

DIGITALISERINGSSTYRELSEN



NL3 Broker Implementation Guide-lines

Contents

Changelog.....	2
References.....	2
1 The Purpose and Target Audience of the Document.....	3
1.1 Document Conventions	3
1.2 Abbreviations.....	3
2 Motivation.....	4
3 Introduction.....	5
4 Broker Integration – overview	7
4.1 Broker signing flow in detail	7
5 Broker Example Code	10
6 Broker Integration – details	11
6.1 Common Steps.....	11
6.2 PAdES Signature.....	13
6.3 XAdES Signature.....	16
7 Signing Formats and Data Models	17
7.1 Signature Parameters	17
7.2 Signing Payload	18
7.3 Signing Client Error	18
8 SAML Integration.....	19
8.1 SAML Assertion requirements	20
8.2 Subject	20
8.3 Conditions.....	21
8.4 AttributeStatement	21
8.5 Signing Certificate Types.....	25
9 Logging	27
10 Response codes and error handling.....	28
11 Security Guidelines.....	28
12 Appendix – Broker API.....	29
12.1 Authentication	29
12.2 Signing API.....	29
12.3 Signing API Models.....	33
12.4 Signer forwarder API	35
13 Appendix – Error API	40
14 Appendix – Broker SAML Assertion.....	41
15 Appendix – SAP API Utilities	43
15.1 Code example – formatting saml (saml) assertion	43

Changelog

Date	Version	Change description	Initials
09-07-2020	0.1	Draft	
06-11-2020	0.2	Internal review and update	
12-11-2020	1.0	Version updated as part of release	
11-12-2020	1.0.1	Updated based on review from Digst	
04-01-2020	1.0.2	Support for multiple IdPs	
09-03-2021	1.0.3	Added nemlogin-broker-mock to SignSDK	
11-03-2021	1.0.4	Updated OIOSAML Attribute name format. Updated based on review from Digst	JKJER
22-04-2021	1.0.4a	Reformatting	SPEDE
13-02-2024	1.0.5	ÆA 084, ÆA087 NemID references removed	
28-11-2025	1.0.6	Add use of OCES certificates	
27-02-2026	1.0.7	Update reg. signing certificate types and SAML	ATISH

1.1 References

Term	Reference
NL-SP-IMPL	Signature SP Implementation Guideline, NemLog-in NL3 Signature SP Implementation Guideline.pdf
NL-SIG-PROFILE	AdES Signature Profile, NemLog-in. NL3 AdES Signature Format.pdf
SIGNSDK	The Java- or .Net-based NemLog-In SignSDK library used for first stage of the signing process.
OIOSAML	OIOSAML Web SSO Profile 3.0.3 is a SAML implementation profile governed by the Danish Agency for Digitisation. The specification is available here: https://digst.dk/media/1gxag1pn/oiosaml-web-ss0-profile-303.pdf
eIDAS	Regulation (EU) No 2024/1183 of the European Parliament and of the Council of 11 April 2024 amending Regulation (EU) No 910/2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC. Available here: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32024R1183
PDF	ISO 32000-1, Document management – Portable document format – Part 1: PDF 1.7. Available here: https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/PDF32000_2008.pdf
PAdES	ETSI EN 319 142-1: AdES digital signatures; Part 1: Building blocks and PAdES baseline signatures, ETSI ESI. Available here: https://www.etsi.org/deliver/etsi_en/319100_319199/31914201/01.02.01_60/en_31914201v010201p.pdf
XAdES	ETSI EN 319 132-1: XAdES digital signatures; Part 1: Building blocks and XAdES baseline signature, ETSI ESI. Available here: https://www.etsi.org/deliver/etsi_en/319100_319199/31913201/01.03.01_60/en_31913201v010301p.pdf
NSIS	National Standard for Identity Assurance Levels (NSIS), Version 2.1, Digitaliseringsstyrelsen, 2024. Available here:

	https://digst.dk/media/s0spdxxi/national-standard-for-identiteters-sikringsniveauer-nsis.pdf
--	---

2 The Purpose and Target Audience of the Document

This document is part of the NemLog-in Broker Package. The purpose of this document is to provide the technical documentation required to develop a signing client that integrates with the NemLog-in Signing backend.

This document is aimed at developers and architects.

2.1 Document Conventions

Code examples and XML snippets are written using a fixed width font. References are marked in square-brackets, e.g. [OIOSAML].

2.2 Abbreviations

Abbreviation	Description
SP	Service Provider. The entity delivering the documents to be signed. It is important to distinguish between NemLog-in SPs and the Broker SPs. In this document SP is used as a term for the Brokers SPs.
Signer	End user identity performing the actual document signing. This can be a person or an employee signing on behalf of an organization
SD Document Format	Signer's document input format (HTML, XML, Text or PDF)
SD	Signer's document. The original document that is input to the signing process. SD is in a valid Document Format.
Signature Format	The format (XAdES, PAdES) of the DTBS payload to be signed.
Signature	Signatures are produced for an individual. F.ex. a person or an employee
Seal	A Seal is a signature produced for a legal identity, f.ex. a business or organization.
DTBS	Data To Be Signed. The transformed document in a valid Signature Format that is to be signed
SDO	Signed Data Object. The signed document XAdES or PAdES
JWS	JSON Web Signature https://tools.ietf.org/html/rfc7515
PAdES	PDF Advanced Electronic Signatures
XAdES	XML Advanced Electronic Signature
ASN.1	Abstract Syntax Notation 1
AdES	Advanced Electronic Signature
CMS	Cryptographic Message Syntax. For PDF a CMS may contain a AdES.
OCSP	Online Certificate Status Provider
QSCD	A Qualified Secure Signature Generation, part of the NemLog-In Signing backend.
LoIP	Level of Identity Proofing. https://www.etsi.org/deliver/etsi_ts/119400_119499/119461/02.01.01_60/ts_119461v020101p.pdf

3 Motivation

The NemLog-in signature client is aimed for SP's to quickly integrate signature facilities using NemLog-in as an identity provider.

For Brokers who use other identity schemes, the NemLog-in infrastructure supports them building their own signature client using the NemLog-in backend systems for certificate issuing and signature generation.

Now, as the signature flow used in NemLog-in, ensures that the document to be signed remains within the Brokers environment, the integration is quite technical as the Broker must implement part of the Advanced formatting into the final signature object. Brokers pursuing this road should consult the references describing the signature formats.

4 Introduction

This document serves as the technical documentation for brokers on how to implement a Signing Client using the NemLog-in Broker API. It supplements the document [NL-SP-IMPL] aimed for Service Providers using the NemLog-in signature client by providing details specific for using the Broker API. Anyone who wishes to become a Broker must be registered with and be approved by NemLog-In.

Part of the requirements for Brokers to implement a Signing Client is to use the SignSDK [SIGNSDK] to prepare a `Signing Payload` consisting of the Data To Be Signed (DTBS) and JWS-sealed signature parameters.

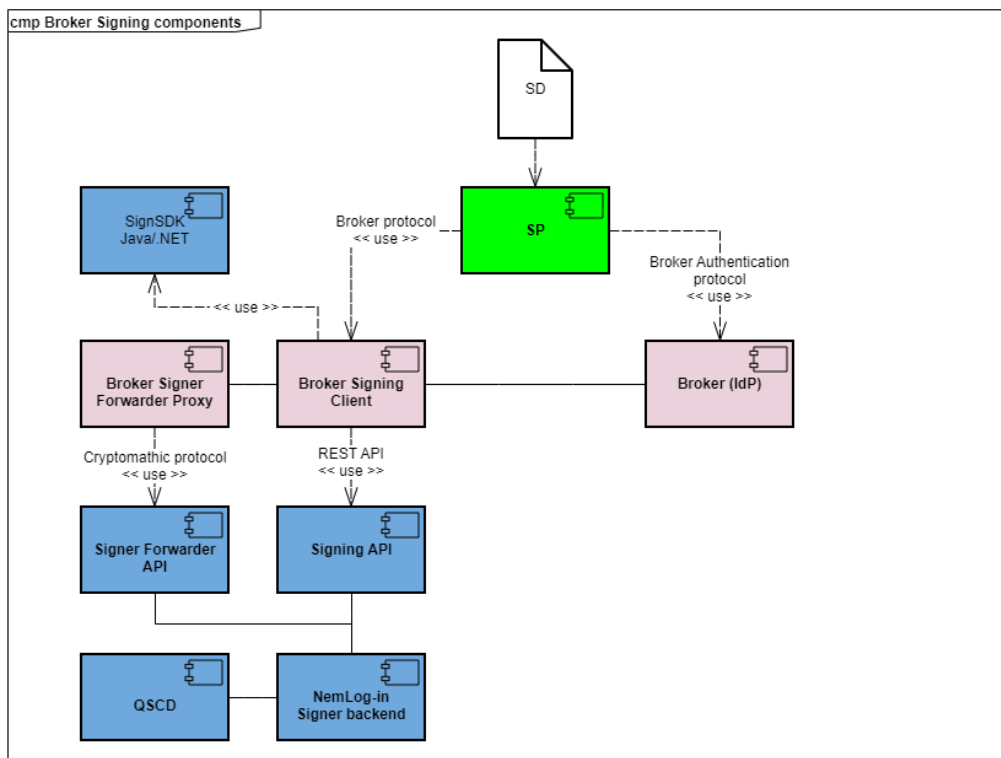
The SignSDK is documented in detail in chapter 4 of the Service Provider implementation guidelines [NL-SP-IMPL] for NemLog-in SPs.

The NemLog-in Broker API supports the creation of PAdES-B-LTA [PAdES] and XAdES-B-LTA [XAdES] signatures and seals based on the input formats HTML, Text, XML and PDF. The signature formats are described in [NL-SP-IMPL] chapter 3.

In this document, *signature* will be used at the general term for both signatures and seals, unless it is important for the implementation details.

A Broker must implement the `Broker Signing Client` and `Broker IdP` components shown below. Depending on the Broker's choice of technology they can be implemented as one component. The SPs integrate with the `Broker Signing Client`. The details of the integration between the Broker and their SPs are up to the Broker.

A Broker must be registered to NemLog-in and have a valid Broker's entity ID and VOCES certificate. Once the registration and issuing are done, NemLog-in will use the entity ID and certificate to authorize the Broker and verify the SAML Assertions.



The Broker signing process involves the following actors:

- Broker Signing Client. The Signing Client implemented by the Broker capable of creating PAdES and XAdES signatures.
- Broker (IdP). The Broker (IdP) must be capable of authenticating end users and to create valid OIOSAML Assertions. See '9 SAML Integration'.
- Broker Signer Forwarder Proxy. The Broker must implement a Signer Forwarder Proxy that enables the requests to Signer Forwarder API from the Broker Signing Client to be proxied through the Brokers backend.
- Signer. The end entity performing the signing of the Signers Document (SD).
- SignSDK: A Java- or .Net-based NemLog-In SignSDK library used for first stage of signing. The library is documented in detail in [NL-SP-IMPL] chapter 4. The SignSDK creates the Signing Payload required to initiate the signing process using the NemLog-in backend APIs.
- Signing API. Part of the NemLog-in backend and responsible for creating the AdES signatures.
- Signer forwarder API. Part of the NemLog-in backend and responsible for providing the signature value using a short-term qualified certificate issued to the Signer.
- NemLog-in backend. The NemLog-in backend components required to create the signature.

5 Broker Integration – overview

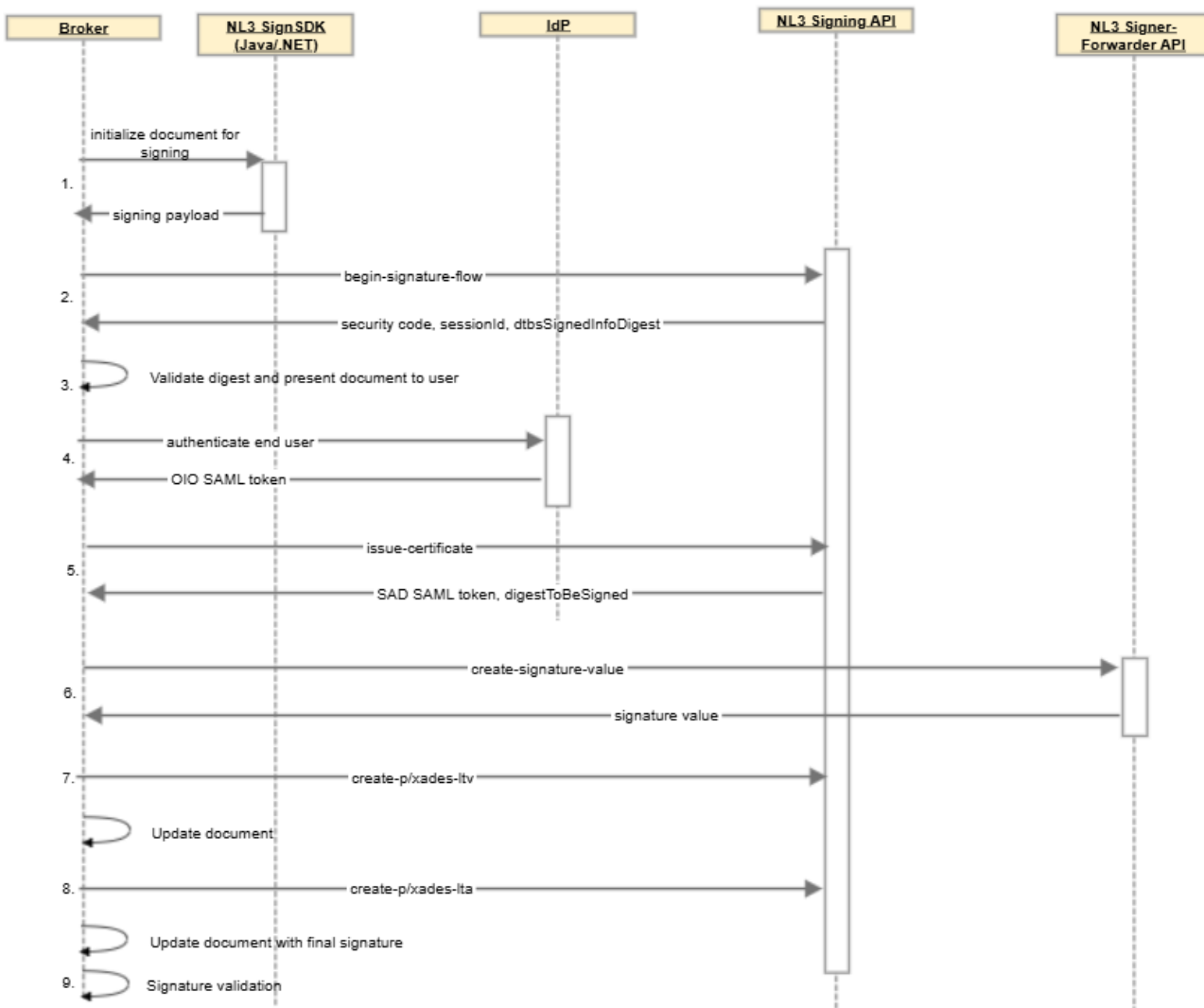
Signing a document involves a series of steps. The following is a high-level description of the required steps. See '6 Broker Integration – ' for a detailed description of each step.

The supported types of the Signer’s Documents (SD) to be signed – hereafter denoted as SD Document Format – are either HTML, Text, XML or PDF. The resulting output of the signing process can be either PAdES or XAdES. All combinations are supported by the SignSDK.

Brokers are obligated to validate that any SD only contains content permitted by the whitelists defined in [NL-SP-IMPL] appendix C & D.

5.1 Broker signing flow in detail

The illustration shows in detail the relevant parts of the signing solution and steps to be taken by the Broker to produce a signature.



- Step 1: Prepare Signing Payload. Before any document can be signed, the Signing Payload consisting of JWS-sealed Signature Parameters and Data To Be Signed (DTBS) must be initialized using the SignSDK. Please refer [NL-SP-IMPL] for details.

- **Step 2: Create Signing Session.**
Before presenting the document for the end user, a signing session must be initiated by sending the JWS-sealed Signature Parameters to the backend using the Signing API. If the backend can validate the Signature Parameters a Signing Session is created and a sessionId and security code is returned to the Broker.
- **Step 3: Present document and security code to end user.**
When presenting the document for the end user to read and sign, the security code should be visible for the end user.
- **Step 4: Authenticate end user and create OIOSAML Assertion.**
When the end user has agreed to sign the document the end user must be authenticated and an OIOSAML Assertion created. The Assertion must be unencrypted but signed.
- **Step 5: Issue certificate.**
Based on the OIOSAML Assertion, a short-term signing certificate (either Qualified or OCES Advanced certificate) is issued using the Signing API and a SAD SAML Assertion is returned. See more reg. signing certificate types - [8.5 Signing Certificate Types]

The issued certificate and corresponding private key are kept securely at the NemLog-in backend and after the signing is done, the private key is deleted. This ensures that a given certificate can only be used to sign a document once.

- **Step 6: Calculate signature value.**
Based on the response values returned when issuing the signing certificate, the raw signature value is then calculated by the QSCD using the signer-forwarder API.
- **Step 7: Create PAdES/XAdES LTV signature.**
The raw signature value is then used to create the LTV signature. When creating the LTV signature, the Signing API will perform revocation checks on all certificates used during the signing process.

For PAdES signatures, the signature AdES object is returned alongside a list of certificates used in the signing process and revocation information. The placeholder signature AdES object created by the SignSDK must then be replaced with the signature AdES object in the PDF document. Certificate and revocation dictionaries must also be added to the PDF document.

For XAdES signatures a base64 encoded string is returned which must be used when preparing the LTA signature.

- **Step 8: Create PAdES/XAdES LTA signature.**
In order to create the final LTA signature, the broker must calculate the document digest to be used when creating the archive timestamp.

For PAdES signatures, this requires a placeholder timestamp dictionary to be added to the PDF and then the digest value to be calculated over the byte ranges covering the document, LTV signature dictionary,

certificate and revocation dictionaries. The digest is then sent to the Signing API and the archive time stamp is created. The placeholder timestamp dictionary in the PDF document must then be replaced by the timestamp returned by the Signing API.

For XAdES signatures a digest must be calculated over the value of the SignText element in the XML, postfixed with the base64-decoded value returned by the Signing API in the previous step. The XMLDSig signature element created by the SignSDK must then be replaced by the signature element returned by the Signing API.

- Step 9: Validate Signature

When the document has been signed, the Broker must verify that the document has been signed by the expected end user.

6 Broker Example Code

SignSDKJava contains an example web application, *nemlogin-broker-mock*, which illustrates how Brokers may use the SignSDK to generate a *Signing Payload* and how to subsequently call the back end NemLog-In Signing API involved in the generating a signed document.

The screenshot shows the 'Broker Signing Mock' application interface. The main content area is titled 'Signing of petersen2017120113.pdf as PAdES'. On the left, a document preview is shown with the title 'Når kirsebærtærerne blomstrer' and a 'Forfatterbiografi' section. On the right, a console log displays the following information:

```
Validating signing payload:
signatureParameters: eyJ4bW90I01siTU1JR1BQ0NCU2FnQkdJQkFnSUVYT24xVURBT
dtbs: JVBERi0xLjQ3fbk/NBkMSAwIG91ago8PAovVt1wZ5AvQ2F0YhbxvZmovVmwY21w

Called Signing API -> begin-signature-flow. Result:
sessionId: meek4k411AFCl-WUEdrKOjIKPKs
securityCode: BENYWE
dtbsSignedInfoDigest: Z10ok49kAijmy8b64u0aknjVI+MDlEgo3mNDVDwlb7H=
dtbsSignedInfoDigestAlgorithm: 2.16.840.1.101.3.4.2.1

Parsing signature parameters:
{"dtbsDigestAlgorithm":"SHA-256","signatureFormat":"PAdES","entityID":

Broker TODO:
  Validate flowType
  Validate DTBS digest algorithm
  Validate stated DTBS digest against computed digest
  Validate stated Signed Info digest against computed digest

Initialized viewer

Broker TODO:
  Implement Broker SAML IdP sign-in

Awaiting simulated Broker IdP sign-in

Simulated Broker IdP sign-in. Result:
samlAssertion: PD94bWgdMvyc2lvbj0iMS4wIiBlbmV2GluZz0iVVRGLTg1Pz48C2F

Called Signing API -> issue-certificate. Result:
digestToBeSigned: 37789fc7bf1317bfa7500f0371a57749b791dc211af0fc74b465
sad: PD94bWgdMvyc2lvbj0iMS4wIiBlbmV2GluZz0iDlRMLTg1Pz48czp8C3NlcnRpl

Called Cryptomathic Signer Forwarder API. Result:
signatureValue: MEYCIQDZKQwIAmhuU1oqP+SM0IU6c2X808lTm6Ibcj65j1xmQThAl

Called Signing API -> create-pades-ltv. Result:
cmsSignedData: MIIjzgyJKoZlIhvcNAQCCoIIjvzCCI75CAQEXDZANBg1ghkgBZQMEAg
certificates: 6 certificates
```

The Broker Signing Mock applications handles all Signer's Document formats (PDF, XML, HTML, TEXT) and Signature Formats (PAdES and XAdES) and performs all the back-end API calls needed to sign a document. However, the code is not for production usage, and the task of converting the DTBS (Data To Be Signed) generated by SignSDK into the final signed document, using data obtained by calling the NemLog-In Signing API, is left to the Broker, in accordance with the specification of the following chapters.

7 Broker Integration – details

This section contains a more in-depth explanation of the steps required to create an AdES LTA signature. The process has been divided into three parts:

1. Common Steps. With details of the first six steps, which are identical for both PAdES and XAdES signatures
2. PAdES Signature. The final steps required to create PAdES Signatures.
3. XAdES Signature. The final steps required to create XAdES Signatures.

7.1 Common Steps

7.1.1 Step 1: Prepare Signing Payload

Before any document can be signed, the Signing Payload consisting of Signature Parameters and Data To Be Signed (DTBS) must be initialized using the SignSDK. Please refer to “16 Signing Formats and Data Models” and [NL-SP-IMPL] chapter 3 & 4 for details.

The JWS-sealed Signature Parameters contain the initial information required for the NemLog-in Signing backend to ensure that the signing request originates from a trusted broker and the integrity of the DTBS is intact. The Signature Parameters must contain the entity ID provisioned for the Broker using the NemLog-In Administration Component, and the parameters must be signed using the Brokers VOCES certificate and registered in NemLog-in.

The DTBS created by the SignSDK contains a placeholder signature element that must be replaced as part of the signature flow. Please note that the DTBS is never sent to the NemLog-in backend.

For PAdES signatures a placeholder Signature Dictionary is added to the PDF. This must be updated by the Broker with the real AdES signature during the signing process. Example placeholder Signature Dictionary generated by SignSDK:

```
<<
/Type /Sig
/Filter /Adobe.PPKLite
/SubFilter /ETSI.CAdES.detached
/Name (NemLog-In Signing SDK)
/Reference [24 0 R]
/M (D:20200710105856+02'00')
/Contents <000000000000.....00000000000000>
/ByteRange [0 7602 40372 512]
>>
```

For XAdES the SignedDocument element contains a preliminary XMLDSig element that will be extended by the NemLog-in backend during the signing process. The example XMLDSig signature element below is reproduced in a non-canonicalized format for increased legibility:

```
<ds:Signature Id="id-caaa8398-b576-4515-81bf-59030f476bb4">
  <ds:SignedInfo>
    <ds:CanonicalizationMethod
      Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
    <ds:SignatureMethod
      Algorithm="http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256" />
    <ds:Reference Id="r-id-caaa8398-b576-4515-81bf-59030f476bb4-1"
      URI="#id-b09c1f7a-74d8-49ed-bc91-b41bf0824673"
      Type="http://dk.gov.certifikat/nemlogin#SignText">
      <ds:Transforms>
        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
      </ds:Transforms>
      <ds:DigestMethod
        Algorithm="http://www.w3.org/2001/04/xmenc#sha256" />
      <ds:DigestValue>
        T+7K+dtYvTNjbgjDBcUpRswbDXBATYW4d/tEXlkuRZQ=
      </ds:DigestValue>
    </ds:Reference>
  </ds:SignedInfo>
</ds:Signature>
```

7.1.2 Step 2: Create Signing Session

A Signing Session is created by calling the Signing API REST operation `begin-signature-flow`. Input to this operation is the JWS-sealed Signature Parameters created by the SignSDK.

The NemLog-in backend will validate Signature Parameters using the VOCES certificate and verify that the VOCES certificate is associated to a Broker.

When the NemLog-in backend has validated the Signature Parameters, a signing session is created and a `sessionId` is returned to the Broker. The `sessionId` must be provided in a http header in all subsequent calls to the NemLog-in backend.

Furthermore, a security code is returned and is provided as a mean to let end users identify their signing session during the signing process. The security code shall be visible for the Signer when presented with the document to be signed, on the login screen etc.

A mean to ensure the document to be signed matches the document pre-signed by the SignSDK is also returned by `begin-signature-flow` in the form of the Base64-encoded `dtbsSignedInfoDigest` and `dtbsSignedInfoDigestAlgorithm`. The Broker client should recalculate the DTBS digest using the stated algorithm and compare to the returned digest:

- For PAdES, the DTBS digest should be computed based on the ByteRange included in the Signature Dictionary added by SignSDK.
- For XAdES, the DTBS digest should be computed based on the extracted and canonicalized SignText element.

If the stated and computed digests do not match, then the document shown to the end user does not match the document pre-signed by the SignSDK and the signing flow must be aborted.

7.1.3 Step 3: Present document and security code to end user

When presenting the document to sign to the end user, the security code shall also be shown. This is to help the Signer identify their signing session during all steps.

It is the Broker's responsibility to ensure that what is presented to the Signer is also the content signed by the Signer. If e.g. a XLST transformation is used when presenting an XML document, it's the brokers responsibility to ensure all data in the XML document is visible after transformation.

7.1.4 Step 4: Authenticate end user and create OIOSAML Assertion

When the Signer is ready to sign the document, the user must be authenticated using the Broker IdP. The SAML Assertion created by the IdP must adhere to the OIOSAML requirements specified in [OIOSAML] with the additional requirements stated in section '8 SAML Integration'. Furthermore, the Assertion must be unencrypted but be signed using the Brokers VOCES provisioned to the SigningComponent.

For more details of the authentication and SAML Assertion requirements see '9 SAML Integration'

Note that if the SAML Assertion states the Persistence Level as Global, explicit consent must be given by the signer, to comply with privacy and eIDAS regulations.

When presenting the Signer with the login, the security code identifying the signing session shall be visible to the Signer.

7.1.5 Step 5: Issue Certificate

Based on the information in the OIOSAML Assertion, a short-term certificate either Qualified or OCES Advanced will be issued to the Signer. The certificate is created by calling the Signing API REST operation `issue-certificate` with the OIOSAML Assertion as input. The issued certificate is kept securely on the NemLog-in Signing backend, and the private key is deleted after the signature session has been finalized, or when the signing session timeout is reached. See more reg. signing certificate types - [8.5 Signing Certificate Types]

When issuing the signing certificate the NemLog-in Signing backend will also perform SAD exchange with the QSCD and return the `digestToBeSigned` and the SAD SAML assertion required to authenticate with the QSCD in the next step.

7.1.6 Step 6: Calculate signature value

To calculate the signature value – please read about the SAP protocol enabling in 12.4.2 SAP Protocol to be able to calculate the signature value.

After the signature value has been calculated the remaining signing process for PAdES and XAdES will be described separately in the following sections.

7.2 PAdES Signature

To create a PAdES signature, the DTBS generated by the SignSDK must be extended with data returned from the NemLog-in backend Signing API and Signer forwarder API.

The examples do not contain all the implementation details required to create a valid PDF document containing an LTA signature. The Broker is expected to have in-depth knowledge about the PDF specification and to know the additional elements (xref, trailer, root catalog, etc.) required also to be added and/or updated when adding the mentioned objects to a PDF document.

7.2.1 Step 7: Create PAdES LTV signature

When calling the Signing API REST operation `create-pades-ltv` with the signature value calculated in the previous step, the final CMS Signed Data containing a signature adhering to the LTV-profile is returned alongside the certificates and OCSP revocation information.

The hex-encoded CMS Signed Data shall replace the Content of the placeholder `Sig` dictionary created by the SignSDK when initializing the document to be signed. The replacement `Sig` Content must be padded to the exact same length as the placeholder `Sig` content (green below).

```
<<
/Type /Sig
/Filter /Adobe.PPKLite
/SubFilter /ETSI.CAdES.detached
/Name (NemLog-In Signing SDK)
/Reference [24 0 R]
/M (D:20200710105856+02'00')
/Contents <308223eb06092.....00000000000000>
/ByteRange [0 7602 40372 512]
>>
```

Furthermore, the PDF must be incrementally updated to include the other PAdES LTV artifacts. This entails adding an updated root catalogue dictionary containing the certificates and revocation information must also be added to the PDF by the broker.

```
<<
/Type /Catalog
/Version /1.7
/Pages 2 0 R
/Metadata 3 0 R
/MarkInfo 4 0 R
/Lang (EN-US)
/ViewerPreferences 5 0 R
/OutputIntents [6 0 R]
/Perms 21 0 R
/AcroForm <<
/Fields [22 0 R]
/SigFlags 3
>>
/DSS 26 0 R
```

```

>>
endobj
26 0 obj
<<
/OCSPs [27 0 R 28 0 R 29 0 R 30 0 R]
/Certs [31 0 R 32 0 R 33 0 R 34 0 R 35 0 R 36 0 R]
>>
Endobj

```

The referenced objects containing the actual certificates and OSCP revocation information has been omitted here. Please refer to [PAdES] and [PDF] for details.

7.2.2 Step 8: Create PAdES LTA signature

In order to prepare the final PAdES LTA signature, the PDF must again be incrementally extended with a PAdES LTA document timestamp. The updated root catalog should add additional dictionaries, including a placeholder `DocTimeStamp` dictionary. Example:

```

<<
/Type /Catalog
/Version /1.7
/Pages 2 0 R
/Metadata 3 0 R
/MarkInfo 4 0 R
/Lang (EN-US)
/ViewerPreferences 5 0 R
/OutputIntents [6 0 R]
/Perms 21 0 R
/AcroForm <<
/Fields [22 0 R 37 0 R]
/SigFlags 3
>>
/DSS 26 0 R
>>
endobj
37 0 obj
<<
/FT /Sig
/Type /Annot
/Subtype /Widget
/F 132
/T (Signature2)
/V 38 0 R

```

```

/P 8 0 R
/Rect [0.0 0.0 0.0 0.0]
>>
endobj
38 0 obj
<<
/Type /DocTimeStamp
/Filter /Adobe.PPKLite
/SubFilter /ETSI.RFC3161
/Contents <00000000000000000000000000000000.....>
/ByteRange [0 61685 94455 314]
>>
endobj

```

The `ByteRange` must cover the entire document except the content of the `DocTimeStamp Contents` element (in green). The `Contents` value will be replaced by the real `TimeStamp AdES` at a later stage.

Afterwards, a digest of the `ByteRange`-defined part of the document must be calculated using the digest algorithm specified in [NL-SIG-PROFILE]

The digest is input to the Signing API REST operation `create-pades-lta` which returns the time stamp AdES needed to complete the PAdES LTA signature.

The broker must replace the `Contents` of the placeholder `DocTimeStamp` with the HEX-encoded time stamp returned by the NemLog-in Signing backend, padded to the exact length of the placeholder value.

The end result is a PAdES LTA level signature signed using a short term certificate issued to the end entity, specified in the SAML Assertion.

7.2.3 Step 9: Validate Signature

When the document has been signed, the Broker must verify that the document has been signed by the expected end user.

7.3 XAdES Signature

When creating a XAdES signature the NemLog-in Signing backend will extend the preliminary XMLDSig signature created by the SignSDK and return the complete signature to the broker in the final step.

7.3.1 Step 7: Create XAdES LTV signature

When calling the Signing API REST operation `create-xades-ltv` with the signature value calculated in the previous step, the NemLog-in Signing backend will update the XMLDSig signature received in the `Signature Parameters` and return a Base64-encoded string containing the canonicalized value of most of the xml elements required to create the `<SignatureTimeStamp>` in the next step.

7.3.2 Step 8: Create XAdES LTA signature

Since, for privacy reasons, the `<SignText>` element containing the document to be signed is never sent to the NemLog-in Signing backend, the broker must calculate the digest required to create the `<SignatureTimeStamp>`.

The response value returned when calling `create-xades-ltv` in the previous step must be Base64-decoded and postfixed to the value of the `<SignText>` element so the result consists of:

```
<SignText> +
<SignedProperties> +
<SignedInfo> +
<SignatureValue> +
<keyInfo> +
<SignatureTimeStamp> +
<CertificateValues> +
<RevocationValues>
```

The string must then be canonicalized and a digest calculated. The digest is used as input to the Signing API REST operation `create-xades-lta`. The digest algorithm and canonicalization method is defined in [NL-SIG-PROFILE].

The NemLog-in Signing backend creates the document time stamp and makes the final XMLDSig element. The Base64-encoded XMLDSig element returned by the `create-xades-lta` operation must be decoded. Then the preliminary XMLDSig Signature element added to the XAdES document by SignSDK must be replaced with the decoded XMLDSig element.

This finalizes the XAdES LTA signing process.

7.3.3 Step 9: Validate Signature

When the document has been signed, the Broker must verify that the document has been signed by the expected end user.

8 Signing Formats and Data Models

The Signing Formats and Data Models are described in detail in [NL-SP-IMPL]. The referenced document is focused on the requirements related to NemLog-in Service Providers. In this section the differences in the requirements related to Brokers will be described as a supplement to [NL-SP-IMPL]. The Service Providers using the Broker cannot use the SignSDK as the Signing Payload must be signed by the Broker.

8.1 Signature Parameters

The Signature Parameters is used to control the document signing, and must be provided, by the Broker, as input to the SignSDK when producing a signing payload.

Signature Parameters listed below are mandatory.

Parameter	Value	Description
version	1	Signature Parameter version Defaults to 1
flowType	"ServiceProvider"/"Broker"	Can only be "Broker" in this context.
entityID	URI	A Broker-specific entity ID provisioned using the NemLog-In Administration Component.

Parameter	Value	Description
documentFormat	"TEXT"/"HTML"/"PDF"/"XML"	Signer's Document format (TEXT, HTML, XML, PDF)
signatureFormat	"XAdES" / "PAdES"	The type of the signed document "XAdES-B-LTA" or "PAdES-B-LTA"

Signature Parameters listed below are not supported for Brokers and will result in a validation error if specified.

Parameter	Value	Description
referenceText	N/A	
minAge	N/A	
preferredLanguage	N/A	
signerSubjectNameID	N/A	
ssnPersistenceLevel	N/A	Brokers must specify this in the SAML Assertion
anonymizeSigner	N/A	Brokers must specify this in the SAML Assertion
acceptedCertificatePolicies	N/A	Brokers must specify this in the SAML Assertion

8.2 Signing Payload

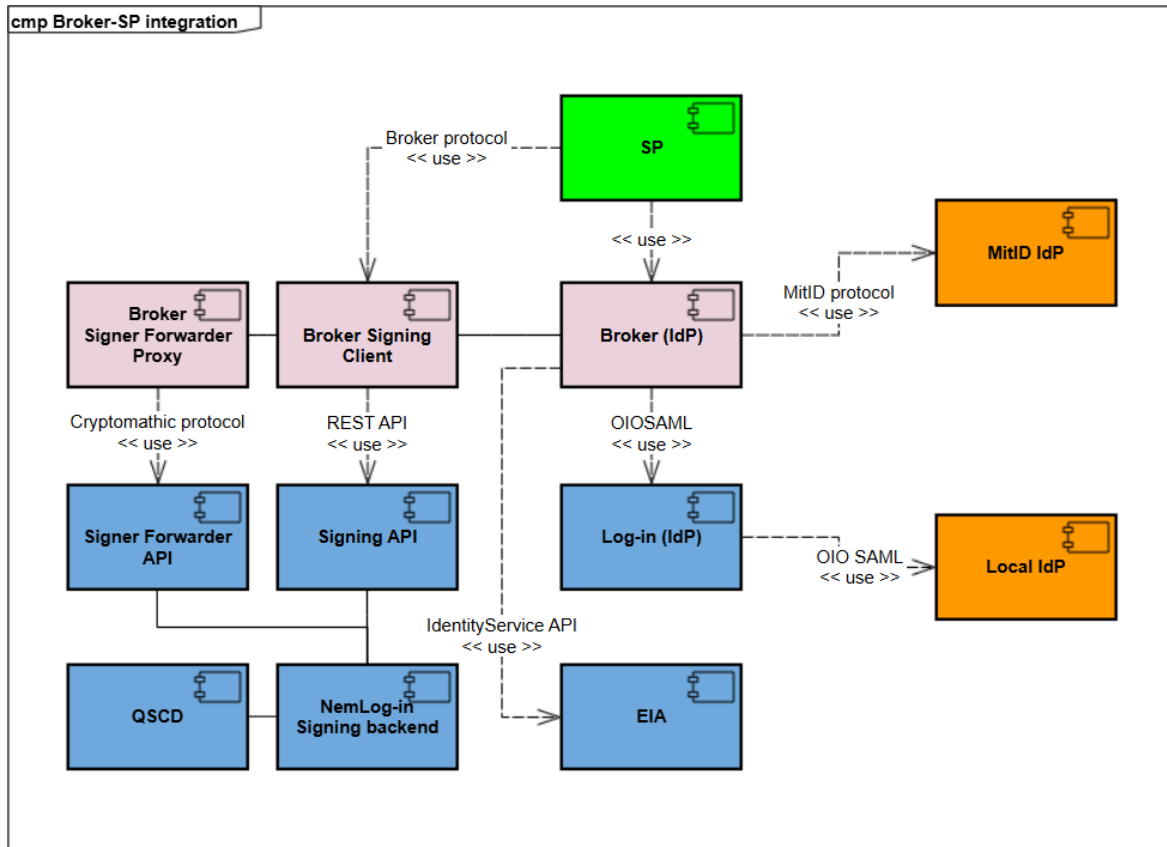
For Brokers the Signing Payload must be Signed by a VOCES keypair provisioned by the Broker using the Administration Client.

8.3 Signing Client Error

This section in [NL-SP-IMPL] is not directly applicable for Brokers as they implement their own Signing Client, however, the `SigningClientError` and `DetailedSigningClientError` models also defines the data structure used for Signing API errors.

9 SAML Integration

Upon authenticating a Signer, the Broker IdP must produce a valid OIOSAML 3 Assertion. The short-term certificate issued when calling `issue-certificate` is issued based on the information received in the SAML Assertion.



The details of how a Broker authenticate their SPs is up to the Broker to design and implement.

The actions of the Broker (IdP) depend on the type of credentials:

- Local credentials.
When using local credentials, the Broker integrates with the Log-in component to perform the authentication. The resulting OIOSAML Assertion must be modified if needed to comply to the requirements in this chapter.
- Personal MitID credentials.
When using MitID credentials, the Broker integrates directly with MitID.
- Employee MitID credentials.
When creating signature or seals using MitID employee credentials, the Broker must query EIA (the IdentityService) accordingly.

According to this integration model, it is the Broker who issues the SAML Assertion used when communicating with the Signing API, independent of the end user's choice of credentials (MitID or local). The SAML Assertion sent to the NemLog-in backend must always be signed using the Brokers VOCES certificate provisioned to the Signing Component.

9.1 SAML Assertion requirements

The NemLog-in backend requires a number of attributes to be present in the SAML Assertion created by the Broker. Some of these attributes are optional in the OIOSAML 3 profile but are mandatory in a signing context.

The short-term certificate used for signing is issued based on the on the information contained in the SAML Assertion.

This section describes the requirements for the SAML Assertions issued by the Broker. The SAML Assertion must always be signed using the Brokers VOCES certificate for the NemLog-in backend to verify it is issued by a trusted party. Furthermore, the SAML Assertion must not be encrypted.

The Broker must ensure that all mandatory attributes in the OIOSAML 3 profile, not mentioned below, are also present in the SAML Assertion.

9.2 Subject

Subject denotes the identity of the Signer (for person and employee signatures) or approver (for seals)

9.2.1 NameID

The Broker must use the value of their own persistent `SubjectNameID` for employees (when signing) or approving employee (when creating a seal). When the Broker creates a seal, the Broker must verify employee authorization using the EIA Identity Service.

E.g.

```
<saml:NameID SPNameQualifier="https://saml.some-broker.dk"
  Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent">
  https://data.gov.dk/model/core/eid/professional/uuid/123e4567-e89b-12d3-a456-426655440000
</saml:NameID>
```

For Personal signatures the Broker must use the `CprUuid` as persistent id. This is the same value as `CertificateHolderIdentifier` (without the 'person' prefix) e.g:

```
<saml:NameID SPNameQualifier="https://saml.some-broker.dk"
  Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent">
  https://data.gov.dk/model/core/eid/person/uuid/123e4567-e89b-12d3-a456-426655440000
</saml:NameID>
```

9.2.2 SubjectConfirmation

SubjectConfirmation is mandatory and is therefore always specified cf. OIOSAML 3.0. The Broker must specify NotOnOrAfter, which is validated by the NemLog-in backend. InResponseTo must always be specified, and the value must be the SessionId created when calling Signing API begin-sign-flow e.g.

```
<saml2:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
  <saml2:SubjectConfirmationData
    InResponseTo="xYyuqxDDkJ2yW3oq9qWmTG1uM"
    NotOnOrAfter="2020-09-02T11:57:05.889Z"
    Recipient=" https://signing.nemlog-in.dk/signing/auth/saml/assertion-consumer"/>
</saml2:SubjectConfirmation>
```

9.3 Conditions

The Conditions element must contain a AudienceRestriction element.

9.3.1 AudienceRestriction

The AudienceRestriction element is specified as the Signing Component's EntityID:

Environment	Value
Customer Test Integration	https://saml.signer-cti-nemlog-in.dk
Production	https://saml.signer-prod-nemlog-in.dk

E.g.:

```
<saml:AudienceRestriction>
  <saml:Audience>https://saml.signer-cti.nemlog-in.dk</saml:Audience>
</saml:AudienceRestriction>
```

9.4 AttributeStatement

The SAML assertion must contain an AttributeStatement element containing a series of attributes used when the short-term signing certificate is issued by the NemLog-in Signing backend.

The attributes will be part of the certificate's subjectDN for the five different Signer-types:

4. Person
5. Employee
6. Anonymized person
7. Anonymized employee
8. Seal

Unless explicitly stated, the attribute is mandatory.

9.4.1 SpecVersion

The specVersion attribute value must be "OIO-SAML-3.0".

```
<saml:Attribute Name="https://data.gov.dk/model/core/specVersion"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>OIO-SAML-3.0</saml:AttributeValue>
</saml:Attribute>
```

9.4.2 Loa

The loa attribute value must be either 'Substantial' or 'High' as pr. [NSIS] requirement.

```
<saml:Attribute Name="https://data.gov.dk/concept/core/nsis/loa"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Substantial</saml:AttributeValue>
</saml:Attribute>
```

9.4.3 LoIP Extended

The LoIP Extended attribute value must either be "True" or "False". In combination with Loa used to determine what certificate type to be issued during signature creation flow.

```
<saml2:Attribute Name="https://nemlog-in.dk/internal/loipextended"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml2:AttributeValue>>true</saml2:AttributeValue>
</saml2:Attribute>
```

9.4.4 CertificatePolicyQualifier

The certificatePolicyQualifier attribute value must be one of:

- Person (for Personal Signature)
- Employee (for Employee Signature)
- Organization (for Seal)

```
<saml:Attribute Name="https://data.gov.dk/model/core/signing/certificatePolicyQualifier"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Person</saml:AttributeValue>
</saml:Attribute>
```

Based on the policy, the signature type will be determined and a short-term certificate either Qualified or OCES will be issued based on the type. See more reg. signing certificate types - [8.5 Signing Certificate Types]

9.4.5 AnonymizedSigner

The anonymizedSigner attribute value must be either 'true' or 'false'

```
<saml:Attribute Name="https://data.gov.dk/model/core/signing/anonymizedSigner"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>>false</saml:AttributeValue>
</saml:Attribute>
```

If true, the certificate will be anonymized. For anonymized certificates, the SubjectDN element's `cn` and `pseudonym` will have the value 'Pseudonym'. Can only be true in conjunction with Person or Employee certificate policy.

Setting anonymizedSigner to true with the certificatePolicyIdentifier set to "Organization" will result in an error being thrown by the Signing API.

9.4.6 CertificateHolderIdentifier

Global identifier for certificate holder. The certificate issued is associated to this identifier in CA to allow revocations to be performed given this identifier. If CertificatePolicy is "Person" the HolderIdentifier is equal to CPR-UUID for signer, if CertificatePolicy is "Employee" or "Organization", the HolderIdentifier is equal to the global identifier for the corporate identity provided by NemLog-in Erhverv (EIA).

Values have the syntax <lowercase(certificatePolicy)><UUID>, see example.

```
<saml:Attribute Name="https://data.gov.dk/model/core/signing/certificateHolderIdentifier"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>person:22222222-ae53-443f-b7c6-01b94fbf7a51</saml:AttributeValue>
</saml:Attribute>
```

9.4.7 CertificateSSNPersistenceLevel

The certificateSSNPersistenceLevel attribute value must be either of 'Session' or 'Global'.

```
<saml:Attribute Name="https://data.gov.dk/model/core/signing/certificateSSNPersistenceLevel"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Session</saml:AttributeValue>
</saml:Attribute>
```

The attribute is used to determine the persistence level for the UUID specified in the subjectSerialNumber of the short-term certificate.

A value of Session instructs the NemLog-in Signing backend to create a session-specific UUID to be used as subjectSerialNumber. Access to Lookup Service is required in order to subsequently identify the entity associated with the given session UUID.

A value of Global will use the global certificateHolderIdentifier as subjectSerialNumber. When using global UUIDs, explicit consent from the end entity is required to be GDPR compliant.

9.4.8 Certificate subjectDN attributes

The subjectDN of the short-term certificate is created based on the SAML Assertion attribute values specified in the table below.

Whether the attribute must be present in the SAML Assertion depends on the Signer-type.

Mandatory SAML Assertion attributes for each signer-type

Attribute (friendly name)	Person	Employee	Anonymized person	Anonymized employee	Seal (Organization)
FirstName	x	x			
LastName	x	x			
CommonName			"Pseudonym"	"Pseudonym"	x*
OrgName		x		x	x
CVR		x		x	x**

* Certificate CommonName is taken from SAML Assertion

** CVR prefixes according to certificate profile

FirstName

```
<saml:Attribute Name="https://data.gov.dk/model/core/eid/firstName"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Julie</saml:AttributeValue>
</saml:Attribute>
```

LastName

```
<saml:Attribute Name="https://data.gov.dk/model/core/eid/lastName"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Sørensen</saml2:AttributeValue>
</saml:Attribute>
```

CommonName

For Person and Employee, `cn` of the certificate Subject DN will be the concatenated value of `firstName` and `lastName`.

If the `anonymizedSigner` attribute is true, however, then `commonName` must contain the value "Pseudonym" and the `firstName` and `lastName` attributes must be unspecified.

For seals, `commonName` must be the appointed name of the organization identity.

```
<saml:Attribute Name="https://data.gov.dk/model/core/signing/certificateCommonName"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Julie Sørensen</saml2:AttributeValue>
</saml:Attribute>
```

OrgName

```
<saml:Attribute Name=" https://data.gov.dk/model/core/eid/professional/orgName"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Bennys Bagelbageri</saml2:AttributeValue>
</saml:Attribute>
```

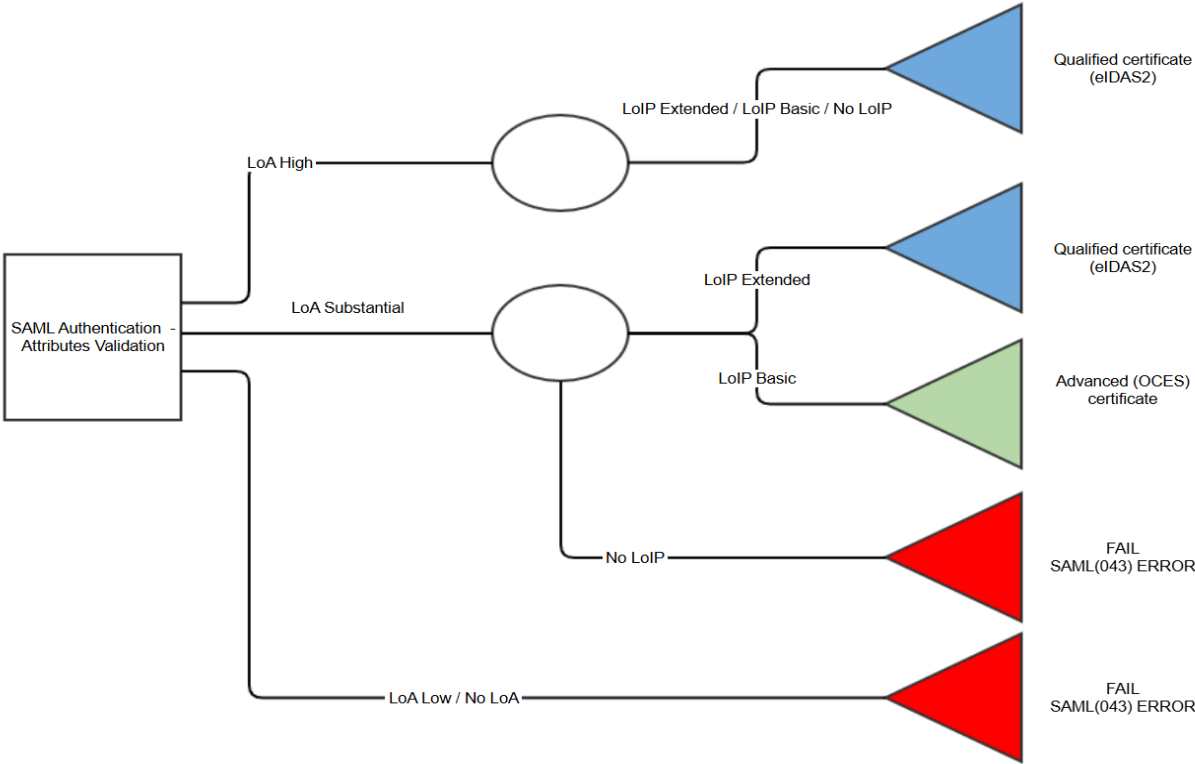
CVR

```
<saml:Attribute Name=" https://data.gov.dk/model/core/eid/professional/cvr"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>20301823</saml2:AttributeValue>
</saml:Attribute>
```

9.5 Signing Certificate Types

Signing certificate types are defined by profile types - Personal, Employee and Organizational and by certification types - Qualified (eIDAS2 compliant) or OCES Advanced short-term certificates which are issued during a signature creation process to sign DTBS.

The decision what certification type to issue during a document signing flow is based on combination of Level of Assurance (LoA) and Level of Identity Proofing (LoIP) SAML attributes during end-user's authentication via Identity Provider (IdP) and validation authentication information during a document signing process. The decision matrix is illustrated below:



9.5.1 Broker implications

It is up to the broker to inform an end-user about different certificate types to be issued depending on LoA and LoIP SAML parameters during authentication at Broker IdP. Brokers require to accept both Qualified and Advanced OCES signed documents.

10 Logging

It is the responsibility of the Broker to store and provide the information needed to prove the correctness of a signature or an authentication at a later point of time, e.g. due to a dispute or fraud case.

The signed document from each signing operation must be stored by the Broker or SP. The user's certificate (which includes a UUID to identify the signer), the time stamp etc. are included in signed documents. Brokers are encouraged to consider storing other relevant information like time stamp, client IP number, user agent, correlation id and similar information subject to regulative requirements.

All requests to the Signing and Signer forwarder REST APIs will be logged by the NemLog-in Signing backend. The Broker must add a "CorrelationIdManager.CorrelationId" HTTP header containing a caller-selected value with a UUID v4 format when calling the Signing API, and all requests will be logged with that value. This makes it possible to trace all requests for a given signing session.

Note that NemLog-in never handles the actual documents to be signed and therefore does not store the signed documents.

Brokers are encouraged to use the same correlationId internally and when interacting with the NemLog-in Signing backend as this will aid the troubleshooting process in case of errors.

11 Response codes and error handling

When interacting with the Signing and Signer forwarder REST APIs the following HTTP response codes may be returned.

Code	Description
200	Success
400	Bad request. When an error occurs, the Broker must cancel the signing, remedy the error and create a new signing session to sign the document
401	Unauthorized. Will be returned if the NemLog-in Signing backend fails to recognize the Brokers VOCES certificate used to seal the Signature Parameters. No signing session can be created while this error occurs

When an error occurs a JSON object describing the error will be returned. See [NL-SP-IMPL] Appendix B for a full list of error codes.

```
{
  "httpStatusCode": 0,
  "timestamp": "string",
  "message": "string",
  "details": [
    {
      "errorCode": "string",
      "errorMessage": "string"
    }
  ]
}
```

If the error can be rectified by the Broker an error message with details of the error is returned.

12 Security Guidelines

The security advice and best-practice in [NL-SP-IMPL] Chapter 6 Security Guidelines, is also recommended for Brokers.

13 Appendix – Broker API

Signing Frontend Service defines a two REST APIs:

- Signing API
- Signer Forwarder API

Required parameters and model fields are tagged with *****.

13.1 Authentication

Access to both REST APIs require TLS.

After a signature flow has been initialized the SigningAPI can only be called using a valid Session ID. The SessionId is created when the Broker invokes `begin-signature-flow` with valid signature parameters signed by the Broker's VOCES certificate provisioned to the Signing Component. All subsequent calls must contain the SessionId as a HTTP header.

If the Brokers VOCES certificate is unknown to the Nemlog-in backend a session will not be created.

13.1.1 Common Request Parameters

All REST API endpoints accepts the following correlation ID header:

Name	In	Type	Description
CorrelationIdManager .CorrelationId*	header	string	The Correlation ID is defined by the caller and will be in all log records generated while signing. The value must have a UUID v4 format, but there is no requirement that it is unique.

13.2 Signing API

13.2.1 begin-signature-flow

Initializes the signature flow by

REQUEST

POST <https://<uri>/signing/begin-signature-flow>

Request Parameters

Name	In	Type	Description
Accept*	header	string	"application/json"
Content-type*	header	string	"application/json"
signatureParameters*	body	BeginSignatureFlowRequest	Signature parameters containing the document to be signed and other parameters required for signing. The JWS signed string is generated by SignSDK

RESPONSE

Code	Content-Type Type	Description
200 OK	"application/json"	

Code	Content-Type Type	Description
	BeginSignatureFlowResponse	
400 Bad Request		Signature Parameters validation failure
401 Unauthorized		Not authorized to perform signing.

13.2.2 issue-certificate

REQUEST

POST <https://<uri>/signing/issue-certificate>

Request Parameters

Name	In	Type	Description
Accept*	header	string	"application/json"
Content-type*	header	string	"application/json"
X-DIGST-Signing-SessionId*	header	string	SessionId returned in BeginSignatureFlowResponse
	body	IssueCertificateRequest	

RESPONSE

Code	Content-Type Type	Description
200 OK	"application/json" IssueCertificateResponse	
400 Bad Request		
401 Unauthorized		Not authorized to perform signing. See 12.1 Authentication

13.2.3 create-pades-ltv

REQUEST

POST <https://<uri>/signing/create-pades-ltv>

Request Parameters

Name	In	Type	Description
Accept*	header	string	"application/json"
Content-type*	header	string	"application/json"
X-DIGST-Signing-SessionId*	header	string	SessionId returned in BeginSignatureFlowResponse
	body	CreatePadesLtvRequest	

RESPONSE

Code	Content-Type Type	Description
200 OK	"application/json" CreatePadesLtvResponse	
400 Bad Request		
401 Unauthorized		Not authorized to perform signing. See 12.1 Authentication

13.2.4 create-pades-lta

REQUEST

POST <https://<uri>/signing/create-pades-lta>

Request Parameters

Name	In	Type	Description
Accept*	header	string	"application/json"
Content-type*	header	string	"application/json"
X-DIGST-Signing-SessionId*	header	string	SessionId returned in BeginSignatureFlowResponse
	body	CreatPadesLtaResponse	

RESPONSE

Code	Content-Type Type	Description
200 OK	"application/json" CreatePadesLtaResponse	
400 Bad Request		
401 Unauthorized		Not authorized to perform signing. See 12.1 Authentication

13.2.5 create-xades-ltv

REQUEST

POST <https://<uri>/signing/create-xades-ltv>

Request Parameters

Name	In	Type	Description
Accept*	header	string	"application/json"
Content-type*	header	string	"application/json"
X-DIGST-Signing-SessionId*	header	string	SessionId returned in BeginSignatureFlowResponse
	body	CreateXadesLtvRequest	

RESPONSE

Code	Content-Type Type	Description
200 OK	"application/json" CreateXadesLtvReponse	
400 Bad Request		
401 Unauthorized		Not authorized to perform signing. See 12.1 Authentication

13.2.6 create-xades-lta

REQUEST

POST <https://<uri>/signing/create-xades-lta>

Request Parameters

Name	In	Type	Description
Accept*	header	string	"application/json"
Content-type*	header	string	"application/json"
X-DIGST-Signing-SessionId*	header	string	SessionId returned in BeginSignatureFlowResponse
	body	CreateXadesLtaRequest	

RESPONSE

Code	Content-Type Type	Description
200 OK	"application/json" CreateXadesLtaResponse	
400 Bad Request		
401 Unauthorized		Not authorized to perform signing. See 12.1 Authentication

13.2.7 session-creation-key.js

Calling Signing frontend service provides the SCK for broker usage.

REQUEST

POST <https://<signer-frontend-service>/signing/session-creation-key.js>

RESPONSE

Code	Content-Type	Description
200 OK	application/javascript;charset=UTF-8	

Response example

```
(function(sck) {
  sck.Details = {
    "KeyID": "0000000000000023",
```

```

    "Modulus":
"00E7E7E0599D73685DB70C75EBB58A941409987F7C7083FE602239BF80F33964F39C1541AB411A59
3F5D409FC6ACDAFE4E576BC32532205914DC9A78E160086493B31FAF003082D1601E03F592F20B9BA
88C45F1F3B0BDC6C0233386FFD8681518AAA3F5015678C36D29C0C2E94C469F19AB59732514973DF0
2DB4B4005B810B32586EF6D05DF5BA20331D139D3CD047035D2E34F12E82CED00952F450A89337855
4958FC92321F45D450D67B8C13ACF91EB3D77597FCFF9B278A072043B0B5DAE8488DCE007994013C4
7D7ABEE82247D93288FF0ED6F431557E445E30C5474838F2B98EEC59D54C7D4FABD31627C5000B2EE
58E21F4EE648AF799E9EA673CEF7149D8289BF77319CAC851293E193289FCD57E6F501025A55B4CC5
B09420E7F096BD90AB66A26B02FF1D2D52E0C438823FBA13A9A46F8D95DECF66AADB77EDC4364CA4B
2A272A0C7CFA886016DB6EAFB7C7047D8DDF4AAE7F496AB01D318E1F702144303725FB01272139C5C
C3420903A98391624FD46ACEDC6413D15C687F2633",
    "PublicExponent": "010001",
    "KeySize": 384
};
}(Cryptomathic.namespace("Cryptomathic.SCK")));

```

13.3 Signing API Models

13.3.1 BeginSignatureFlowRequest

Name	Type	Description
signatureParameters	string	JWS-encoded and signed Signature Parameters

13.3.2 BeginSignatureFlowResponse

Name	Type	Description
securityCode*	string	Security code to use for end-user identification
sessionId*	string	A session ID for the signing work flow
dtbsSignedInfoDigest*	string	Base64-encoded digest extracted from the dtbsSignedInfo field of the signatureParameters This together with the dtbsSignedInfoDigestAlgorithm can be used to verify that the document presented to the end user corresponds to the document initialized by the SignSDK by recalculating the digest value of the document to be signed
dtbsSignedInfoDigestAlgorithm*	string	Digest algorithm extracted from the dtbsSignedInfo field of the signatureParameters

13.3.3 IssueCertificateRequest

Name	Type	Description
samlAssertion*	string	Base64 encoded OIOSAML assertion

13.3.4 IssueCertificateResponse

Name	Type	Description
digestToBeSigned*	string	Hex encoded document digest to be signed by the QSCD
sad*	string	Base64 encoded QSCD SAD SAML assertion to be used for authentication with the QSCD

13.3.5 CreatePadesLtvRequest

Name	Type	Description
signatureValue*	string	Base64 encoded signature value from the QSCD

13.3.6 CreatePadesLtvResponse

Name	Type	Description
cmsSignedData*	string	Base64 encoded AdES CMS containing the signature. The returned AdES must replace the placeholder AdES added to the PDF document by the SignSDK
certificates*	array	List of Base64 encoded x509 certificates to be included in PDF document when creating archive timestamp
ocspResponses*	array	List of Base64 encoded OCSP ASN1 responses to be included in PDF document when creating archive timestamp

13.3.7 CreatePadesLtaRequest

Name	Type	Description
timeStampDigestValue*	string	Base64 encoded byte array containing the digest value to time stamp

13.3.8 CreatePadesLtaResponse

Name	Type	Description
timeStamp*	string	Base64 encoded AdES CMS containing the time stamp. The returned AdES must replace the placeholder timestamp AdES added to the PDF document when calculating the time stamp digest value.

13.3.9 CreateXadesLtvRequest

Name	Type	Description
signatureValue*	string	Base64 encoded signature value from the QSCD

13.3.10 CreateXadesLtvResponse

Name	Type	Description
archiveTimeStampInitialContent*	string	Base64 encoded string with the canonicalised concatenated value of SignedProperties + SignedInfo + SignatureValue + KeyInfo + SignatureTimeStamp + CertificateValues + RevocationValues. This is to be postfixed to the SignText element. The digest value of this is to be used as input for CreateXadesLtaRequest.

13.3.11 CreateXadesLtaRequest

Name	Type	Description
------	------	-------------

timeStampDigestValue*	String	Base64 encoded digest value of canonicalised SignText + SignedProperties + SignedInfo + SignatureValue + KeyInfo + SignatureTimeStamp + CertificateValues + RevocationValues to time stamp
------------------------------	--------	--

13.3.12 CreateXadesLtaResponse

Name	Type	Description
signature*	String	Base64 encoded xml signature element at CAES level LTA. The signature element is to replace the Signarure element in the SignedDocument created by the SignSDK

13.4 Signer forwarder API

Signer Forwarder API is used together with QSCD SDK to calculate signature value (signer-forwarder)

13.4.1 signer-forwarder

REST endpoint uses to expose QSCD SAP protocol to Cryptomathic JavaScript User SDK. All calls from the broker client is routed from Broker Signing Client -> Broker Signer Backend Proxy on Broker backend -> Signer Forwarder API -> Signing backend -> QSCD to handle browser CORS issues.

The signer-forwarder does not support CORS¹ so the Broker Signing Client cannot use the Signer Forwarder API directly from the client but must handle the signer-forward logic from a Signer Forwarder Proxy on the Broker backend.

REQUEST

POST <https://<uri>/signer-forwarder>

Request Parameters

Name	In	Type	Description
Accept*	header	string	"*/"
Content-type*	header	string	"text/plain;charset=UTF-8"
sessionId*	parameter	string	Contains the signing session ID from begin-sign-flow.
correlationId	parameter	string	CorrelationIdManager.CorrelationId used to track QSCD logging
	body		Encrypted payload to forwarder

RESPONSE

Code	Content-Type	Description
200 OK	"*/"	
Errorcode varies		Cryptomathic forwarder errorcodes varies, please refer to Cryptomathic documentation.

¹ CORS : Cross-origin resource sharing see <https://fetch.spec.whatwg.org/>

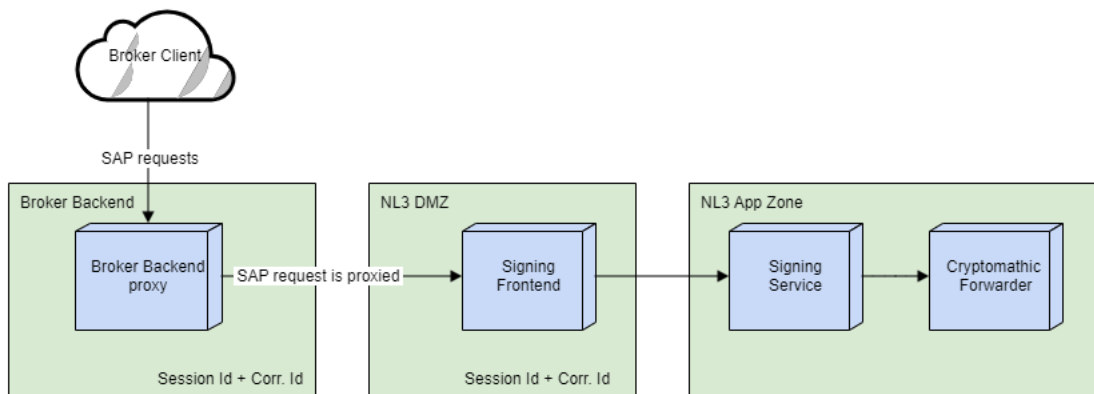
13.4.2 SAP Protocol

When a Broker is implementing a signing client, they must use a qualified SAP protocol provider certified with ETSI EN 419-241-2 to be considered a valid qualified signer. Part of the protocol requirement is that the signing is controlled from the client side all the way to the QSCD server.

In NL3 this has been implemented with Cryptomathic Signer as an SAP API protocol provider that all clients must utilize to be a qualified signing client.

The underlying protocol used is called SAP API and is used in the Broker Client by including Cryptomathic Signer SDK and then calling a Broker Backend service that proxies call to the signer-forwarder referenced in 12.4.1 signer-forwarder.

Example of a Broker Client.



The reason the Broker Backend is proxying calls is to solve CORS issues between browser domains (assuming the Broker Client is running in another domain than signer-frontend service).

Implement the following steps to be able to enable the SAP Protocol.

Step	Description
1. Signer sdk in client	Add signersdk.min.js to client, this is used to call the signer-forwarder. Can be taken from Cryptomathic User SDK.
2. Add SCK	SCK is short for Session Creation Key and is used to establish a protocol to the backend. This can be included from signing frontend service: <code>https://<frontend-service>/signing/session-creation-key.js</code> See [12.2.7 session-creation-key.js]
3. Call Issue Certificate	Implement "Issue Certificate" [6.1.5 Step 5: Issue Certificate] in broker client and extract the field "sad" from response.
4. Initialize QSCD API	Now the client should establish the SAP protocol by using Cryptomathic User SDK. To avoid CORS issues, the <Forwarderurl> used in the example below must be proxied to the signing frontend via the Brokers own signer forwarder proxy. The format of the request being proxied to the signing frontend is: <code>https://<frontend-service>/signer-forwarder?sessionId=<session-id>&correlationId=<correlation-id></code> . The <session-id> is the URI-encoded session ID retrieved in the call to <code>begin-signature-flow</code> .

Step	Description
	<p>The optional <correlation-id> parameter is the URI-encoded correlation ID used for logging purposes – please refer to chapter 9.</p> <pre data-bbox="421 338 1348 521">let sdk = new Cryptomathic.SignerUserSDK.SDK(<Fowarderurl>, 10000); sdk.initialize();</pre>
5 Create Session	<p>Create session with callback objects</p> <pre data-bbox="421 624 1348 745">sdk.createSession(resolve, reject, samlAssertion);</pre> <p>Important: It is very important that the SAD from [6.1.5 Step 5: Issue Certificate] is formatted before calling createSession!</p> <p>Please read [15.1 Code example – formatting saml (sad) assertion]</p>
Implement resolve	<p>Example of calling userSDK sign</p> <p>Convert digestToBeSigned from issue-certificate via convertStringToNumbers</p> <pre data-bbox="421 1104 1348 1167">let messageHashToBeSigned = convertStringToNumbers(digestToBeSigned);</pre> <p>Convert method:</p> <pre data-bbox="421 1234 1348 1473">convertStringToNumbers(values: string): number[] { let bytes = []; for (let c = 0; c < values.length; c += 2) { bytes.push(parseInt(values.substr(c, 2), 16)); } return bytes; }</pre> <p>The call userSDK sign method</p> <pre data-bbox="421 1585 1348 1671">sdk.sign(keyEntry, callForwarderRequest.hashToSign, resolveSign, rejectSign)</pre> <p>Implement resolveSign, this will receive the signing response from QSCD</p> <pre data-bbox="421 1776 1348 1839">let resolveSign = function (signatureBytes) { };</pre>

Client Javascript pseudo example:

```

<html>
<head>
  <script type="text/javascript" src="https://<signing-
client>/assets/signersdk.min.js"></script>
  <script type="text/javascript" src="https://<frontend-
service>/signing/session-creation-key.js"></script>

  <script type="application/javascript">
    function demoSigning() {
      // call previous steps (begin-signature-flow, issue-
certificate)
      // to create session and signing certificate [1]
      // from /begin-signature-flow response sessionId [2]
      const sessionId = "dZOpNilun-MDJ4ZwvIFbrSU3WG4";

      // log correlation from Broker Client (must be generated by
      // broker client) [3]
      const correlationId = "a1fa4b69-62cf-4772-ab99-e846f90ad74f";

      // extract IssueCertificateResponse.digestToBeSigned from
      // /issue-certificate (encoded to number[] array) [4]
      let messageHashToBeSigned = [0x01, 0x02, 0x01];

      let forwarderURL = "https://<broker-backend-
proxy>/?sessionId="
        + sessionId + "&correlationId=" + correlationId;

      let sdk = new Cryptomathic.SignerUserSDK.SDK(forwarderURL,
10000);
      try {
        sdk.initialize();
        console.log("Successfully initialized SDK.");

        let rejectLogOff = function (errorType, message) {
          console.debug("logoff failed msg:" + message + "");
        };

        let cleanup = function () {
          sdk.logoff(resolveLogOff, rejectLogOff);
          sdk.free();
        };

        let resolveLogOff = function () {
          console.debug("logoff successful");
          cleanup();
        };

        let resolveSign = function (signatureBytes) {
          console.info("signed '" + signatureBytes + "'");
        };

        let rejectSign = function (errorType, message) {
          console.error("signing rejected msg:" +
            + message + "");
          cleanup();
        };
      }
    }
  </script>

```

```

        let resolve = function (tokenList, policyList) {
            console.log("Successfully created session.");
            let aPolicy = policyList[0];

            // keyEntry from a PolicyEntry received in
            // createSession resolve callback.
            let keyEntry = aPolicy.KEY_LIST[0];
            let hashToSign = messageHashToBeSigned;

            console.log("executing sign");
            sdk.sign(keyEntry, hashToSign, resolveSign,
rejectSign)
        }

        // sad is taken from /issue-certificate response sad
        // field [5]
        let sad = "PD94bWwgdmVyc2lvcj0iMS4wIiBlbmNvZGluZRMlTg..."
        let samlAssertion = encodeSAMLAssertion(sad);

        sdk.createSession(resolve, reject, samlAssertion);
    } catch (error) {
        console.error("error " + error);
        sdk.free();
    }
}

window.onload = demoSigning;
</script>
</head>
</html>

```

The Broker Client must Implement [1],[2],[3],[4],[5] in the client to make this pseudo example into a real running client.

14 Appendix – Error API

Reference [NL-SP-IMPL] documentation

15 Appendix – Broker SAML Assertion

Example of a SAML Assertion consumed by the NemLog-in backend. The information received in the SAML Assertion is used when issuing the short-term signing certificate used to sign the document.

```
<?xml version="1.0" encoding="UTF-8"?>
<saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
  ID="_ce0e53df9f39c35c695972c054cc5463"
  IssueInstant="2020-09-02T11:52:05.889Z"
  Version="2.0">
  <saml:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity">
    https://broker-entity-id
  </saml:Issuer>
  <saml:Subject>
    <saml:NameID Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent"
      SPNameQualifier="https://broker-entity-id">
      ff7db82f-d5e6-4377-ac62-ef22a4acb134
    </saml:NameID>
    <saml:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
      <saml:SubjectConfirmationData InResponseTo="xYyuqxDDkJ2yW3oq9qWmTGU1uM"
        NotOnOrAfter="2020-09-02T11:57:05.889Z"
        Recipient="https://127.0.0.1:8181/signing/auth/saml/assertion-consumer"/>
    </saml:SubjectConfirmation>
  </saml:Subject>
  <saml:Conditions NotBefore="2020-09-02T11:52:05.889Z" NotOnOrAfter="2020-09-02T12:52:05.889Z">
    <saml:AudienceRestriction>
      <saml:Audience>https://signer.nemlog-in.dk</saml:Audience>
    </saml:AudienceRestriction>
  </saml:Conditions>
  <saml:AuthnStatement AuthnInstant="2020-09-02T11:52:06.208Z"
    SessionIndex="_d264e96a232fc98249a0a9875809f50b">
    <saml:AuthnContext>
      <saml:AuthnContextClassRef>
        urn:oasis:names:tc:SAML:2.0:ac:classes>PasswordProtectedTransport
      </saml:AuthnContextClassRef>
    </saml:AuthnContext>
  </saml:AuthnStatement>
  <saml:AttributeStatement>
    <saml:Attribute FriendlyName="SpecVer"
      Name="https://data.gov.dk/model/core/specVersion"
      NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
      <saml:AttributeValue>OIO-SAML-3.0</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute FriendlyName="LoA"
      Name="https://data.gov.dk/concept/core/nsis/loa"
      NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
      <saml:AttributeValue>Substantial</saml:AttributeValue>
    </saml:Attribute>
    <saml2:Attribute Name="https://nemlog-in.dk/internal/loipextended"
      NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
      <saml2:AttributeValue>true</saml2:AttributeValue>
    </saml2:Attribute>
    <saml:Attribute FriendlyName="CertificatePolicyQualifier"
      Name="https://data.gov.dk/model/core/signing/certificatePolicyQualifier"
      NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
      <saml:AttributeValue>Employee</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute FriendlyName="CertificateSSNPersistenceLevel"
      Name="https://data.gov.dk/model/core/signing/certificateSSNPersistenceLevel"
      NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
      <saml:AttributeValue>Session</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute FriendlyName="CertificateHolderIdentifier"
      Name="https://data.gov.dk/model/core/signing/certificateHolderIdentifier"
      NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
      <saml:AttributeValue>employee:ff7db82f-d5e6-4377-ac62-ef22a4acb134</saml:AttributeValue>
    </saml:Attribute>
    <saml:Attribute FriendlyName="CertificateCommonName"
      Name="https://data.gov.dk/model/core/signing/certificateCommonName"
      NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
      <saml:AttributeValue>Jane Fondax</saml:AttributeValue>
    </saml:Attribute>
  </saml:AttributeStatement>
</saml:Assertion>
```

```

<saml:Attribute FriendlyName="FirstName"
  Name="https://data.gov.dk/model/core/eid/firstName"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Jane</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute FriendlyName="LastName"
  Name="https://data.gov.dk/model/core/eid/lastName"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Fonda</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute FriendlyName="OrganizationName"
  Name="https://data.gov.dk/model/core/eid/professional/orgName"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>Three Mile Island</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute FriendlyName="CVR"
  Name="https://data.gov.dk/model/core/eid/professional/cvr"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>11111111</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute FriendlyName="Age"
  Name="https://data.gov.dk/model/core/eid/age"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>93</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute FriendlyName="AnonymizedSigner"
  Name="https://data.gov.dk/model/core/signing/anonymizedSigner"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>>false</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute FriendlyName="ReferenceText"
  Name="https://data.gov.dk/model/core/signing/referenceText"
  NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
  <saml:AttributeValue>U2lnbiBkb2N1bWVudCBmcm9tIFVua25vd24gU1A=
</saml:Attribute>
</saml:AttributeStatement>
</saml:Assertion>

```

16 Appendix – SAP API Utilities

16.1 Code example – formatting saml (sad) assertion

When a SAML Assertion (SAD) is received from issue-certificate, it must be base64 decoded and then encoded as UTF8. The following JavaScript provides guidance in how to achieve this. The formatting shall be conducted prior establishing a session with `createSession`.

```
let samlAssertion = encodeSAMLAssertion(sad); ← IMPORTANT!
sdk.createSession(resolve, reject, samlAssertion);
```

EncodeSAMLAssertion function

```
function encodeSAMLAssertion(assertion) {
  function toUTF8Array(str) {
    let utf8 = [];
    for (let i = 0; i < str.length; i++) {
      let charcode = str.charCodeAt(i);
      if (charcode < 0x80) utf8.push(charcode);
      else if (charcode < 0x800) {
        utf8.push(0xc0 | (charcode >> 6),
          0x80 | (charcode & 0x3f));
      } else if (charcode < 0xd800 || charcode >= 0xe000) {
        utf8.push(0xe0 | (charcode >> 12),
          0x80 | ((charcode >> 6) & 0x3f),
          0x80 | (charcode & 0x3f));
      }
      // surrogate pair
      else {
        i++;
        // UTF-16 encodes 0x10000-0x10FFFF by
        // subtracting 0x10000 and splitting the
        // 20 bits of 0x0-0xFFFFF into two halves
        charcode = 0x10000 + (((charcode & 0x3fff) << 10)
          | (str.charCodeAt(i) & 0x3ff));
        utf8.push(0xf0 | (charcode >> 18),
          0x80 | ((charcode >> 12) & 0x3f),
          0x80 | ((charcode >> 6) & 0x3f),
          0x80 | (charcode & 0x3f));
      }
    }
    return utf8;
  }

  function base64decode(data) {
    return window.atob(data)
  }

  return toUTF8Array(base64decode(assertion));
}
```