in the response.
- Upon an unsuccessful registration request, the authorization server shall return an HTTP 400 status code with content type "application/json" consisting of a JSON object (IETF RFC 8259 [i.9]), describing the error in the response body, with two members:
 - `error`: single ascii error code string;
 - `error_description`: human-readable ASCII text description of the error (optional).
- The client may be configured with pre-registered (static) client authentication data. Appropriate in the case of closed ecosystems, driving applications issued by a national authority, authorization server and client belonging to the same trust framework. The client is manually registered out-of-band with the authorization server operator, who issues:
 - `client_id`.
 - one of:
 - `client_secret`.
 - a public key, if using `private_key_jwt`.

- a certificate, if using mTLS.

7 Remote signatures creation service API

7.1 Introduction

In the present document the API represent the way by which any driving application can interact with any SCS conforming to the present document. The following clauses define the API that a driving application can invoke within an SCS. For any API the request and response messages pair is specified referring to CSC API [1], when possible, otherwise detailing new components defined in the present document, by means of a table where the following information are provided:

- the components name;
- the presence of the component;
- the data type of the component;
- a brief description of the component.

The value included in the column "Presence" of the tables has the meaning defined in table 1.

Table 1

Value	Description
M	The component shall be included in the request to or response from the SCS.
O	The component may be included in the request to or response from the SCS.
C	The component shall be included in the request or response based on the occurrence of certain conditions.

7.2 Service information API

7.2.1 Description

Returns information about the remote service and the list of the API methods it supports. The *info* API method referenced in CSC API [1], Clause 11.1 shall be applied and fully implemented to ensure compliance with the present document.

7.2.2 Request message

7.2.2.1 Component for requesting service information

The message for requesting the service information shall contain the components defined in CSC API [1], clause 11.1 section Input.

7.2.3 Response message

7.2.3.1 Component for responding to service information requests

The message for returning the service information shall contain the components defined in CSC API [1], clause 11.1 section Output with the addition of the components defined in Table 2.

Table 2

Parameter	Presence	Type	Description
oauth2FAPISupport	O	boolean	This parameter shall be "true" if the SCS authorization server supports the FAPI 2.0 Security Profile and can - authenticate clients using private_key_jwt as specified in Section 9 of OIDC; - issue sender-constrained access tokens according to DPoP method as described in IETF RFC 9449 [25].

7.3 Credentials list API

7.3.1 Description

Returns the list of credentials of a certain user. A user may have one or multiple credentials hosted by a single RSSP.

If requested, it shall also return information about the signing certificate(s) and/or about the authorization mechanisms required to authorize the usage of the credentials for remote signing.

The *credentials/list* API method referenced in CSC API [1], clause 11.6 shall be applied and fully implemented to ensure compliance with the present specification.

7.3.2 Request message

7.3.2.1 Component for requesting credentials list

The message for requesting the list of credentials shall contain the components defined in CSC API [1], clause 11.6 section Input. The parameter *onlyValid* needs not be supported.

7.3.3 Response message

7.3.3.1 Component for responding to credentials list requests

The message for returning the requested list of credentials information shall contain the components defined in CSC API [1], clause 11.6 section Output. The parameter *onlyValid* needs not be supported if the input parameter of the same name is not supported.

7.4 Credentials info API

7.4.1 Description

Retrieves a credential and returns information about signing certificate and authorization mechanisms required to authorize the usage of the credential for remote signing.

The *credentials/info* API method referenced in CSC API [1], clause 11.7 shall be applied and may be implemented to ensure compliance with the present document.

7.4.2 Request message

7.4.2.1 Component for requesting credential information

The message for requesting the information about a credential shall contain the components defined in CSC API [1], clause 11.7 section Input.

7.4.3 Response message

7.4.3.1 Component for responding to credential information requests

The message for returning the service information shall contain the components defined in CSC API [1], clause 11.7 section Output.

7.5 Hash(es) signing API

7.5.1 Description

Computes the digital signature of one or multiple hash values. The driving application shall pass the authorization data that allows the signing credential usage.

The *signatures/signHash* API method referenced in CSC API [1], clause 11.13 shall be applied and fully implemented to ensure compliance with the present specification.

7.5.2 Request message

7.5.2.1 Component for requesting hash(es) signature creation

The message for requesting the hash(es) signature creation shall contain the components defined in CSC API [1], clause 11.13 section Input. The parameters *operationMode*, *validity_period* and *response_uri* need not be supported, so that only synchronous operation mode shall be supported by the SSASC.

7.5.3 Response message

7.5.3.1 Component for responding to hash(es) signature creation requests

The message for returning the signed hash(es) shall contain the components defined in CSC API [1], clause 11.13 section Output. The parameter *responseID* needs not be supported if the input parameter *operationMode* is not supported.

7.6 Document(s) signing API

7.6.1 Description

Creates one or more AdES signatures. The driving application shall pass the authorization data that allows the signing credential usage.

The *signatures/signDoc* API method referenced in CSC API [1], clause 11.14 shall be applied and should be implemented to ensure compliance with the present specification.

7.6.2 Request message

7.6.2.1 Component for requesting documents signature creation

The message for requesting the hash(es) signature creation shall contain the components defined in CSC API [1], clause 11.14 section Input. The parameters *operationMode*, *validity_period* and *response_uri* need not be supported, so that only synchronous operation mode shall be supported by the SSASC. The values "C", "X" and "J" that can be assigned to the parameter *signature_format* defined in the object *adesParameters* that is one of the objects that compose the object *signatureCreationRequest* need not be supported. The values "AdES-B", "AdES-T", "AdES-LT" and "AdES-LTA" that can be assigned to the parameter *conformance_level* defined in the object *adesParameters* that is one of the objects that compose the object *signatureCreationRequest* need not be supported.

7.6.3 Response message

7.6.3.1 Component for responding to documents signature creation requests

The message for returning the signed hash(es) shall contain the components defined in CSC API [1], clause 11.14 section Output. The parameter *responseID* needs not be supported if the input parameter *operationMode* is not supported.

7.7 Signature(s) creation polling API

7.7.1 Description

Returns the outcome corresponding to a previous signature(s) creation request when processed in asynchronous mode or the indication that the signing process has not yet been completed.

The *signatures/signPolling* API method referenced in CSC API [1], clause 11.15 shall be applied. The *signatures/signPolling* API method needs not be supported, unless signatures creation in asynchronous operation mode is supported by the SSASC.

7.7.2 Request message

7.7.2.1 Component for polling previous signature creation requests

The message for requesting the responses corresponding to a previous signature(s) creation request shall contain the components defined in CSC API [1], clause 11.15 section Input.

7.7.3 Response message

7.7.3.1 Component for responding to polling previous signature creation requests

The message for returning the outcome of a previous signature creation request shall contain the components defined in CSC API [1], clause 11.15 section Output.

7.8 Credentials creation API

7.8.1 Description

The *credentials/create* API method identified in CSC API [1], clause 11.4 creates a new signing credential. If requested, it can also return the new signing certificate, the whole associated certificate chain, additional information about the signing certificate and/or information about the authorization mechanism required to authorize the access to the credential for remote signing.

This method requires credential creation authorization.

The remote signature creation service shall collect the subject data needed for the certificate for the new credential.

The *credentials/create* API method referenced in CSC API [1], clause 11.4, shall be applied and fully implemented to ensure compliance with the present specification.

7.8.2 Request message

7.8.2.1 Component for requesting credential creation

The message for requesting a credential creation shall contain the components defined in CSC API [1], clause 11.4 section Input.

7.8.3 Response message

7.8.3.1 Component for responding to credential creation requests

The message for returning the information of the newly created credential shall contain the components defined in CSC API [1], clause 11.4 section Output.

7.8.4 Subject distinguished name of the new credential certificate

The remote signature creation service shall support the parameter *subjectData* given in the *credentialCreationRequest* input parameter in order to construct the subject distinguished name of the certificate for the new credential.

When Rich Authorization Requests (RAR) are supported, the remote signature creation service shall support receiving the subject data needed to construct the subject distinguished name of the certificate for the new credential from the authorization server in the authorization details of the access token expressed using OIDC claims notation. When Rich Authorization Requests (RAR) are supported, the remote signature creation service may support receiving the subject distinguished name data of the certificate for the new credential from the authorization server as custom claims in the access token [i.16] or retrieving them at a token introspection endpoint [i.8]. The remote signature creation service may ignore or overwrite all or some of the subject distinguished name data provided by the parameter *subjectData* given in the *credentialCreationRequest* input parameter.

The following subject data expressed using standard OIDC claims shall be collected by the remote signature creation service: *given_name*, *family_name*, *gender*, *birthdate*, *address:country*. The following subject data expressed using standard OIDC claims should be collected by the remote signature creation service: *sub* (if provided it shall contain a subject unique identifier assigned by a government or civil authority). The following subject data expressed using standard OIDC claims may be collected by the remote signature creation service: *address:locality*, *address:region*, *email*.

7.9 Credentials deletion API

7.9.1 Description

The *credentials/delete* API identified in CSC API [1], clause 11.5 deletes a signing credential.

This method requires credential deletion authorization. The *credentials/delete* API method referenced in CSC API [1], clause 11.5, shall be applied and may be implemented to ensure compliance with the present specification.

7.9.2 Request message

7.9.2.1 Component for requesting credential creation

The message for requesting a credential deletion shall contain the components defined in CSC API [1], clause 11.5 section Input.

7.9.3 Response message

7.9.3.1 Component for responding to credential deletion requests

This API has no output values and the HTTP status code 204 "No Content", defined in clause 15.3.5 of IETF RFC 9110 [16], is provided in response as specified in CSC API [1], clause 11.5 section Output.

8 Remote signatures creation service API based on OASIS DSS-X

8.1 Introduction and General Provisions

In the present document, the API represent the way by which any client application can interact with any SCS conforming to the present document. The following clauses define the API that a client application can invoke within an SCS. For simplifying API mapping comparison, the subsequent sections presents DSS-X API elements in the order of the above-cited CSC API elements (see clause 7 of the present document). The subsequent clauses also reference certain mandatory elements and thus implicitly define a profile of DSS-X with regard to the application context of the present document.

NOTE 1: Given the fact that OASIS DSS-X and CSC follow different protocol design paradigms, API elements may not match regarding all details of the abstract signature creation process. Nevertheless, and specifically due to the generic nature of DSS-X, numerous equivalent patterns exist for implementing the same approach using either protocol.

NOTE 2: DSS-X has an abstract syntax specification with a mapping to XML and an alternative mapping to JSON.

DSS-X can have various transport bindings as specified in section 8 of [4]. The present document recommends the HTTP Post binding (see section 8.1 of [4]), but also admits alternative transports.

DSS-X can have various security bindings to ensure authenticity, integrity and confidentiality of protocol messages as detailed in section 8.3 of [4]. Reliable DSS-X security bindings shall apply to all protocol messages in the context of the present document. In particular, TLS (see [7]) should supply the security base layer, while alternative and/or complementary security layers are admissible. For instance, a complementary security layer may consist in digitally signing DSS-X messages, with the chosen DSS-X syntax variant (see section 2.2.5 of [4]) determining an eligible signature format (see [8] and [9]).

DSS-X supports various options for transmitting signature activation data. The service metadata (see clauses 8.2 and 8.3 of the present document) shall provide details regarding the applicable mechanism embodied by one of the following variants:

- The *ClaimedIdentity* optional component (see section 4.4.9 of [4]) can identify the signatory regarding service calls described in clauses 8.3, 8.4, 8.5, 8.6 and 8.7 of the present document, with the *SupportingInfo* optionally carrying corresponding authorization data. This is the recommended approach when using the API in combination with the OAuth2.0 authorization framework (e.g. when using the API in the context of the European Digital Identity Wallet).
- The transport binding or an underlying security layer, or a combination of both can identify the signatory and carry authorization data. For instance, when using the API in combination with SAMLv2, an outer SAML envelope can provide identification and authorization data regarding the signatory of an inner DSS-X layer. Alternatively, an outer security layer can implicitly identify the signatory and provide authorization data. This latter approach is particularly suitable for an automated electronic seal creation service, when the service requestor is also the signatory.
- Finally, a sub-protocol that interleaves a DSS *SignRequest* and *SignResponse* scenario, may be used for identifying the signatory and providing authorization data.

8.2 Service information API

8.2.1 Description

Returns technical information about the service.

OASIS DSS-X [4] provides various options for conveying service-specific information as detailed in clause 4 of [5] that represents a complementary standard document of DSS-X.

The element name specification of clause 3.2 of [5] shall apply.

A metadata instance shall be as complete as necessary to enable integrators for obtaining accurate information regarding available feature support of the service.

For enabling metadata discovery, the mechanism described in clause 4 of [5] shall apply.

8.3 Credentials list API

8.3.1 Description

Returns the list of credentials of a certain user.

OASIS DSS-X [4] does not provide a specific API for requesting user credentials, which specifically protects privacy of the user by not exposing any credentials without explicit authentication or without user consent. By contrast, the metadata *AuthInfo* element (see clause 3.2 of [5]) can supply general information about the authentication mechanisms supported by the underlying signature service.

If the signer certificate is required upfront for preparing a signature request, e.g. for calculating a signer certificate hash as part of the DTBS or for specifying a PDF signature appearance based on the signer certificate subject Distinguished Name, the signature service may expose means for explicitly requesting the signer certificate. Such a request may consist of a *VerifyRequest* containing an *OptionalInputs* component with a *ReturnSignerIdentity* element (see clause 4.4.5 of [4]).

Alternatively, the response of the applied authentication protocol may supply the signer certificate.

Alternatively, the signer certificate may already be available on the client side by out-of-band means.

Alternatively, no signer certificate may be required upfront on the client side, e.g. when creating a signer certificate on the fly for single use, e.g. in the form of a validity-assured short-term certificate.

The metadata shall indicate via clause 3.1.5 of [5], which of the above-cited approaches the service supports.

8.4 Credentials info API

8.4.1 Description

Retrieves a credential and returns information about signing certificate and authorization mechanisms required to authorize the usage of the credential for remote signing.

See clause 8.3 of the present document.

8.5 Hash(es) signing API

8.5.1 Description

Computes the digital signature of one or multiple hash values.

For this purpose, the *SignRequest* with the *DocumentHash* variant of the *InputDocuments* component (see clause 4.5.5 of [4]) shall apply. The *KeySelector* component (see clause 4.4.12 of [4]) shall apply whenever the signing key cannot be determined implicitly.

8.6 Document(s) signing API

8.6.1 Description

Creates one or more AdES signatures.

For this purpose, the *SignRequest* with the *Document* variant of the *InputDocuments* component (see clause 4.5.3 of [4]) shall apply. For conveying more complex DTBS, particularly in the case of XAdES, the *TransformedData* variant (see clause 4.5.4 of [4]) may be more appropriate and is therefore an admissible alternative. The *KeySelector* component (see clause 4.4.12 of [4]) shall apply whenever the signing key cannot be determined implicitly.

8.7 Signature(s) creation polling API

8.7.1 Description

Returns the outcome corresponding to a previous signature(s) creation request when processed in asynchronous mode or the indication that the signing process has not yet completed.

For this purpose, the *PendingRequest* element (see clauses 4.3.5 and 7 of [4]) shall apply.

8.8 Credentials creation API

8.8.1 Description

The API *credentials/create* creates a new signing credential.

OASIS DSS-X does not provide a specific API for signer certificate creation. This happens implicitly when the corresponding *SignRequest* implementation creates the signer certificate on the fly for single use, e.g. in the form of a validity-assured short-term certificate upon user consent, while creation of long-term valid signer certificates is subject to a dedicated PKI process outside the scope of a signature creation process.

Consequently, a credential creation API is not applicable in the case of the present DSS-X profile.

Annex A (normative): OpenID4VP EUDIW-centric signing flow profile

A.1 Overview

The present annex defines a profile describing how an RP can request the creation of qualified electronic signatures/seals (QES) from a European Digital Identity Wallet (EUDIW) that integrates a Signature Creation Application, using OpenID for Verifiable Presentations (OpenID4VP) and OpenID for Verifiable Credentials (OID4VC) [i.14] as the transport/interaction protocols, and the CSC DM [2] and CSC data model bindings [3] for transaction data and results.

The profile focuses on:

- Requests: how an RP formulates a QES signing request towards the EUDIW (dcql_query + transaction_data) and optionally constrains acceptable X.509 credentials.
- Responses: how the EUDIW returns signatures inline (via VP Token) or out-of-band to a responseURI.
- Transaction data: required JSON structures for qesRequest and, when the Signature Activation Component runs server-side at the TSP, the complementary qesApproval flow.

This profile assumes AdES signature creation processing (PAdES/XAdES/CAdES/JAdES) per ETSI EN 319 102-1 [i.12].

The following flow is considered:

- the Relying Party (RP/Verifier) asks the EUDIW to create one or more electronic signatures,
- the EUDIW (user-held wallet) orchestrates signature creation, collects user consent, obtains any required credentials, calls a remote QTSP/signature creation service to create the signature(s), and
- the EUDIW returns the produced signature(s) (or signed documents) to the RP in the OpenID4VP authorization response or to a responseURI.

The profile defined in the present annex covers only the RP↔EUDIW interface (request + response). The internal EUDIW↔QTSP interaction is outside the scope of the profile defined in the present annex.

NOTE: The flow outlined here does not constitute the only possible approach for implementing a signing process in which requesting to an EUDIW the creation of QES and obtaining the corresponding QES in response.

A.2 Roles & Standards

- Relying Party (RP): verifier initiating a signature transaction to the EUDIW. Uses OpenID4VP authorization request with dcql_query and transaction_data. Provides documents by reference with integrity protection or as data URLs. Receives signatures from the EUDIW.
- EUDIW: implements OpenID4VP; holds or may obtain an X.509 signing credential, presents credentials, captures consent, verifies documents integrity before signing, interacts with a remote signature creation service via CSC API and returns signatures.
- Authorization Server (AS) / Signature Creation Service Provider (SCSP): runs server signing with Signature Creation Device (SCDev/QSCD); enforces signature policies; exposes CSC API semantics; the SAM may enforce sole control using Signature Activation Data (SAD).
- QTSP/SAM (out of scope here): remote signature creation service (JSON binding reusing CSC API constructs).

This profile aligns with the EUDI Wallet Architecture and Reference Framework, which foresees wallet-based QES creation and interoperable presentation protocols.

A.3 Profile scope & transport

The profile makes use of OAuth2 Authorization Request / Response as defined by OpenID for Verifiable Presentations 1.0.

In the flow RP→EUDIW a signed Request Object (JAR) is provided at request_uri. EUDIW resolves it after the user opens a deep link or scans a QR code.

In order to invoke the EUDIW, a custom URL scheme openid4vp:// may be supported in addition to other ways to invoke the EUDIW.

The response_type parameter value shall be vp_token. The response_mode parameter value should be direct_post or direct_post.jwt. The RP shall return redirect_uri in response to the HTTP POST request from the EUDIW, where the EUDIW redirects the user to, as defined in Section 8.2 of OpenID for Verifiable Presentations 1.0.

The Authorization Request is sent using the request_uri parameter as defined in JWT-Secured Authorization Request (JAR) IETF RFC 9101 [31]. The DCQL query and response parameters are defined in Section 6 of OpenID for Verifiable Presentations 1.0.

In the flow EUDIW→RP the response_mode=direct_post or direct_post.jwt values are used to return vp_token (and the signatures/results) in a signed POST body.

A.4 Profile identifiers

- Transaction data type for QES request: <https://cloudsignatureconsortium.org/2025/qes>
- Transaction data type for QES approval: <https://cloudsignatureconsortium.org/2025/qes-approval> (see Annex B)
- X.509 credential format identifier in OID4VC/VP: <https://cloudsignatureconsortium.org/2025/x509>

A.5 High-level interaction models

A.5.1 EUDIW: RP ↔ EUDIW, EUDIW ↔ QTSP

- 1) RP sends an OpenID4VP Authorization Request that includes:
 - a dcql_query selecting acceptable X.509 credentials, and
 - transaction_data carrying qesRequest (base64url-encoded JSON).
- 2) EUDIW renders transaction data, obtains consent, and interacts with the remote signature creation service to create the signatures.
- 3) EUDIW returns inline the signatures in the VP Token or posts out-of-band to the responseURI provided in the request.

A.5.2 QTSP: QTSP ↔ EUDIW

The QTSP (providing remote signature creation service) requests the EUDIW to present an attestation that includes qesApproval, cryptographically binding approval to the transaction data; the AS uses it to authorize the CSC API signing call (see Annex B).

A.6 Request construction (RP → EUDIW)

A.6.1 Introduction

The RP shall use OpenID4VP 1.0 and send an OAuth/OIDC authorization request with the parameters specified in the following sub-clauses.

A.6.2 Required parameters (OpenID4VP layer)

- response_type=vp_token and an appropriate response_mode (fragment, direct_post, or direct_post.jwt) per OpenID4VP.
- client_id, redirect_uri, state, nonce and dcql_query and transaction_data parameters shall be included as specified in OpenID4VP.

A.6.3 dcql_query (selecting an acceptable signing certificate)

The RP shall use the format identifier "https://cloudsignatureconsortium.org/2025/x509" and optionally constrain policies, keys, or fingerprints. A request for proof of possession of an X.509 certificate shall include transaction_data for creating electronic signatures or seals using the queried certificate.

A non-normative example DCQL query constraining certificate policies by using the X.509 credential format is:

```
{
  "dcql_query": {
    "credentials": [{
      "id": "xyz123",
      "format": "https://cloudsignatureconsortium.org/2025/x509",
      "meta": {
        "certificatePolicies": ["0.4.0.2042.1", "0.4.0.194112.1"]
      }
    }]
  },
  "transaction_data": [
    "ew0KICAgICJ0eXB1IjogImh0dHBzOi8vY2xvdWRzaWduYXR1cmVjb25zb3J0aXVtLm9yZy8yMDI1Lw..."
  ]
}
```

Another non-normative example DCQL query constraining certificate fingerprints by using the X.509 credential format is:

```
{
  "dcql_query": {
    "credentials": [{
      "id": "xyz123",
      "format": "https://cloudsignatureconsortium.org/2025/x509",
      "meta": {
        "certificateFingerprints": [{
          "hashValue": "Jal8JexCNec2FBjKNGNM6WSiiE44NWjbvJBHav+vCXU=",
          "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1"
        },
        {
          "hashValue": "ax8xsy0gdqyJtxmcw8xgZ85DbiC905oKdYBQspddxmK=",
          "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1"
        }
      ]
    }]
  },
  "transaction_data": [
    "ew0KICAgICJ0eXB1IjogImh0dHBzOi8vY2xvdWRzaWduYXR1cmVjb25zb3J0aXVtLm9yZy8yMDI1Lw..."
  ]
}
```

Requirements:

- format identifier shall be <https://cloudsignatureconsortium.org/2025/x509>;
- parameters in the meta parameter in the credential query shall have x509MetadataQuery semantics (certificateFingerprints, certificatePolicies, keys) as defined in CSC data model binding [3] clause 8.1.

A.6.4 transaction_data (the QES transaction)

transaction_data is a base64url-encoded UTF-8 JSON string whose decoded content is a qesRequest object as defined in CSC data model bindings [3] clause 6.2.1.

The transaction data type identifier for requesting QES is "https://cloudsignatureconsortium.org/2025/qes".

The parameter signatureRequests specifying an array of signatureRequest shall be included in the transaction data.

The parameter signatureQualifier shall be included in any signatureRequest object specifying the value "eu_eidas_qes" in order to indicate the applicable trust framework.

Any signatureRequest object shall include one of the following objects:

- documentData as specified in clause 8.1 of CSC DM
- documentReference as specified in clause 8.3 of CSC DM

The parameter label should be specified in documentData and documentReference objects in order to provide a human-readable description of the document to be signed to the EUDIW holder.

The following is a non-normative example of a transaction_data after base64url decoding:

```
{
  "type": "https://cloudsignatureconsortium.org/2025/qes",
  "credential_ids": ["xyz123"],           // optional, protocol-dependent
  "signatureRequests": [{
    "label": "Example Terms of Service",
    "access": { "type": "public" },
    "href": "https://public.rp-cdn.example/terms-and-conditions.pdf",
    "checksum": {
      "value": " HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
      "algorithmOID": "2.16.840.1.101.3.4.2.1"
    },
  },
  "signature_format": "P",
  "conformance_level": "AdES-B-B",
  "signed_envelope_property": "Certification",
  "signAlgo": "1.2.840.113549.1.1.1",
  "signatureQualifier": "eu_eidas_qes",
  "responseURI": "https://example.com/signatureResponse/123/"
}]
}
```

Requirements:

- type shall be <https://cloudsignatureconsortium.org/2025/qes>;
- signatureQualifier shall be present and shall indicate eu_eidas_qes (or the appropriate variant) for qualified signatures creation requests;
- each signatureRequests shall be a CSC DM signatureRequest with documentReference semantics (href, access, checksum) or documentData semantics (document, documentType) and include adesParameters semantics. Data URLs shall be supported; media types/size may be restricted;

- checksum parameter uses hash object syntax as defined in clause 7.4 of CSC DM;
- credential_ids may appear to bind the transaction to the credential selected in dcql_query (different from CSC API credentialID at SCSP);
- responseURI shall be included if an out-of-band POST is requested (see clause A.7.2). Otherwise, inline response in VP Token will be expected.

Access control (optional), to mitigate shoulder-surfing in cross-device flows, use one-time password (OTP):

```
"access": { "type": "OTP", "oneTimePassword": "51623" }
```

The EUDIW shall render the OTP so the user can release the protected resource from the RP's channel.

A.7 Response handling (EUDIW → RP)

A.7.1 Inline (VP Token) - x509PresentationResponse

If no responseURI is provided in the qesRequest, the EUDIW shall return signatures inline in the VP token response payload under the credential id, using qesResponse object as defined in CSC data model bindings [3] clause 6.2.2 with either documentWithSignature, when the signature is embedded (e.g. PAdES), or signatureObject otherwise when the signature is detached.

The following is a non-normative example of a response from an EUDIW:

```
{
  "xyz123": {
    "qes": {
      "documentWithSignature": ["<base64-encoded document 1>", "<base64-encoded document 2>"]
      // or
      // "signatureObject": ["<base64-encoded detached signature>"]
    }
  }
}
```

documentWithSignature is used when the signature is embedded (e.g. PAdES), signatureObject otherwise. Values are base64 (padded) by CSC API convention (even though OpenID4VP typically uses base64url).

A.7.2 Out of band (HTTP POST to responseURI)

If responseURI is present in the qesRequest, the EUDIW shall POST a JSON qesResponse to that HTTPS endpoint, using HTTP/1.1 and Content-Type: application/json, then return an empty presentation. Example response body:

```
{
  "documentWithSignature": ["<base64-encoded document>"]
}
```

The server should accept only the first successfully received response.

If response_mode=direct_post is set, the EUDIW POSTs response parameters. If, instead, response_mode=direct_post.jwt is set, the EUDIW still POSTs, but places all response parameters inside a signed JWT carried in a single form field named response.

A.7.3 EUDIW processing

After receiving the `dcql\_query` and the base64url-encoded `qesRequest` string inside the OpenID4VP authorization request, the wallet will authenticate the user, display a QES consent screen (with doc labels, OTPs, format/conformance, hash algorithm, etc.), and, upon consent, create the QES.

```
GET /authorize?
  response_type=vp_token&
```

```

client_id=https%3A%2F%2Frp.example%2Fclient&
redirect_uri=https%3A%2F%2Frp.example%2Fcb&
state=0f0c8d6a...&
nonce=8b4d2a67...&
// JSON-serialize the object from §3 and url-encode it as a single value
dcql_query={...}&
// Base64url of the JSON object from A.6.3
transaction_data=eyJ0eXB1IjoiaHR0cHM6Ly9jbG91ZHNpZ25hdHVyZWVbnNvcnR... HTTP/1.1
Host: wallet.example

```

The EUDIW

- 1) Resolves and verifies the JAR, including signatures.
- 2) If documents are provided by reference, fetches and verifies the integrity hash; computes DTBSR per format (e.g. PAdES ByteRange). (DTBS/DTBSR definitions according to ETSI EN 319 102.1 [i.12].)
- 3) Shows the user: requested credentials, each document name + DTBSR (hash + alg), signature format/policy, and the RP origin; captures explicit consent.
- 4) Presents credentials as requested (DCQL) using OpenID4VP and, internally, invokes the QTSP (as specified in the present document) with the hashes (DTBSRs) and the requested signature parameters (policy/format/alg).
- 5) Post processes PAdES embedding if return mode is embedded; otherwise keeps detached CMS.
- 6) Returns the OpenID4VP Authorization Response including signatures or result references.

A.8 Transaction data processing & UX rendering (EUDIW)

Processing

- If a signatureRequest contains both href and checksum, the EUDIW shall verify integrity and abort on mismatch.
- If a qesRequest references an X.509 credential that cannot produce a QES for the requested signatureQualifier, the EUDIW shall abort.
- The EUDIW should log transaction data and the user's approval/rejection.

EUDIW shall clearly render: the type (QES), the trust framework (eIDAS), whether Signature vs Seal, document labels, href (and whether integrity was verified), signature format/conformance level, and any signed attributes; and show responseURL.

A.9 Complementary approval flow

The SCSP AS may require a separate qesApproval bound to the transaction data. See Annex B.

A.10 Security & privacy requirements

- JWT signing and issuer validation: if an RP transports a CSC DM signatureRequest as a JWT, it shall be signed and the EUDIW shall validate signature and issuer authorization.
- Checksum: use resources integrity with SHA-256 or stronger; EUDIW verifies before signing.
- Encoding rules: OpenID4VP parameters (transaction_data) are base64url encoded; CSC API data elements inside remain base64 (padded).

- ## A.11 Conformance checklist (RP)

- ## A.12 Examples

A.12.1 Authorization request (core parameters)

(Whitespace added for readability.)

A.12.2 Example 'gesRequest' (transaction data before base64url encoding)

ETSI

```

    "signed_envelope_property": "Certification",
    "signatureQualifier": "eu_eidas_qes",
    "signAlgo": "1.2.840.113549.1.1.1"
  },
  {
    "label": "Annex A - JSON config",
    // Provide the object inline; wallet still verifies checksum
    "href": "data:application/json;base64,eyJleGFtcGxLS2V5IjoizXhhbXBsZVZhbnVlIn0K",
    "signature_format": "J",
    "conformance_level": "AdES-B-B",
    "signed_envelope_property": "Attached",
    "signAlgo": "1.2.840.113549.1.1.1",
    "checksum": {
      "value": "cuKv8Ee9H/rQsteQ1MQZ2Ld2ERXRkkulihFh3/X0XFQ=",
      "algorithmOID": "2.16.840.1.101.3.4.2.1"
    },
    "signatureQualifier": "eu_eidas_qes"
    // OPTION A: inline delivery (omit responseURI) -> wallet returns QES inside vp_token
    // "responseURI": undefined
    // OPTION B: out-of-band delivery (recommended for large PDFs):
    // "responseURI": "https://rp.example/qes/receive"
  ]
}

```

Notes & rules:

- input mode:
 - by-reference: the Wallet fetches the document and computes DTBSR itself; integrity is assured via sha256 over the fetched bytes;
 - inline: large payloads discouraged; if used, provide base64, mime, size.
- return mode:
 - embedded → EUDIW returns PAdES embedded PDF to the RP (by reference or inline);
 - detached → EUDIW returns a detached CMS signature over DTBSR. (PAdES/XAdES/CAdES policies follow ETSI EN 319 102-1 [i.12]);
 - when responseURI is used, the wallet POSTs the `qes` response to that endpoint, mutually authenticated TLS is recommended.

A.12.3 VP Token (inline) - PAdES result

```

{
  "xyz123": {
    "qes": {
      "documentWithSignature": ["<base64-encoded PDF with embedded PAdES>"]
    }
  }
}

```

A.12.4 Out of band POST to responseURI

POST /signatureResponse/123/ HTTP/1.1
 Host: rp.example
 Content-Type: application/json

```
{  
  "documentWithSignature": ["<base64-encoded PDF with embedded PAdES>"]  
}
```

The Wallet returns an empty presentation to the OpenID4VP flow.

Annex B (normative): EUDIW QES creation approval profile

B.1 Overview

This profile specifies how a QTSP's remote signature creation service can request approval for Qualified Electronic Signature/Seal creation from an EUDI Wallet (EUDIW) using the OpenID for Verifiable Presentations (OpenID4VP) protocol and the CSC data model bindings [3]. It defines requests, responses, and the transaction_data structure that binds the user's consent to concrete data to be signed and to the target credential/remote QSCD operation.

It aligns with:

- CSC DM, for signatureCreationApproval object and document digests, and CSC data model bindings [3] defining <https://cloudsignatureconsortium.org/2025/qes-approval> transaction data type identifier and qesApproval data component.
- EN 419 241-1 [6] (SAM, SAD and sole control) as architectural baseline for remote QSCD; the approval is used by the AS/SAM to authorize the signature.
- ETSI EN 319 102-1 [i.12] for AdES creation/validation procedures referenced by QTSP policies.

B.2 Roles and architecture

- QTSP Remote Signing Server (RSSP-centric): runs Authorization Server (AS) + SAM in front of a QSCD. The AS is the Verifier in OpenID4VP and asks the EUDIW for a presentation of a credential containing QES Approval. The SAM uses the token obtained with the code provided by the AS after successful validation of qesApproval to authorize the key operation. The QTSP creates and issues the attestation whose presentation is requested to EUDIW to authorize signatures creation.
- EUDI Wallet (Driving Application): receives the approval request (OpenID4VP), renders the transaction data, collects user consent, and returns a presentation embedding qesApproval per credential format (ISO mdoc or SD-JWT VC).
- QSCD: Performs the private key operation once the SAM validates the approval and policy constraints.

B.3 Signature authorization attestation

The QTSP managing SCS creates and issues this attestation.

Recommended minimal claims:

Claim	Purpose
iss	QTSP identifier
sub	Wallet holder pseudonymous identifier (NOT PID)
purpose_of_use	It shall be "signature creation authorization"
certificate_binding	Reference to the signing certificate or key to be used
valid_for	One signing operation or limited timeframe
aud	The Authorization Server / Signing Service
nonce / challenge	Prevent replay

It ensures binding to the certificate or signing key.

Two recommended bindings:

- (A) Binding to the signing certificate including one of:

- signing certificate hash (SHA 256)
- SKID / AKID
- public key hash

(B) Binding to the wallet instance (optional) including a proof-of-possession element similar to:

- jwks (as used in WUA)

This prevents forwarding or misuse.

B.4 Profile scope & transport

The profile makes use of OAuth2 Authorization Request / Response as defined by OpenID for Verifiable Presentations 1.0.

In the flow RSSP→EUDIW a signed Request Object (JAR) is provided at request_uri. EUDIW resolves it after the user opens a deep link or scans a QR code.

In order to invoke the EUDIW, a custom URL scheme openid4vp:// may be supported in addition to other ways to invoke the EUDIW.

The response_type parameter value shall be vp_token. The response_mode parameter value should be direct_post or direct_post.jwt. The RP shall return redirect_uri in response to the HTTP POST request from the EUDIW, where the EUDIW redirects the user to, as defined in Section 8.2 of OpenID for Verifiable Presentations 1.0.

The Authorization Request is sent using the request_uri parameter as defined in JWT-Secured Authorization Request (JAR) IETF RFC 9101 [31].

B.5 Profile identifiers

- Approval transaction data type: <https://cloudsignatureconsortium.org/2025/qes-approval> (qesApprovalRequest).
- Approval result name: qesApproval (a base64-encoded hash digest bound to the original request).
- Credential format identifiers (for the presented credential carrying qesApproval), e.g. ISO mdoc or SD-JWT VC as defined in CSC data model bindings [3].

B.6 Data model for approval

B.6.1 Introduction

The SCSP shall use OpenID4VP 1.0 and send an OAuth/OIDC authorization request with the parameters specified below.

B.6.2 qesApprovalRequest

The transaction data for requesting authorization to a signature creation shall include the qesApprovalRequest object specified in CSC data model bindings [3], clause 7.1.1.

A signatureCreationApproval object, specified in CSC DM clause 10.1, representing information required by the signer to give consent to a signature creation, shall be included in any qesApprovalRequest object. Differently from signatureCreationApproval object specification stated in CSC DM clause 10.1 the parameter documentDigests is defined as Array of union of documentInfo and documentReference.

The transaction data type identifier for approving Qualified Electronic Signatures or seals (QES) is: "https://cloudsignatureconsortium.org/2025/qes-approval". The transaction data shall be associated with a credential for approval of QES creation that can be identified by a field `credential_ids` as specified in CSC data model bindings [3] clause 7.1. One way to identify the credential is an `x509MetadataQuery` object as specified in CSC data model bindings [3], clause 8.1.

The following is a non-normative example of a `transaction_data` after `base64url` decoding

```
{
  "type": "https://cloudsignatureconsortium.org/2025/qes-approval",
  "credential_ids": ["xyz123"],
  "credentialID": "GX0112348",
  "signatureQualifier": "eu_eidas_qes",
  "numSignatures": 2,
  "documentDigests": [{
    "label": "Example Contract",
    "hash": "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
    "hashType": "sodr",
    "access": { "type": "OTP", "oneTimePassword": "51623" },
    "href": "https://protected.example/doc-01.pdf?token=...",
    "checksum": {
      "value": " HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
      "algorithmOID": "2.16.840.1.101.3.4.2.1"
    }
  }],
  {
    "label": "Terms of Service",
    "hash": "HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
    "hashType": "sodr",
    "access": { "type": "public" },
    "href": "https://public.example/tos.pdf",
    "checksum": {
      "value": " HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
      "algorithmOID": "2.16.840.1.101.3.4.2.1"
    }
  }
],
  "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1"
}
```

NOTE 1: `credential_ids` binds to an EUDIW-local identifier per OpenID4VP (distinct from the SCSP's `credentialID`).

NOTE 2: `signatureCreationApproval`, defined in CSC DM, includes the QTSP `credentialID` (if known) or at least `signatureQualifier`, `numSignatures`, and an array of `documentDigests` with the digests and optional `href/checksum/access` for render and integrity checks. `hashAlgorithmOID` identifies the digest algorithm.

NOTE 3: `hashType` values like `sodr/dtbsr` follow CSC DM semantics; integrity checksum is base64 and includes algorithm prefix per CSC DM.

B.6.3 Computing `qesApproval` (EUDIW)

- The EUDIW computes `qesApproval` = `base64(hash(original UTF-8 qesApprovalRequest))`; the hash algorithm is the request's `hashAlgorithmOID`. The encoding of hash input preserves the original UTF-8 JSON (decode `base64url` if `transaction_data` came that way).
- Embedding depends on the approval credential format:

- ISO/IEC 18013-5 mdoc [i.21]: put qesApproval as a DeviceSigned data element under namespace org.cloudsignatureconsortium.dm.1, identifier qesApproval, value type bstr of the raw digest (no base64 inside CBOR).
- SD-JWT VC [i.13]: include a top-level claim org.cloudsignatureconsortium.dm.1.qesApproval in the Key Binding JWT; value is a String (base64 of the digest).

B.7 OpenID4VP request (QTSP → Wallet)

The SCSP's AS, acting as Verifier, sends an OpenID4VP authorization request:

- response_type=vp_token; choose response_mode (fragment, direct_post, or direct_post.jwt) for same-device vs cross-device.
- include dcql_query selecting the approval credential to be presented (e.g. a service attestation issued to the user that supports qesApproval). The query can request specific claims/doctype needed to carry qesApproval.
- include transaction_data = base64url (JSON qesApprovalRequest as in clause B.6.2).

Example (decoded)

```
{
  "dcql_query": {
    "credentials": [{
      "id": "qscd_service_attestation",
      "format": "mso_mdoc",
      "meta": { "doctype_value": "com.example.service.1.attestation" },
      "claims": [
        { "path": ["com.example.service.1", "userName"], "values": ["willeke"] },
        { "path": ["com.example.service.1", "credentialID"], "values": ["GX0112348"] },
        { "path": ["org.cloudsignatureconsortium.dm.1", "qesApproval"] }
      ]
    }]
  },
  "transaction_data": [ "<base64url of qesApprovalRequest JSON>" ]
}
```

B.8 EUDIW response (EUDIW → QTSP)

B.8.1 VP Token / DC API response

The EUDIW returns a verifiable presentation containing the selected approval credential with qesApproval set. For ISO mdoc this is a DeviceResponse; for SD-JWT VC it is the presentation with a Key Binding JWT that includes the claim org.cloudsignatureconsortium.dm.1.qesApproval.

Example (conceptual)

```
{
  "qscd_service_attestation": [
    "<base64url-encoded DeviceResponse or SD-JWT VC presentation with qesApproval>"
  ]
}
```

The SCSP's AS validates the presentation like a normal Verifier and extracts qesApproval.

B.8.2 Using qesApproval at the AS/SCSP

After successful validation, the AS verifies:

- The digest in qesApproval equals hash (the exact qesApprovalRequest it sent) under hashAlgorithmOID.
- The bound credentialID/user identity in the approval credential matches the targeted remote QSCD credential and policy. Then the AS/SAM authorizes the signature (e.g. CSC API signatures/signHash or signatures/signDoc).

B.9 Processing & rendering requirements (EUDIW)

- Render: Show QES vs Seal, trust framework (eIDAS), document labels, href, and whether integrity (checksum) was verified; show OTP if present so the user can release protected resources.
- Abort if: hash integrity check fails, number of items differs from numSignatures, or the requested signatureQualifier cannot be produced with the approval credential/remote QSCD.

B.10 Security & privacy

- JWT/issuer: if the SCSP transports the approval request in a JWT, it shall be signed; the EUDIW shall validate signature and issuer authorization.
- Encodings: transaction_data outer is base64url encoded; integrity checksum and any binary items defined by CSC DM remain base64 (padded).
- AS/SAM: enforce sole control via SAD and trusted path; use qesApproval only within its lifetime/purpose.

B.11 Conformance checklist (SCSP)

- 1) Build OpenID4VP request with dcql_query asking for an approval credential (doctype/claims supporting qesApproval).
- 2) Include transaction_data = base64url(JSON) of qesApprovalRequest with signatureCreationApproval (labels, hashes, hashAlgorithmOID, counts).
- 3) Validate the returned presentation and extract qesApproval bound to the exact bytes of the request.
- 4) Authorize the QSCD signing step at the AS/SAM only if qesApproval and user/credential bindings match policy (EN 419 241-1 [6]).

B.12 Working sequence

- 1) SCSP AS → EUDIW: OpenID4VP Authorization Request with dcql_query (service attestation) + transaction_data (qes-approval).
- 2) EUDIW: Renders data, user approves; computes qesApproval; produces presentation.
- 3) EUDIW → SCSP AS: VP/DeviceResponse with qesApproval.
- 4) AS/SAM: Validates presentation; verifies qesApproval; calls QSCD (e.g. CSC API signatures/signHash and/or signatures/signDoc) to create the QES.

Annex C (normative): Specific recommendations for qualified signatures and/or seals creation

C.1 Introduction

This annex provides formal recommendations for implementing Qualified Electronic Signature (QES) creation workflows, while ensuring a clear separation between the authentication/authorization layer and the signature creation API layer.

C.2 Separation of Responsibilities

Authentication and authorization should be performed outside the SCS using high-assurance mechanisms (e.g. OAuth 2.0, OIDC, FAPI 2.0). The SCS is responsible exclusively for signature creation within the remote QSCD operated by the QTSP.

C.3 Authentication and Authorization Layer

The authentication/authorization layer responsible for user identity verification should implement OAuth 2.0/OIDC with PAR and PKCE, and `private_key_jwt` authentication mechanisms.

C.4 Signature Creation Service

The SCS that provides standardized operations for credential listing, credential information retrieval, credential authorization, and signature creation should validate authorization tokens and DTBS binding.

C.5 Strong Binding Between Consent and DTBS

Authorization tokens should embed specific DTBS hashes, credential identifiers and signature parameters to ensure non-repudiation and prevent replay attacks.

C.6 Use of FAPI 2.0 for EUDIW-Based Clients

When the EUDIW acts as the OAuth client, FAPI 2.0 mechanisms should be adopted: `private_key_jwt` with WUA keys, PAR for request integrity, and mTLS/DPoP for sender-constrained access tokens.

C.7 SCS API Security Recommendations

All SCS API calls should be authenticated using OAuth access tokens, possibly following mTLS or DPoP constraints, validating nonces, and enforcing one-time-use authorization objects.

Annex D (informative): Change history

Date	Version	Information about changes
March 2019	1.1.1	Publication as ETSI TS 119 432
June 2020	1.1.2	CR#1 - Alignments with dss-core-v2.0-cs02 CR#2 - Correction of errors found in the 119 432 JSON schema CR#3 - Alignment with draft CSC API v2.0
February 2026	1.2.13	CR#1 - Definition of authentication and authorization mechanisms CR#2 - Definition of signature schemes involving the EUDIW CR#3 - Alignment with CSC API v2.2

History

Version	Date	Status
V1.1.1	March 2019	Publication
V1.1.2	June 2020	Publication
V1.3.1	March 2026	Publication