Digital Signature Service

version : 4.6.0 - 2016-02-22

Table of Contents

Introduction

Purpose of the document

This document describes some examples of how to develop in Java using the DSS framework. The aim is to show to the developers, in a progressive manner, the different uses of the framework. It will familiarise them with the code step by step.

Scope of the document

This document provides examples of code which allow easy handling of digital signatures. The examples are consistent with the Release 4.6.0 of DSS framework which can be downloaded via https://joinup.ec.europa.eu/software/sd-dss/release/all

Three main features can be distinguished within the framework :

- The digital signature;
- The extension of a digital signature and;
- The validation of a digital signature.

On a more detailed manner the following concepts and features are addressed in this document:

- Formats of the signed documents: XML, PDF, DOC, TXT, ZIP…;
- Packaging structures: enveloping, enveloped and detached;
- Forms of digital signatures: XAdES, CAdES, PAdES and ASiC;
- Profiles associated to each form of the digital signature;
- Trust management;
- Revocation data handling (OCSP and CRL sources);
- Certificate chain building;
- Signature validation and validation policy;
- Validation of the signing certificate.

This is not an exhaustive list of all the possibilities offered by the framework and the proposed examples cover only the most useful features. However, to discover every detail of the operational principles of the framework, the JavaDoc is available within the source code.

Please note that the DSS framework is still under maintenance and new features will be released in the future.

Abbreviations and Acronyms

Code	Description
AdES	Advanced Electronic Signature
API	Application Programming Interface
ASiC	Associated Signature Containers
BB	Building Block (CEF)
CA	Certificate authority
CAdES	CMS Advanced Electronic Signatures
CD	Commission Decision
CEF	Connecting Europe Facility
CMS	Cryptographic Message Syntax
CRL	Certificate Revocation List
CSP	Core Service Platform (CEF)
CSP	Cryptographic Service Provider
DER	Distinguished Encoding Rules
DSA	Digital Signature Algorithm - an algorithm for public-key cryptography
DSI	Digital Service Infrastructure (CEF)
DSS	Digital Signature Service
EC	European Commission
eID	Electronic Identity Card
ESI	Electronic Signatures and Infrastructures
ETSI	European Telecommunications Standards Institute
EUPL	European Union Public License
FSF	Free Software Foundation
GS	Generic Service (CEF)
GUI	Graphical User Interface
HSM	Hardware Security Modules
HTTP	Hypertext Transfer Protocol

I18N	Internationalisation
Java EE	Java Enterprise Edition
JavaDoc	JavaDoc is developed by Sun Microsystems to create API documentation in HTML format from the comments in the source code. JavaDoc is an industrial standard for documenting Java classes.
JAXB	Java Architecture for XML Binding
JCA	Java Cryptographic Architecture
JCE	Java Cryptography Extension
JDBC	Java DataBase Connectivity
LGPL	Lesser General Public License
LOTL	List of Trusted List or List of the Lists
LSP	Large Scale Pilot
MIT	Massachusetts Institute of Technology
MOCCA	Austrian Modular Open Citizen Card Architecture; implemented in Java
MS / EUMS	Member State
MS CAPI	Microsoft Cryptographic Application Programming Interface
OCF	OEBPS Container Format
OCSP	Online Certificate Status Protocol
ODF	Open Document Format
ODT	Open Document Text
OEBPS	Open eBook Publication Structure
OID	Object Identifier
OOXML	Office Open XML
OSI	Open Source Initiative
OSS	Open Source Software
PAdES	PDF Advanced Electronic Signatures
PC/SC	Personal computer/Smart Card
PDF	Portable Document Format

PDFBox	Apache PDFBox - A Java PDF Library: http://pdfbox.apache.org/
PKCS	Public Key Cryptographic Standards
PKCS#12	It defines a file format commonly used to store X.509 private key accompanying public key certificates, protected by symmetrical password
PKIX	Internet X.509 Public Key Infrastructure
RSA	Rivest Shamir Adleman - an algorithm for public-key cryptography
SCA	Signature Creation Application
SCD	Signature Creation Device
SME	Subject Matter Expert
SMO	Stakeholder Management Office (CEF)
SOAP	Simple Object Access Protocol
SSCD	Secure Signature-Creation Device
SVA	Signature Validation Application
TL	Trusted List
TLManager	Application for managing trusted lists.
TSA	Time Stamping Authority
TSL	Trust-service Status List
TSP	Time Stamp Protocol
TSP	Trusted Service Provider
TST	Time-Stamp Token
UCF	Universal Container Format
URI	Uniform Resource Identifier
WSDL	Web Services Description Language
WYSIWYS	What you see is what you sign
XAdES	XML Advanced Electronic Signatures
XML	Extensible Markup Language
ZIP	File format used for data compression and archiving

References

Ref.	Title	Reference	Version
R01	XAdES Specifications	ETSI TS 101 903	1.4.2
R02	CAdES Specifications	ETSI TS 101 733	2.2.1
R03	PAdES Specifications	ETSI TS 102 778 part 1-6	1.x.x
R04	ASiC Specifications	ETSI TS 102 918	1.1.1
R05	Document management - Portable document format - Part 1: PDF 1.7	ISO 32000-1	1
R06	XAdES Baseline profiles	ETSI TS 103 171	2.1.1
R07	CAdES Baseline profiles	ETSI TS 103 173	2.2.1
R08	PAdES Baseline profiles	ETSI TS 103 172	2.1.1
R09	ASiC Baseline profiles	ETSI TS 103 174	2.1.1
R10	Directive 1999/93/EC of the European Parliament and of the Council of 13 December 1999 on a Community framework for electronic signatures.	DIRECTIVE 1999/93/EC	
R11	Internet X.509 Public Key Infrastructure - Time-Stamp Protocol (TSP)	RFC 3161	
R12	Electronic Signatures and Infrastructures - Signature verification procedures and policies	ETSI TS 102 853	1.1.1

Useful links

- Joinup
- Source code
- Report a bug

General framework structure

DSS framework is a multi-modules project which can be builded with Maven.

dss-model

Data model used in almost every modules.

dss-token

Token definitions and implementations for MS CAPI, PKCS#11, PKCS#12.

dss-document

Common module to sign and validate document. This module doen't contain any implementation.

dss-asic

Implementation of the ASiC-S and ASiC-E signature, extension and validation.

dss-cades

Implementation of the CAdES signature, extension and validation.

dss-pades

Implementation of the PAdES signature, extension and validation.

dss-xades

Implementation of the XAdES signature, extension and validation.

dss-spi

Interfaces, util classes to manipulate ASN1, compute digests,...

dss-service

Implementations to communicate with online resources (TSP, CRL, OCSP).

dss-tsl-validation

Module which allows to load / parse / validate LOTL and TSLs.

dss-demo-applet

Applet which allows to sign a document with different formats and tokens. This applet uses SOAP webservices.

dss-demo-applet-package

Packaging module for dss-demo-applet.

dss-light-applet

Ligth applet which allows to interact with SSCD (load certificates and sign digest).

dss-light-applet-package

Packaging module for dss-light-applet.

dss-standalone-app

Standalone application which allows to sign a document with different formats and tokens (JavaFX).

dss-standalone-app-package

Packaging module for dss-standalone-app.

dss-demo-webapp

Demonstration web application which presents a part of the DSS possibilities.

dss-demo-bundle

Packaging module for dss-demo-webapp.

validation-policy

Business of the signature's validation.

dss-rest

REST webservices to sign (getDataToSign, signDocument methods) and extend a signature.

dss-rest-client

Client for the REST webservices.

dss-soap

SOAP webservices to sign (getDataToSign, signDocument methods) and extend a signature.

dss-soap-client

Client for the SOAP webservices.

dss-remote-services

Common code between dss-rest and dss-soap.

dss-webservices

Old SOAP webservices which allows to sign / extend / validate a signature.

dss-webservices-client

Client for dss-webservices.

sscd-mocca-adapter

Implementation for MOCCA token.

dss-policy-jaxb

JAXB model of the validation policy.

dss-tsl-jaxb

JAXB model of the TSL.

dss-diagnostic-jaxb

JAXB model of the diagnostic data.

dss-test

Mocks and util classes for unit tests.

dss-cookbook

Samples and documentation of DSS used to generate this documentation.

Signature's profile simplification

The different formats of the digital signature make possible to cover a wide range of real live cases of use of this technique. Thus we distinguish the following formats: XAdES, CAdES, PAdES and ASIC. To each one of them a specific standard is dedicated. The wide variety of options, settings and versions of the standards makes their interoperability very difficult. This is the main reason for which new standards commonly called "baseline profiles" were published. Their goal is to limit the number of options and variants thereby making possible a better interoperability between different actors.

In general can be said that for each format of the digital signature the number of security levels defined in the new standards has been reduced. Below is a comparative table of old and new levels for each format of the signature:

<table>
<tr><td colspan="2">XAdES</td><td colspan="2">CAdES</td><td colspan="2">PAdES</td></tr>
<tr><td>STANDARD</td><td>BASELINE</td><td>STANDARD</td><td>BASELINE</td><td>STANDARD</td><td>BASELINE</td></tr>
<tr><td>XAdES-BES</td><td rowspan="2">XAdES-B</td><td>CAdES-BES</td><td rowspan="2">CAdES-B</td><td>PAdES-BES</td><td rowspan="2">PAdES-B</td></tr>
<tr><td>XAdES-EPES</td><td>CAdES-EPES</td><td>PAdES-EPES</td></tr>
<tr><td>XAdES-T</td><td>XAdES-T</td><td>CAdES-T</td><td>CAdES-T</td><td>PAdES-T</td><td>PAdES-T</td></tr>
<tr><td>XAdES-XL</td><td>XAdES-LT</td><td>CAdES-XL</td><td>CAdES-LT</td><td>PAdES-XL</td><td>PAdES-LT</td></tr>
<tr><td>XAdES-A</td><td>XAdES-LTA</td><td>CAdES-A</td><td>CAdES-LTA</td><td>PAdES-LTV</td><td>PAdES-LTA</td></tr>
</table>

Note that the new version (v4) of the DSS framework is compatible with the baseline profiles, it is no longer possible to use the standard profiles for signing purpose. The validation of the signature still takes into account the old profiles.

The XML Signature (XAdES)

The simplest way to address the digital signature passes through the XAdES format. Indeed, it allows to

visualize the content of the signature with a simple text editor. Thus it becomes much easier to make the connection between theoretical concepts and their implementation. Before embarking on the use of the DSS framework, it is advisable to read the following documents:

- XAdES Specifications (cf. [R01])
- XAdES Baseline Profile (cf. [R06])

After reading these documents, it is clear that:

- To electronically sign a document, a signing certificate (that proves the signer's identity) and the access to its associated private key is needed.
- To electronically validate a signed document the signer's certificate containing the public key is needed. To give a more colourful example: when a digitally signed document is sent to a given person or organization in order to be validated, the certificate with the public key used to create the signature must also be provided.

XAdES Profiles

The new ETSI standard [17] defines four conformance levels to address the growing need to protect the validity of the signature in time. Henceforth to denote the level of the signature the word "level" will be used. Follows the list of levels defined in the standard:

- XAdES-BASELINE-**B**: Basic Electronic Signature The lowest and simplest version just containing the SignedInfo, SignatureValue, KeyInfo and SignedProperties. This level combines the old -BES and -EPES levels. This form extends the definition of an electronic signature to conform to the identified signature policy.
- XAdES-BASELINE-**T**: Signature timestamp A timestamp regarding the time of signing is added to protect against repudiation.
- XAdES-BASELINE-**LT**: Long Term level Certificates and revocation data are embedded to allow verification in future even if their original source is not available. This level is equivalent to the old -XL level.
- XAdES-BASELINE-**LTA**: Long Term with Archive timestamp By using periodical timestamping (e.g. each year) compromising is prevented which could be caused by weakening previous signatures during a long-time storage period. This level is equivalent to the old -A level.

Old levels: -BES, -EPES, -C, -X, -XL, -A are not supported any more when signing.

XAdES-BASELINE-B

To start, let's take a simple XML document:

xml_example.xml

```
<?xml version="1.0"?>
<test>Hello World !</test>
```

Since this is an XML document, we will use the XAdES signature and more particularly XAdES-BASELINE-B level, which is the lowest level of protection: just satisfying Directive (cf. [R10]) legal requirements for advanced signature. The normal process of signing wants to sign first with the level -B or level-T, and then later when it becomes necessary to complete the signature with superior levels. However, the framework allows signing directly with any level. The use of CAdES format for signing an XML document is also possible, but will be discussed later. When signing data, the resulting signature needs to be linked with the data to which it applies. This can be done either by creating a data set which combines the signature and the data (e.g. by enveloping the data with the signature or including a signature element in the data set) or placing the signature in a separate resource and having some external means for associating the signature with the data. So, we need to define the packaging of the signature, namely ENVELOPED, ENVELOPING or DETACHED.

- **ENVELOPED :** when the signature applies to data that surround the rest of the document;
- **ENVELOPING :** when the signed data form a sub-element of the signature itself;
- **DETACHED :** when the signature relates to the external resources separated from it.

For our example we will use ENVELOPED packaging.

To write our Java code, we still need to specify the type of KeyStore to use for signing our document, more simply, where the private key can be found. We can choose between three different connection tokens:

- PKCS#11,
- PKCS#12,
- MS CAPI

The DSS also provides the support for MOCCA framework to communicate with the Smartcard with PC/SC, but it involves the installation of the MOCCA and IAIK libraries. To use a Java KeyStore please refer to the following paragraphs: #JavaKeyStore, #JKSSignatureToken and #Signing_with_JKSSignatureToken. In the package "eu.europa.ec.markt.dss.signature.token", we can find three corresponding Java classes:

- Pkcs11SignatureToken,
- Pkcs12SignatureToken,
- MSCAPISignatureToken.

To know more about the use of the different signature tokens, please consult "Management of Signature Tokens" chapter.

In our example the class: "Pkcs12SignatureToken" will be used. A file in PKCS#12 format must be provided to the constructor of the class. It contains an X.509 private key accompanying the public key certificate and protected by symmetrical password. The certification chain can also be included in this file. It is possible to generate dummy certificates and their chains with OpenSSL. Please visit http://www.openssl.org/ for more details.

This is the complete code that allows you to sign our XML document.

SignXmlXadesB.java

```
// Preparing parameters for the XAdES signature
XAdESSignatureParameters parameters = new XAdESSignatureParameters();
// We choose the level of the signature (-B, -T, -LT, -LTA).
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_B);
// We choose the type of the signature packaging (ENVELOPED, ENVELOPING, DETACHED).
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);
// We set the digest algorithm to use with the signature algorithm. You must use the
// same parameter when you invoke the method sign on the token. The default value is
SHA256
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();

// Create XAdES service for signature
XAdESService service = new XAdESService(commonCertificateVerifier);

// Get the SignedInfo XML segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
SignatureValue signatureValue = signingToken.sign(dataToSign, parameters
.getDigestAlgorithm(), privateKey);

// We invoke the service to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

What you may notice is that to sign a document we need to:

- Create an object based on SignatureParameters class. The number of specified parameters depends on the type of signature. Generally, the number of specified parameters depends on the profile of signature. This object also defines some default parameters.
- Choose the profile, packaging, signature digest algorithm.
- Indicate the private key entry to be used.
- Instantiate the adequate signature service.
- Carry out the signature process.

The encryption algorithm is determined by the private key and therefore cannot be compelled by the setter of the signature parameters object. It will cause an inconsistency in the signature making its validation impossible. This setter can be used in a particular context where the signing process is distributed on different machines and the private key is known only to the signature value creation process. See clause "Signing process" for more information. In the case where the private key entry object is not available, it is possible to choose the signing certificate and its certificate chain as in the following example:

```
// We set the signing certificate
parameters.setSigningCertificate(certificateToken);
// We set the certificate chain
parameters.setCertificateChain(certificateChain);
```

Integrating the certificate chain in the signature simplifies the build of a prospective certificate chain during the validation process.

By default the framework uses the current date time to set the signing date, but in the case where it is necessary to indicate the different time it is possible to use the setter "setSigningDate(Date)" as in the example:

```
// We set the date of the signature.
parameters.bLevel().setSigningDate(new Date());
```

When the specific service is instantiated a certificate verifier must be set. This object is used to provide four different sources of information:

- the source of trusted certificates (based on the trusted list(s) specific to the context);
- the source of intermediate certificates used to build the certificate chain till the trust anchor. This source is only needed when these certificates are not included in the signature itself;
- the source of OCSP;
- the source of CRL.

In the current implementation this object is only used when profile -LT or -LTA are created. In the next

release it will be used to identify the trust anchor and by the same limit the number of certificates included within the KeyInfo.

Signing process

Once the parameters of the signature were identified the service object itself must be created. The service used will depend on the type of document to sign. In our case it is an XML file, so we will instantiate a XAdES service. The process of signing takes place in three stages. The first is the "getDataToSign ()" method call, passing as a parameter the document to be signed and the previously selected settings. This step returns the data which is going to be digested and encrypted. In our case it corresponds to the SignedInfo XMLDSig element.

```
// Create XAdES service for signature
XAdESService service = new XAdESService(commonCertificateVerifier);

// Get the SignedInfo XML segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);
```

The next step is a call to the function "sign()" which is invoked on the object token representing the KeyStore and not on the service. This method takes three parameters. The first is the array of bytes that must be signed. It is obtained by the previous method invocation. The second is the algorithm used to create the digest. You have the choice between SHA1, SHA256, and SHA512 (this list is not exhaustive). And the last one is the private key entry.

```
DigestAlgorithm digestAlgorithm = parameters.getDigestAlgorithm();
SignatureValue signatureValue = signingToken.sign(dataToSign, digestAlgorithm,
privateKey);
```

The last step of this process is the integration of the signature value in the signature and linking of that one to the signed document based on the selected packaging method. This is the method "signDocument()" on the service. We must pass to it three parameters: again the document to sign, the signature parameters and the value of the signature obtained in the previous step.

This separation into three steps allows use cases where different environments have their precise responsibilities: specifically the distinction between communicating with the token and executing the business logic.

When the breakdown of this process is not necessary than a simple call to only one method can be done as in the following example:

```
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

Additional attributes

For this type (XAdES-BASELINE-B) of signature it is possible to identify some additional attributes:

- SignerRole - contains claimed or certified roles assumed by the signer when creating the signature.
- SignatureProductionPlace - contains the indication of the purported place where the signer claims to have produced the signature.
- CommitmentTypeIndication - identifies the commitment undertaken by the signer in signing (a) signed data object(s) in the context of the selected signature policy.
- AllDataObjectsTimeStamp - each time-stamp token within this property covers the full set of references defined in the Signature's SignedInfo element, excluding references of type "SignedProperties".
- IndividualDataObjectsTimeStamp - each time-stamp token within this property covers selected signed data objects.
- CounterSignature - contains signature(s) produced on the signature.

The DSS framework allows to setup the following signed properties: SignerRole, SignatureProductionPlace, CommitmentTypeIndication, AllDataObjectsTimestamp, IndividualDataObjectsTimeStamp and CounterSignature.

SignXmlXadesBProperties.java

```
XAdESSignatureParameters parameters = new XAdESSignatureParameters();
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_B);
parameters.setDigestAlgorithm(DigestAlgorithm.SHA512);

parameters.setSigningCertificate(privateKey.getCertificate());
parameters.setCertificateChain(privateKey.getCertificateChain());

BLevelParameters bLevelParameters = parameters.bLevel();
bLevelParameters.addClaimedSignerRole("My Claimed Role");

SignerLocation signerLocation = new SignerLocation();
signerLocation.setCountry("BE");
signerLocation.setStateOrProvince("Luxembourg");
signerLocation.setPostalCode("1234");
signerLocation.setLocality("SimCity");
bLevelParameters.setSignerLocation(signerLocation);

List<String> commitmentTypeIndications = new ArrayList<String>();
commitmentTypeIndications.add("http://uri.etsi.org/01903/v1.2.2#ProofOfOrigin");
commitmentTypeIndications.add("http://uri.etsi.org/01903/v1.2.2#ProofOfApproval");
bLevelParameters.setCommitmentTypeIndications(commitmentTypeIndications);

CommonCertificateVerifier verifier = new CommonCertificateVerifier();
XAdESService service = new XAdESService(verifier);
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);
SignatureValue signatureValue = signingToken.sign(dataToSign, parameters
.getDigestAlgorithm(), privateKey);

DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

This code adds the following elements into the signature :

```
    <xades:SignerRole>
        <xades:ClaimedRoles>
            <xades:ClaimedRole>My Claimed Role</xades:ClaimedRole>
        </xades:ClaimedRoles>
    </xades:SignerRole>
    <xades:SignatureProductionPlace>
        <xades:City>SimCity</xades:City>
        <xades:PostalCode>1234</xades:PostalCode>
        <xades:StateOrProvince>Luxembourg</xades:StateOrProvince>
        <xades:CountryName>BE</xades:CountryName>
    </xades:SignatureProductionPlace>
</xades:SignedSignatureProperties>
<xades:SignedDataObjectProperties>
    <xades:DataObjectFormat ObjectReference="#xml_ref_id">
        <xades:MimeType>text/xml</xades:MimeType>
    </xades:DataObjectFormat>
    <xades:CommitmentTypeIndication>
        <xades:CommitmentTypeId>
            <xades:Identifier>
http://uri.etsi.org/01903/v1.2.2#ProofOfOrigin</xades:Identifier>
            <xades:Identifier>
http://uri.etsi.org/01903/v1.2.2#ProofOfApproval</xades:Identifier>
        </xades:CommitmentTypeId>
        <xades:AllSignedDataObjects/>
    </xades:CommitmentTypeIndication>
```

Handling signature policy

With the new standards the policy handling is linked to -B level. The old -EPES level is not used anymore by the framework. This does not alter the structure of the old signature but only modifies how to control the process of its creation.

The DSS framework allows you to reference a signature policy, which is a set of rules for the creation and validation of an electronic signature. It includes two kinds of text:

- In human readable form: It can be assessed to meet the requirements of the legal and contractual context in which it is being applied.
- In a machine processable form: To facilitate its automatic processing using the electronic rules.

If no signature policy is identified then the signature may be assumed to have been generated or verified without any policy constraints, and hence may be given no specific legal or contractual significance through the context of a signature policy.

The signer may reference the policy either implicitly or explicitly. An implied policy means the signer follows the rules of the policy but the signature does not indicate which policy. It is assumed the choice of policy is clear from the context in which the signature is used and SignaturePolicyIdentifier element

will be empty. When the policy is not implied, the signature contains an ObjectIdentier that uniquely identifies the version of the policy in use. The signature also contains a hash of the policy document to make sure that the signer and verifier agree on the contents of the policy document.

This example demonstrates an implicit policy identifier. To implement this alternative you must set SignaturePolicyId to empty string.

SignXmlXadesBImplicitPolicyTest.java

```
XAdESSignatureParameters parameters = new XAdESSignatureParameters();
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_B);
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

BLevelParameters bLevelParameters = parameters.bLevel();

Policy policy = new Policy();
policy.setId("");

bLevelParameters.setSignaturePolicy(policy);

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create xadesService for signature
XAdESService service = new XAdESService(commonCertificateVerifier);

// Get the SignedInfo segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
DigestAlgorithm digestAlgorithm = parameters.getDigestAlgorithm();
SignatureValue signatureValue = signingToken.sign(dataToSign, digestAlgorithm,
privateKey);

// We invoke the xadesService to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

An XML segment will be added to the signature's qualified and signed properties:

```
<xades:SignaturePolicyIdentifier>
    <xades:SignaturePolicyId>
        <xades:SignaturePolicyImplied/>
    </xades:SignaturePolicyId>
</xades:SignaturePolicyIdentifier>
```

The next example demonstrates an explicit policy identifier. This is obtained by setting -B profile signature policy and assigning values to the policy parameters. The Signature Policy Identifier is a URI or OID that uniquely identifies the version of the policy document. The signature will contain the identifier of the hash algorithm and the hash value of the policy document. The DSS framework does not automatically calculate the hash value; it is to the developer to proceed with the calculation using for example java.security.MessageDigest class (rt.jar). It is important to keep the policy file intact in order to keep the hash constant. It would be wise to make the policy file read-only. See also chapter 7 for further information.

SignXmlXadesBExplicitPolicyTest.java

```
XAdESSignatureParameters parameters = new XAdESSignatureParameters();
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_B);
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

BLevelParameters bLevelParameters = parameters.bLevel();

//Get and use the explicit policy
String signaturePolicyId = "http://www.example.com/policy.txt";
DigestAlgorithm signaturePolicyHashAlgo = DigestAlgorithm.SHA256;
String signaturePolicyDescription = "Policy text to digest";
byte[] signaturePolicyDescriptionBytes = signaturePolicyDescription.getBytes();
byte[] digestedBytes = DSSUtils.digest(signaturePolicyHashAlgo,
signaturePolicyDescriptionBytes);

Policy policy = new Policy();
policy.setId(signaturePolicyId);
policy.setDigestAlgorithm(signaturePolicyHashAlgo);
policy.setDigestValue(digestedBytes);

bLevelParameters.setSignaturePolicy(policy);

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create xadesService for signature
XAdESService service = new XAdESService(commonCertificateVerifier);

// Get the SignedInfo segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
DigestAlgorithm digestAlgorithm = parameters.getDigestAlgorithm();
SignatureValue signatureValue = signingToken.sign(dataToSign, digestAlgorithm,
privateKey);

// We invoke the xadesService to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

The following XML segment will be added to the signature qualified & signed properties (<QualifyingProperties><SignedProperties>):

```
<xades:SignaturePolicyIdentifier>
    <xades:SignaturePolicyId>
        <xades:SigPolicyId>
            <xades:Identifier>http://www.example.com/policy.txt</xades:Identifier>
        </xades:SigPolicyId>
        <xades:SigPolicyHash>
            <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
        <ds:DigestValue>Uw3PxkrX4SpF03jDvkSu6Zqm9UXDxs56FFXeg7MWy0c=</ds:DigestValue>
        </xades:SigPolicyHash>
    </xades:SignaturePolicyId>
</xades:SignaturePolicyIdentifier>
```

XAdES-BASELINE-T

XAdES-BASELINE-T is a signature for which there exists a trusted time associated to the signature. It provides the initial steps towards providing long term validity and more specifically it provides a protection against repudiation. This extension of the signature can be created as well during the generation process as validation process. However, the case when these validation data are not added during the generation process should no longer occur. The XAdES-BASELINE-T trusted time indications must be created before the signing certificate has been revoked or expired and close to the time that the XAdES signature was produced. The XAdES-BASELINE-T form must be built on a XAdES-BASELINE-B form. The DSS framework allows extending the old -BES and -EPES profiles to the new BASELINE-T profile, indeed there is no difference in the structure of the signature.

To implement this profile of signature you must indicate to the service the TSA source, which delivers from each Timestamp Request a Timestamp Response (RFC 3161 (cf. [R11])) containing tokens. Below is the source code that creates a XAdES-BASELINE-T signature. For our example we will use a virtual provider. In a real situation, you can use OnlineTSPSource class encompassing communication with Time Stamping Authority based on RFC 3161 (cf. [R11]) or you can implement your own class using TSPSource interface (see "TSP Sources" chapter for more details).

SignXmlXadesTTest.java

```
// Preparing parameters for the XAdES signature
XAdESSignatureParameters parameters = new XAdESSignatureParameters();
// We choose the level of the signature (-B, -T, -LT, -LTA).
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_T);
// We choose the type of the signature packaging (ENVELOPED, ENVELOPING, DETACHED).
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);
// We set the digest algorithm to use with the signature algorithm. You must use the
// same parameter when you invoke the method sign on the token. The default value is
SHA256
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create XAdES service for signature
XAdESService service = new XAdESService(commonCertificateVerifier);

// Set the TimeStamp
MockTSPSource mockTSPSource;

try {
    mockTSPSource = new MockTSPSource(new CertificateService().generateTspCertificate
(SignatureAlgorithm.RSA_SHA256));
    service.setTspSource(mockTSPSource);
} catch (Exception e) {
    new DSSException("Error during MockTspSource", e);
}

// Get the SignedInfo XML segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
SignatureValue signatureValue = signingToken.sign(dataToSign, parameters
.getDigestAlgorithm(), privateKey);

// We invoke the service to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

The SignatureTimeStamp mandated by the XAdES-T form appears as an unsigned property within the QualifyingProperties:

```
<SignatureTimeStamp Id="time-stamp-28a441da-4030-46ef-80e1-041b66c0cb96">
    <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
    <EncapsulatedTimeStamp
        Id="time-stamp-token-76234ed8-cc15-46fc-aa95-9460dd601cad">
            MIAGCSqGSIb3DQEHAqCAMIACAQMxCzAJBgUrDgMCGg
            UAMIAGCyqGSIb3DQEJEAEEoIAkgARMMEoCAQEGBoIS
            ...
    </EncapsulatedTimeStamp>
</SignatureTimeStamp>
```

Use of online TSP source

If you know the address of an online TSP source, so you can make the call as below:

SignXmlXadesTWithOnlineSourceTest.java

```
// Preparing parameters for the XAdES signature
XAdESSignatureParameters parameters = new XAdESSignatureParameters();
// We choose the level of the signature (-B, -T, -LT, -LTA).
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_T);
// We choose the type of the signature packaging (ENVELOPED, ENVELOPING, DETACHED).
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);
// We set the digest algorithm to use with the signature algorithm. You must use the
// same parameter when you invoke the method sign on the token. The default value is
SHA256
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create XAdES service for signature
XAdESService service = new XAdESService(commonCertificateVerifier);

//Set the Timestamp source
String tspServer = "http://services.globaltrustfinder.com/adss/tsa";
OnlineTSPSource onlineTSPSource = new OnlineTSPSource(tspServer);
service.setTspSource(onlineTSPSource);

// Get the SignedInfo XML segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
SignatureValue signatureValue = signingToken.sign(dataToSign, parameters
.getDigestAlgorithm(), privateKey);

// We invoke the service to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

If the timestamp source is not set a NullPointerException is thrown.

XAdES-BASELINE-LT

This level has to prove that the certification path was valid, at the time of the validation of the signature, up to a trust point according to the naming constraints and the certificate policy constraints

from the "Signature Validation Policy". It will add to the signature the CertificateValues and RevocationValues unsigned properties. The CertificateValues element contains the full set of certificates that have been used to validate the electronic signature, including the signer's certificate. However, it is not necessary to include one of those certificates, if it is already present in the ds:KeyInfo element of the signature. This is like DSS framework behaves. In order to find a list of all the certificates and the list of all revocation data, an automatic process of signature validation is executed. To carry out this process an object called CertificateVerifier must be passed to the service. The implementer must set some of its properties like par example the source of trusted certificates. The code below shows how to use the default parameters with this object. Please refer to "The Signature Validation" chapter to have the further information. It also includes an example of how to implement this level of signature:

SignXmlXadesLTTest.java

```
// Preparing parameters for the XAdES signature
XAdESSignatureParameters parameters = new XAdESSignatureParameters();
// We choose the level of the signature (-B, -T, -LT, -LTA).
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_LT);
// We choose the type of the signature packaging (ENVELOPED, ENVELOPING, DETACHED).
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);
// We set the digest algorithm to use with the signature algorithm. You must use the
// same parameter when you invoke the method sign on the token. The default value is
SHA256
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();

CommonsDataLoader commonsHttpDataLoader = new CommonsDataLoader();

KeyStoreCertificateSource keyStoreCertificateSource = new KeyStoreCertificateSource(new
File("src/main/resources/keystore.p12"), "PKCS12", "dss-password");

TrustedListsCertificateSource tslCertificateSource = new TrustedListsCertificateSource();
tslCertificateSource.setDssKeyStore(keyStoreCertificateSource);

TSLRepository tslRepository = new TSLRepository();
tslRepository.setTrustedListsCertificateSource(tslCertificateSource);

TSLValidationJob job = new TSLValidationJob();
job.setDataLoader(commonsHttpDataLoader);
job.setDssKeyStore(keyStoreCertificateSource);
```

```
job.setLotlUrl("https://ec.europa.eu/information_society/policy/esignature/trusted-
list/tl-mp.xml");
job.setLotlCode("EU");
job.setRepository(tslRepository);
job.refresh();

commonCertificateVerifier.setTrustedCertSource(tslCertificateSource);

OnlineCRLSource onlineCRLSource = new OnlineCRLSource();
onlineCRLSource.setDataLoader(commonsHttpDataLoader);
commonCertificateVerifier.setCrlSource(onlineCRLSource);

OnlineOCSPSource onlineOCSPSource = new OnlineOCSPSource();
onlineCRLSource.setDataLoader(commonsHttpDataLoader);
commonCertificateVerifier.setOcspSource(onlineOCSPSource);

// Create XAdES service for signature
XAdESService service = new XAdESService(commonCertificateVerifier);
try {
    service.setTspSource(getMockTSPSource());
} catch (Exception e) {
    new DSSException("Error during MockTspSource", e);
}

// Get the SignedInfo XML segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
SignatureValue signatureValue = signingToken.sign(dataToSign, parameters
.getDigestAlgorithm(), privateKey);

// We invoke the service to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

The following XML segment will be added to the signature qualified and unsigned properties:

```
<CertificateValues>
    <EncapsulatedX509Certificate>
        MIIFNTCCBB2gAwIBAgIBATANB...
    </EncapsulatedX509Certificate>
    <EncapsulatedX509Certificate>
        MIIFsjCCBJqgAwIBAgIDAMoBM...
    </EncapsulatedX509Certificate>
    <EncapsulatedX509Certificate>
        MIIFRjCCBC6gAwIBAgIBATANB...
    </EncapsulatedX509Certificate>
</CertificateValues>
<RevocationValues>
    <OCSPValues>
        <EncapsulatedOCSPValue>
            MIIGzAoBAKCCBsUwggbBBgkr...
        </EncapsulatedOCSPValue>
    </OCSPValues>
</RevocationValues>
```

The use of online sources can significantly increase the execution time of the signing process. For testing purpose you can create your own source of data.

In last example the CommonsHttpDataLoader is used to provide the communication layer for HTTP protocol. Each source which need to go through the network to retrieve data need to have this component set.

XAdES-BASELINE-LTA

When the cryptographic data becomes weak and the cryptographic functions become vulnerable the auditor should take steps to maintain the validity of the signature. The XAdES-BASELINE-A form uses a simple approach called "archive validation data". It adds additional time-stamps for archiving signatures in a way that they are still protected, but also to be able to prove that the signatures were validated at the time when the used cryptographic algorithms were considered safe. The time-stamping process may be repeated every time the protection used becomes weak. Each time-stamp needs to be affixed before either the signing key or the algorithms used by the TSA are no longer secure. XAdES-A form adds the ArchiveTimestamp element within the UnsignedSignatureProperties and may contain several ArchiveTimestamp elements.

Below is an example of the implementation of this level of signature (but in practice, we will rather extend the signature to this level when there is a risk that the cryptographic functions become vulnerable or when one of certificates arrives to its expiration date):

```
...
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_LTA);
...
```

The following XML segment will be added to the signature qualified and unsigned properties:

```
<ns4:ArchiveTimeStamp
    Id="time-stamp-22b92602-2670-410e-888f-937c5777c685">
    <ds:CanonicalizationMethod
        Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
    <EncapsulatedTimeStamp
        Id="time-stamp-token-0bd5aaf3-3850-4911-a22d-c98dcaca5cea">MIAGCSqGSDHAqCAM···
    </EncapsulatedTimeStamp>
</ns4:ArchiveTimeStamp>
```

Various settings

Trust anchor inclusion policy

It is possible to indicate to the framework if the certificate related to the trust anchor should be included to the signature or not. The setter #setTrustAnchorBPPolicy of the BLevelParameters class should be used for this purpose.

This rule applies as follows: when -B level is constructed the trust anchor is not included, when -LT level is constructed the trust anchor is included.

when trust anchor baseline profile policy is defined only the certificates previous to the trust anchor are included when -B level is constructed.

Multiple signatures

In everyday life, there are many examples where it is necessary to have multiple signatures covering the same document, such as a contract to purchase a vehicle. Depending on the fact that the order of the signatures is important or not, we can establish two basic categories for multiple signatures:

- independent signatures;
- countersignatures.

Independent signatures are parallel signatures where the ordering of the signatures is not important. The computation of these signatures is performed on exactly the same input but using different private keys.

The XML Signature Extension (XAdES)

The -B level contains immutable signed properties. Once this level is created, these properties cannot be changed.

The levels -T/-LT/-LTA add unsigned properties to the signature. This means that the properties of these levels could be added afterwards to any AdES signature. This addition helps to make the signature more resistant to cryptographic attacks on a longer period of time. The extension of the signature is incremental, i.e. when you want to extend the signature to the level -LT the lower level (-T) will also be added. The whole extension process is implemented by reusing components from signature production. To extend a signature we proceed in the same way as in the case of a signature, except that you have to call the function "extendDocument" instead of the "sign" function. Note that when the document is signed with several signatures then they are all extended.

XAdES-BASELINE-T

The XAdES-BASELINE-T trusted time indications have to be created before a certificate has been revoked or expired and close to the time that the XAdES signature was produced. It provides a protection against repudiation. The framework adds the timestamp only if there is no timestamp or there is one but the creation of a new extension of the level-T is deliberate (using another TSA). It is not possible to extend a signature which already incorporates higher level as -LT or -LTA. In the theory it would be possible to add another -T level when the signature has already reached level -LT but the framework prevents this operation. Note that if the signed document contains multiple signatures, then all the signatures will be extended to level -T. It is also possible to sign a document directly at level -T.

Here is an example of creating an extension of type T:

ExtendSignXmlXadesBToTTest.java

```
DSSDocument document = new FileDocument("src/test/resources/signedXmlXadesB.xml");

XAdESSignatureParameters parameters = new XAdESSignatureParameters();
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_T);

CommonCertificateVerifier certificateVerifier = new CommonCertificateVerifier();
XAdESService xadesService = new XAdESService(certificateVerifier);
try{
    xadesService.setTspSource(getMockTSPSource());
}catch (Exception e) {
    new DSSException("Error during MockTspSource",e);
}

DSSDocument extendedDocument = xadesService.extendDocument(document, parameters);
```

Here is the result of adding a new extension of type-T to an already existing -T level signature:

```
<UnsignedSignatureProperties>
    <SignatureTimeStamp Id="time-stamp-b16a2552-b218-4231-8982-40057525fbb5">
        <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
        <EncapsulatedTimeStamp Id="time-stamp-token-39fbf78c-9cec-4cc1-ac21-a467d2238405
">          MIAGCSqGSIb3DQEHAq...
        </EncapsulatedTimeStamp>
    </SignatureTimeStamp>
    <SignatureTimeStamp Id="time-stamp-5ffab0d9-863b-414a-9690-a311d3e1af1d">
        <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
        <EncapsulatedTimeStamp Id="time-stamp-token-87e8c599-89e5-4fb3-a32a-e5e2a40073ad
">          MIAGCSqGSIb3DQEHAq...
        </EncapsulatedTimeStamp>
    </SignatureTimeStamp>
</UnsignedSignatureProperties>
```

XAdES-BASELINE-LT and -LTA

For these types of extensions, the procedure to follow is the same as the case of the extension of type T. Please refer to the chapter XAdES Profiles (XAdES) to know specific parameters for each level of signature and which must be positioned.

XAdES and specific schema version

Some signatures may have been created with an older version of XAdES standard using different schema definition. To take into account the validation of such signatures the class eu.europa.esig.dss.xades.validation.XPathQueryHolder was created. This class includes all XPath queries which are used to explore the elements of the signature. It is now easy to extend this class in order to define specific queries to a given schema. The DSS framework proposes in standard the class eu.europa.esig.dss.xades.validation.XAdES111XPathQueryHolder that defines the XPath queries for the version "http://uri.etsi.org/01903/v1.1.1#" of XAdES standard.

When carrying out the validation process of the signature, the choice of query holder to be used is taken by invoking the method: eu.europa.esig.dss.xades.validation.XPathQueryHolder#canUseThisXPathQueryHolder

This choice is made based on the namespace. If the namespace is: http://uri.etsi.org/01903/v1.3.2# then the default query holder is used, if the namespace is http://uri.etsi.org/01903/v1.1.1# the XAdES111XPathQueryHolder is used. The element used to choose the namespace is "QualifyingProperties".

To implement another query holder the class XPathQueryHolder must be extended, new XPath queries defined and the method canUseThisXPathQueryHolder overridden.

In case there is a need to use only a specific query holder the following steps should be followed:

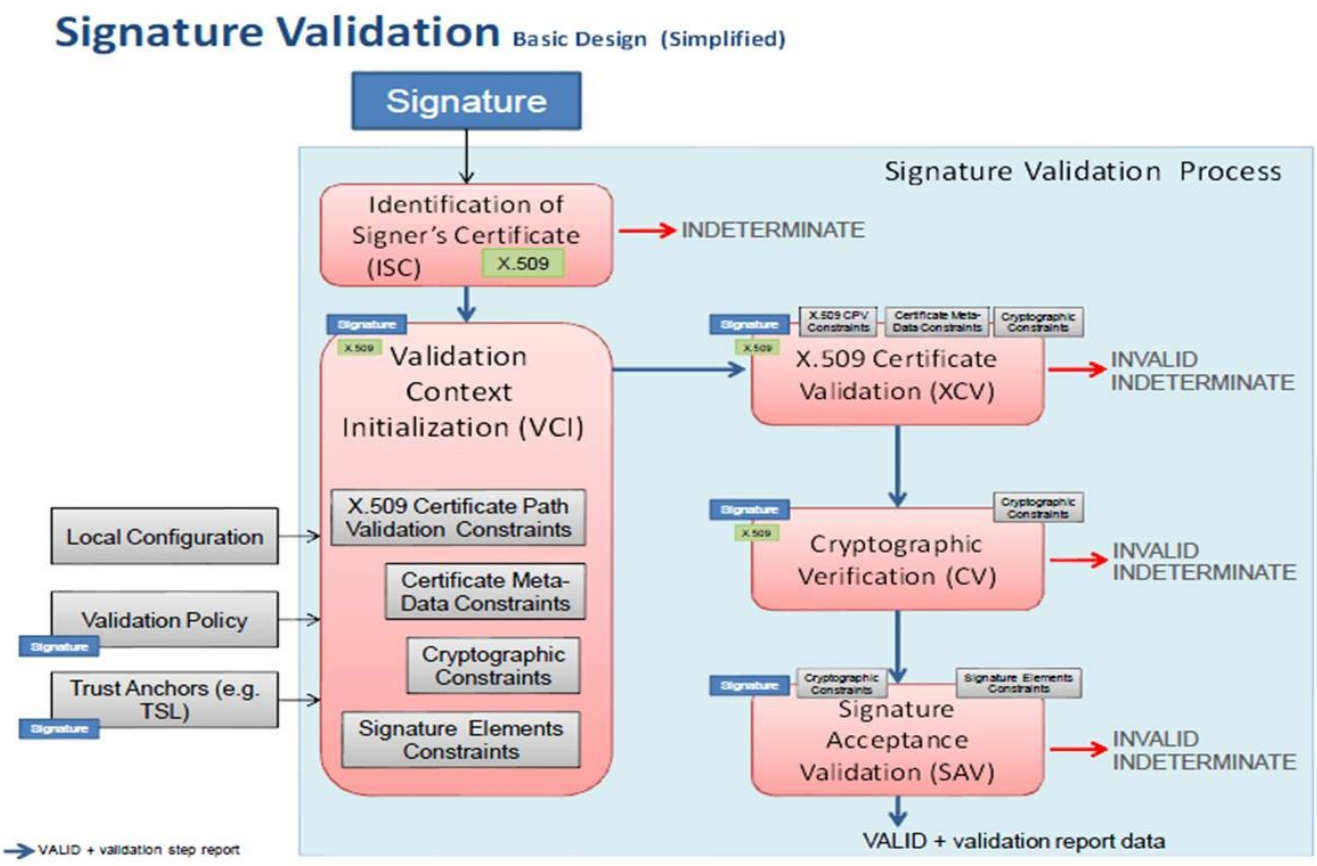
- Call: eu.europa.esig.dss.xades.validation.XMLDocumentValidator#clearQueryHolders
- Call: eu.europa.esig.dss.xades.validation.XMLDocumentValidator#addXPathQueryHolder and pass the specific query holder

The signature validation

Generally and following ETSI standard, the validation process of an electronic signature must provide one of the three following status: INVALID, VALID or INDETERMINATE. A VALID response indicates that the signature has passed verification and it complies with the signature validation policy. An INVALID response indicates that either the signature format is incorrect or that the digital signature value fails verification. An INDETERMINATE validation response indicates that the format and digital signature verifications have not failed but there is insufficient information to determine if the electronic signature is valid. For each of the validation checks, the validation process must provide information justifying the reasons for the resulting status indication as a result of the check against the applicable constraints. In addition, the ETSI standard defines a consistent and accurate way for justifying statuses under a set of sub-indications.

Validation Process

Since version 3 of the DSS framework the validation process is based on the new ETSI standard [R12]. It is driven by the validation policy and allows long term signature validation. It not only verifies the existence of certain data and their validity, but it also checks the temporal dependences between these elements. The signature check is done following basic building blocks. On the simplified diagram below, showing the process of the signature validation, you can follow the relationships between each building block which represents a logic set of checks used in validation process.



Note that the current version of the framework during the validation process does not indicate what part of document was signed. However, in the case of the XAdES signature XPath transformations present in the signature will be applied, in the case of CAdES or PAdES signature the whole document must be signed.

At the end of the validation process three reports are created. They contain the different details level concerning the validation result. They provide three kinds of visions of validation process: macroscopic, microscopic and input data. For more information about these reports, please refer to "Simple Report" chapter.

Below is the simplest example of the validation of the signature of a document. The first thing to do is instantiating an object named validator, which orchestrates the verification of the different rules. To perform this it is necessary to invoke a static method fromDocument() on the abstract class SignedDocumentValidator. This method returns the object in question whose type is chosen dynamically based on the type of source document. The DSS framework provides four types of validators:

- XMLDocumentValidator,
- CMSDocumentValidator,
- PDFDocumentValidator,
- ASiCXMLDocumentValidator.

The next step is to create an object that will check the status of a certificate using the Trusted List

model (see "Trusted Lists of Certification Service Provider" for more information). In our example, this object is instantiated from the TrustedListCertificateVerifier class. In turn, this object needs an OCSP and/or CRL source and a TSL source (which defines how the certificates are retrieved from the Trusted Lists). See chapter "Management of CRL and OCSP Sources" for more information concerning sources.

Note this validation process uses the default validation policy which can be changed by each implementer. To see how to use a customised validation policy, please refer to "5.4" More information can be found on joinup: https://joinup.ec.europa.eu/asset/sd-dss/topic/how-validate-signature-real-crl-ocsp-and-certificate-sources#comment-15459

ValidateSignedXmlXadesBTest.java

```
// To be able to validate our fake signature, we must define one of the certificates in
the chain as trusted anchor.
// If you have a real signature for which it is possible to build the chain till the TSL
then just skip this point.
preparePKCS12TokenAndKey();
final CertificateToken[] certificateChain = privateKey.getCertificateChain();
final CertificateToken trustedCertificate = certificateChain[0];

// Already signed document - Created with the SignXmlXadesB Class
DSSDocument document = new FileDocument(new File("src/test/resources/signedXmlXadesB.xml
"));
SignedDocumentValidator validator = SignedDocumentValidator.fromDocument(document);

CommonCertificateVerifier verifier = new CommonCertificateVerifier();
AlwaysValidOCSPSource ocspSource = new AlwaysValidOCSPSource();
verifier.setOcspSource(ocspSource);

/**
 * This Trusted List Certificates Source points to
 * "https://ec.europa.eu/information_society/policy/esignature/trusted-list/tl-mp.xml"
 */
MockTSLCertificateSource trustedCertSource = new MockTSLCertificateSource();
ServiceInfo mockServiceInfo = new MockServiceInfo();
trustedCertSource.addCertificate(trustedCertificate, mockServiceInfo);
verifier.setTrustedCertSource(trustedCertSource);

validator.setCertificateVerifier(verifier);

Reports reports = validator.validateDocument();
SimpleReport simpleReport = reports.getSimpleReport();
```

It is also possible to use the real sources of OSCP, CRL and certificates:

ValidateXmlXadesLTWithOnlineSourcesTest.java

```
// To be able to validate our fake signature, we must define one of the certificates in
the chain as trusted anchor.
// If you have a real signature for which it is possible to build the chain till the TSL
then just skip this point.
preparePKCS12TokenAndKey();
final CertificateToken[] certificateChain = privateKey.getCertificateChain();
final CertificateToken trustedCertificate = certificateChain[0];

// Already signed document
DSSDocument document = new FileDocument(new File("
src/test/resources/signedXmlXadesLT.xml"));

SignedDocumentValidator validator = SignedDocumentValidator.fromDocument(document);

CommonsDataLoader commonsDataLoader = new CommonsDataLoader();

CommonCertificateVerifier verifier = new CommonCertificateVerifier();
OnlineCRLSource crlSource = new OnlineCRLSource();
crlSource.setDataLoader(commonsDataLoader);
verifier.setCrlSource(crlSource);

OnlineOCSPSource ocspSource = new OnlineOCSPSource();
// The default OCSPDataLoader is created. You can also create your own HttpDataLoader.
verifier.setOcspSource(ocspSource);

// SEE NOTE 1
FileCacheDataLoader fileCacheDataLoader = new FileCacheDataLoader();
File cacheFolder = new File("/temp");
fileCacheDataLoader.setFileCacheDirectory(cacheFolder);

KeyStoreCertificateSource keyStoreCertificateSource = new KeyStoreCertificateSource(new
File("src/main/resources/keystore.p12"), "PKCS12", "dss-password");

TrustedListsCertificateSource certificateSource = new TrustedListsCertificateSource();
certificateSource.setDssKeyStore(keyStoreCertificateSource);

TSLRepository tslRepository = new TSLRepository();
tslRepository.setTrustedListsCertificateSource(certificateSource);

TSLValidationJob job = new TSLValidationJob();
job.setDataLoader(new CommonsDataLoader());
job.setDssKeyStore(keyStoreCertificateSource);
job.setLotlUrl("https://ec.europa.eu/information_society/policy/esignature/trusted-
list/tl-mp.xml");
job.setLotlCode("EU");
job.setRepository(tslRepository);

job.refresh();
```

```
certificateSource.addCertificate(trustedCertificate, new MockServiceInfo());
verifier.setTrustedCertSource(certificateSource);

verifier.setDataLoader(fileCacheDataLoader);

validator.setCertificateVerifier(verifier);

Reports reports = validator.validateDocument();
SimpleReport simpleReport = reports.getSimpleReport();
DetailedReport detailedReport = reports.getDetailedReport();
```

When using the TrustedListsCertificateSource class, for performance reasons, consider creating a single instance of this class and initialize it only once.

In general, the signature must cover the entire document so that the DSS framework can validate it. However, for example in the case of a XAdES signature, some transformations can be applied on the XML document. They can include operations such as canonicalization, encoding/decoding, XSLT, XPath, XML schema validation, or XInclude. XPath transforms permit the signer to derive an XML document that omits portions of the source document. Consequently those excluded portions can change without affecting signature validity. Note that the current version of the framework can sign only the entire document.

EU Trusted Lists of Certification Service Providers

On 16 October 2009 the European Commission adopted a Decision setting out measures facilitating the use of procedures by electronic means through the "points of single contact" under the Services Directive. One of the measures adopted by the Decision consisted in the obligation for Member States to establish and publish by 28.12.2009 their Trusted List of supervised/accredited certification service providers issuing qualified certificates to the public. The objective of this obligation is to enhance cross-border use of electronic signatures by increasing trust in electronic signatures originating from other Member States. The Decision was updated several times since 16.10.2009; the last amendment was made on 01.02.2014. The consolidated version is available here for information.

In order to allow access to the trusted lists of all Member States in an easy manner, the European Commission has published a central list with links to national "trusted lists". This central list will now be designated in the document under the abbreviation LOTL.

TODO

Validation Result Materials

The result of the validation process consists of three elements:

- the simple report,
- the detailed report and,
- the diagnostic data.

All these reports are encoded using XML, which allows the implementer to easily manipulate and extract information for further analysis. You will find below a detailed description of each of these elements.

Simple Report

This is a sample of the simple validation report.

Simple Report

```
<SimpleReport xmlns="http://dss.markt.ec.europa.eu/validation/diagnostic">
    <Policy>
        <PolicyName>QES AdESQC TL based</PolicyName>
        <PolicyDescription>Validate electronic signatures and...</PolicyDescription>
    </Policy>
    <ValidationTime>2013-11-26T12:18:26Z</ValidationTime>
    <DocumentName>signature_tobevalidated.xml</DocumentName>
    <SignatureForm />
    <Signature Id="signature-id9c" SignatureFormat="XAdES-BASELINE-LTA">
        <SigningTime>2013-11-26T11:18:24Z</SigningTime>
        <SignedBy>user a rsa</SignedBy>
        <Indication>VALID</Indication>
        <Info Field="TimestampProductionTime">2013-11-26T11:18:24Z</Info>
        <SignatureLevel>AdES</SignatureLevel>
    </Signature>
    <ValidSignaturesCount>1</ValidSignaturesCount>
    <SignaturesCount>1</SignaturesCount>
</SimpleReport>
```

The result of the validation process is based on very complex rules. The purpose of this report is to make as simple as possible the information while keeping the most important elements. Thus the end user can, at a glance, have a synthetic view of the validation. To build this report the framework uses some simple rules and the detailed report as input.

Detailed Report

This is a sample of the detailed validation report. Its structure is based on the ETSI standard [R12] and is built around Basic Building Blocks, Basic Validation Data, Timestamp Validation Data, AdES-T Validation Data and Long Term Validation Data. Some segments were deleted to make reading easier. They are marked by three dots:

Detailed Report

```
<ValidationData xmlns="http://dss.markt.ec.europa.eu/validation/diagnostic">
    <BasicBuildingBlocks>
        <Signature Id="signature-id9ca819444e4476b476ee29c96943db82">
            <ISC>...
            <VCI>...
            <CV>...
            <SAV>...
            <XCV>...
            <Conclusion>
                <Indication>VALID</Indication>
            </Conclusion>
        </Signature>
    </BasicBuildingBlocks>
    <BasicValidationData>
        <Signature Id="signature-id9ca819444e4476b476ee29c96943db82">
            <Conclusion>
                <Indication>VALID</Indication>
            </Conclusion>
        </Signature>
    </BasicValidationData>
    <TimestampValidationData>
        <Signature Id="signature-id9ca819444e4476b476ee29c96943db82">
            <Timestamp Category="SIGNATURE_TIMESTAMP" Id="1">
                <BasicBuildingBlocks>...
                    <Conclusion>
                        <Indication>VALID</Indication>
                    </Conclusion>
                </BasicBuildingBlocks>
            </Timestamp>
            <Timestamp Category="ARCHIVE_TIMESTAMP" Id="2">
                <BasicBuildingBlocks>...
                    <Conclusion>
                        <Indication>VALID</Indication>
                    </Conclusion>
                </BasicBuildingBlocks>
            </Timestamp>
        </Signature>
    </TimestampValidationData>
    <AdESTValidationData>
        <Signature Id="signature-id9ca819444e4476b476ee29c96943db82">...
            <Conclusion>
                <Indication>VALID</Indication>
                <Info Field="TimestampProductionTime">2013-11-26T11:18:24Z</Info>
            </Conclusion>
        </Signature>
    </AdESTValidationData>
    <LongTermValidationData>
```

```
        <Signature Id="signature-id9ca819444e4476b476ee29c96943db82">...
        </Signature>
    </LongTermValidationData>
</ValidationData>
```

For example the Basic Building Blocks are divided into five elements:

- ISC - Identification of Signer's Certificate
- VCI - Validation Context Initialization
- XCV - X.509 Certificate Validation
- CV - Cryptographic Verification
- SAV - Signature Acceptance Validation

Each block contains a number of rules that are executed sequentially. The rules are driven by the constraints defined in the validation policy. The result of each rule is OK or NOT OK. The process is stopped when the first rule fails. Each block also contains a conclusion. If all rules are met then the conclusion node indicates VALID. Otherwise INVALID or INDETERMINATE indication is returned depending on the ETSI standard definition.

Diagnostic Data

This is a data set constructed from the information contained in the signature itself, but also from information retrieved dynamically as revocation data and information extrapolated as the mathematical validity of a signature. All this information is independent of the applied validation policy. Two different validation policies applied to the same diagnostic data can lead to different results.

This is an example of the diagnostic data for a XAdES signature. Certain fields and certain values were trimmed or deleted to make reading easier:

Diagnostic Data

```
<DiagnosticData xmlns="http://dss.markt.ec.europa.eu/validation/diagnostic">
   <DocumentName>···\XAdES detachedMOAID G2_qual_valid.xml</DocumentName>
   <Signature Id="Signature-6362c43e-1">
      <DateTime>2013-12-20T13:34:13Z</DateTime>
      <SignatureFormat>XAdES_BASELINE_B</SignatureFormat>
      <BasicSignature>
         <EncryptionAlgoUsedToSignThisToken>ECDSA</EncryptionAlgoUsedToSignThisToken>
         <KeyLengthUsedToSignThisToken>192</KeyLengthUsedToSignThisToken>
         <DigestAlgoUsedToSignThisToken>RIPEMD160</DigestAlgoUsedToSignThisToken>
         <ReferenceDataFound>true</ReferenceDataFound>
         <ReferenceDataIntact>true</ReferenceDataIntact>
         <SignatureIntact>true</SignatureIntact>
         <SignatureValid>true</SignatureValid>
```

```
      </BasicSignature>
      <SigningCertificate Id="24">
         <DigestValueMatch>true</DigestValueMatch>
         <IssuerSerialMatch>true</IssuerSerialMatch>
      </SigningCertificate>
      <CertificateChain>
         <ChainCertificate Id="24">
            <Source>SIGNATURE</Source>
         </ChainCertificate>
         <ChainCertificate Id="12">
            <Source>TRUSTED_LIST</Source>
         </ChainCertificate>
      </CertificateChain>
      <ContentType>text/xml</ContentType>
      <Policy>
         <Id/>
         <Identified>false</Identified>
         <Status>true</Status>
         <DigestAlgorithmsEqual>false</DigestAlgorithmsEqual>
      </Policy>
   </Signature>
   <UsedCertificates>
      <Certificate Id="17">
         <SubjectDistinguishedName Format="CANONICAL">…</SubjectDistinguishedName>
         <SubjectDistinguishedName Format="RFC2253">…</SubjectDistinguishedName>
         <IssuerDistinguishedName Format="CANONICAL">…</IssuerDistinguishedName>
         <IssuerDistinguishedName Format="RFC2253">…</IssuerDistinguishedName>
         <IssuerCertificate>0</IssuerCertificate>
         <SerialNumber>1026306</SerialNumber>
         <DigestAlgAndValue>
            <DigestMethod>RIPEMD160</DigestMethod>
            <DigestValue>5HXTKVzoWuDjBgk9RVDJCzQXE+M=</DigestValue>
         </DigestAlgAndValue>
         <NotAfter>2018-09-11T12:14:19Z</NotAfter>
         <NotBefore>2013-09-11T14:14:19Z</NotBefore>
         <PublicKeySize>2048</PublicKeySize>
         <PublicKeyEncryptionAlgo>RSA</PublicKeyEncryptionAlgo>
         <IdKpOCSPSigning>true</IdKpOCSPSigning>
         <BasicSignature>
            <EncryptionAlgoUsedToSignThisToken>RSA</EncryptionAlgoUsedToSignThisToken>
            <KeyLengthUsedToSignThisToken>?</KeyLengthUsedToSignThisToken>
            <DigestAlgoUsedToSignThisToken>SHA1</DigestAlgoUsedToSignThisToken>
            <ReferenceDataFound>true</ReferenceDataFound>
            <ReferenceDataIntact>true</ReferenceDataIntact>
            <SignatureIntact>true</SignatureIntact>
            <SignatureValid>true</SignatureValid>
         </BasicSignature>
         <Trusted>true</Trusted>
```

```
            <SelfSigned>false</SelfSigned>
         </Certificate>
         <Certificate Id="24">
            <SubjectDistinguishedName Format="CANONICAL">…</SubjectDistinguishedName>
            <SubjectDistinguishedName Format="RFC2253">…</SubjectDistinguishedName>
            <IssuerDistinguishedName Format="CANONICAL">…</IssuerDistinguishedName>
            <IssuerDistinguishedName Format="RFC2253">…</IssuerDistinguishedName>
            <IssuerCertificate>12</IssuerCertificate>
            <SerialNumber>397752</SerialNumber>
            <DigestAlgAndValue>
               <DigestMethod>RIPEMD160</DigestMethod>
               <DigestValue>5vllGPyjfO0f0mjk2kyVo8lPsGs=</DigestValue>
            </DigestAlgAndValue>
            <NotAfter>2014-10-13T08:28:09Z</NotAfter>
            <NotBefore>2009-10-13T08:28:09Z</NotBefore>
            <PublicKeySize>192</PublicKeySize>
            <PublicKeyEncryptionAlgo>ECDSA</PublicKeyEncryptionAlgo>
            <BasicSignature>
               <EncryptionAlgoUsedToSignThisToken>RSA</EncryptionAlgoUsedToSignThisToken>
               <KeyLengthUsedToSignThisToken>2048</KeyLengthUsedToSignThisToken>
               <DigestAlgoUsedToSignThisToken>SHA1</DigestAlgoUsedToSignThisToken>
               <ReferenceDataFound>true</ReferenceDataFound>
               <ReferenceDataIntact>true</ReferenceDataIntact>
               <SignatureIntact>true</SignatureIntact>
               <SignatureValid>true</SignatureValid>
            </BasicSignature>
            <SigningCertificate Id="12">
               <DigestValueMatch>true</DigestValueMatch>
               <IssuerSerialMatch>true</IssuerSerialMatch>
            </SigningCertificate>
            <CertificateChain>
               <ChainCertificate Id="12">
                  <Source>TRUSTED_LIST</Source>
               </ChainCertificate>
            </CertificateChain>
            <Trusted>false</Trusted>
            <SelfSigned>false</SelfSigned>
            <QCStatement>
               <QCP>false</QCP>
               <QCPPlus>true</QCPPlus>
               <QCC>true</QCC>
               <QCSSCD>false</QCSSCD>
            </QCStatement>
            <TrustedServiceProvider>
               <TSPName>A-Trust Gesellschaft für Sicherheitssysteme im elektronischen
Datenverkehr GmbH</TSPName>
               <TSPServiceName>a-sign-Premium-Sig-02 (key no. 1)</TSPServiceName>
               <TSPServiceType>http://uri.etsi.org/TrstSvc/Svctype/CA/QC</TSPServiceType>
```

```
            <Status>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/accredited</Status>
            <StartDate>2004-12-14T23:00:00Z</StartDate>
            <WellSigned>true</WellSigned>
         </TrustedServiceProvider>
         <Revocation>
            <Source>OCSPToken</Source>
            <SourceAddress>http://ocsp.a-trust.at/ocsp</SourceAddress>
            <Status>false</Status>
            <DateTime>2014-01-08T05:23:38Z</DateTime>
            <Reason>CRLReason: keyCompromise</Reason>
            <IssuingTime>2014-02-12T10:20:26Z</IssuingTime>
            <BasicSignature>
               <EncryptionAlgoUsedToSignThisToken>RSA</EncryptionAlgoUsedToSignThisToken>
               <KeyLengthUsedToSignThisToken>2048</KeyLengthUsedToSignThisToken>
               <DigestAlgoUsedToSignThisToken>SHA1</DigestAlgoUsedToSignThisToken>
               <ReferenceDataFound>true</ReferenceDataFound>
               <ReferenceDataIntact>true</ReferenceDataIntact>
               <SignatureIntact>true</SignatureIntact>
               <SignatureValid>true</SignatureValid>
            </BasicSignature>
            <SigningCertificate Id="17">
               <DigestValueMatch>true</DigestValueMatch>
               <IssuerSerialMatch>true</IssuerSerialMatch>
            </SigningCertificate>
            <CertificateChain>
               <ChainCertificate Id="17">
                  <Source>TRUSTED_LIST</Source>
               </ChainCertificate>
            </CertificateChain>
         </Revocation>
      </Certificate>
      <Certificate Id="12">
         <SubjectDistinguishedName Format="CANONICAL">…</SubjectDistinguishedName>
         <SubjectDistinguishedName Format="RFC2253">…</SubjectDistinguishedName>
         <IssuerDistinguishedName Format="CANONICAL">…</IssuerDistinguishedName>
         <IssuerDistinguishedName Format="RFC2253">…</IssuerDistinguishedName>
         <IssuerCertificate>0</IssuerCertificate>
         <SerialNumber>58531</SerialNumber>
         <DigestAlgAndValue>
            <DigestMethod>RIPEMD160</DigestMethod>
            <DigestValue>CC+ntcCafY5zsjFmY3gQsbPzrU0=</DigestValue>
         </DigestAlgAndValue>
         <NotAfter>2014-12-13T23:00:00Z</NotAfter>
         <NotBefore>2004-12-14T23:00:00Z</NotBefore>
         <PublicKeySize>2048</PublicKeySize>
         <PublicKeyEncryptionAlgo>RSA</PublicKeyEncryptionAlgo>
         <BasicSignature>
            <EncryptionAlgoUsedToSignThisToken>RSA</EncryptionAlgoUsedToSignThisToken>
```

```
            <KeyLengthUsedToSignThisToken>?</KeyLengthUsedToSignThisToken>
            <DigestAlgoUsedToSignThisToken>SHA1</DigestAlgoUsedToSignThisToken>
            <ReferenceDataFound>true</ReferenceDataFound>
            <ReferenceDataIntact>true</ReferenceDataIntact>
            <SignatureIntact>true</SignatureIntact>
            <SignatureValid>true</SignatureValid>
         </BasicSignature>
         <Trusted>true</Trusted>
         <SelfSigned>false</SelfSigned>
      </Certificate>
   </UsedCertificates>
</DiagnosticData>
```

Customised Validation Policy

The validation process may be driven by a set of constraints that are contained in the XML file constraint.xml.

constraint.xml

```
<ConstraintsParameters Name="QES AdESQC TL based" xmlns=
"http://dss.esig.europa.eu/validation/diagnostic">
    <Description>Validate electronic signatures and indicates whether they are Advanced
electronic Signatures (AdES), AdES supported by a Qualified Certificate (AdES/QC) or a
Qualified electronic Signature (QES). All certificates and their related chains
supporting the signatures are validated against the EU Member State Trusted Lists (this
includes signer's certificate and certificates used to validate certificate validity
status services - CRLs, OCSP, and time-stamps).
    </Description>
    <MainSignature>
        <AcceptablePolicies Level="FAIL">
            <Id>ANY_POLICY</Id>
            <Id>NO_POLICY</Id>
        </AcceptablePolicies>
        <ReferenceDataExistence Level="FAIL" />
        <ReferenceDataIntact Level="FAIL" />
        <SignatureIntact Level="FAIL" />
        <SigningCertificate>
            <Recognition Level="FAIL"/>
            <AttributePresent Level="FAIL"/>
            <DigestValuePresent Level="FAIL"/>
            <DigestValueMatch Level="FAIL" />
            <IssuerSerialMatch Level="WARN" />
            <Signed Level="FAIL" />
            <Signature Level="FAIL"/>
            <Expiration Level="FAIL"/>
```

```
            <RevocationDataAvailable Level="FAIL"/>
            <RevocationDataIsTrusted Level="FAIL"/>
            <RevocationDataFreshness Level="WARN"/>
            <ProspectiveCertificateChain Level="FAIL"/>
            <KeyUsage Level="WARN">
                <Id>nonRepudiation</Id>
            </KeyUsage>
            <Revoked Level="FAIL"/>
            <OnHold Level="FAIL"/>
            <TSLValidity Level="WARN"/>
            <TSLStatus Level="WARN"/>
            <TSLStatusAndValidity Level="FAIL"/>
            <Qualification Level="WARN"/>
            <SupportedBySSCD Level="WARN"/>
            <IssuedToLegalPerson Level="INFORM"/>
            <Cryptographic Level="FAIL">
                <AcceptableEncryptionAlgo>
                    <Algo>RSA</Algo>
                    <Algo>DSA</Algo>
                    <Algo>ECDSA</Algo>
                </AcceptableEncryptionAlgo>
                <MiniPublicKeySize>
                    <Algo Size="128">DSA</Algo>
                    <Algo Size="1024">RSA</Algo>
                    <Algo Size="192">ECDSA</Algo>
                </MiniPublicKeySize>
                <AcceptableDigestAlgo>
                    <Algo>SHA1</Algo>
                    <Algo>SHA224</Algo>
                    <Algo>SHA256</Algo>
                    <Algo>SHA384</Algo>
                    <Algo>SHA512</Algo>
                    <Algo>RIPEMD160</Algo>
                </AcceptableDigestAlgo>
            </Cryptographic>
        </SigningCertificate>
        <CACertificate>
            <Signature Level="FAIL"/>
            <Expiration Level="FAIL"/>
            <RevocationDataAvailable Level="FAIL"/>
            <RevocationDataIsTrusted Level="FAIL"/>
            <RevocationDataFreshness Level="WARN"/>
            <Revoked Level="FAIL"/>
            <Cryptographic Level="FAIL">
                <AcceptableEncryptionAlgo>
                    <Algo>RSA</Algo>
                    <Algo>DSA</Algo>
                    <Algo>ECDSA</Algo>
```

```
            </AcceptableEncryptionAlgo>
            <MiniPublicKeySize>
                <Algo Size="128">DSA</Algo>
                <Algo Size="1024">RSA</Algo>
                <Algo Size="192">ECDSA</Algo>
            </MiniPublicKeySize>
            <AcceptableDigestAlgo>
                <Algo>SHA1</Algo>
                <Algo>SHA224</Algo>
                <Algo>SHA256</Algo>
                <Algo>SHA384</Algo>
                <Algo>SHA512</Algo>
                <Algo>RIPEMD160</Algo>
            </AcceptableDigestAlgo>
        </Cryptographic>
    </CACertificate>
    <Cryptographic Level="FAIL">
        <AcceptableEncryptionAlgo>
            <Algo>RSA</Algo>
            <Algo>DSA</Algo>
            <Algo>ECDSA</Algo>
        </AcceptableEncryptionAlgo>
        <MiniPublicKeySize>
            <Algo Size="128">DSA</Algo>
            <Algo Size="1024">RSA</Algo>
            <Algo Size="192">ECDSA</Algo>
        </MiniPublicKeySize>
        <AcceptableDigestAlgo>
            <Algo>SHA1</Algo>
            <Algo>SHA224</Algo>
            <Algo>SHA256</Algo>
            <Algo>SHA384</Algo>
            <Algo>SHA512</Algo>
            <Algo>RIPEMD160</Algo>
        </AcceptableDigestAlgo>
    </Cryptographic>
    <MandatedSignedQProperties>
        <SigningTime Level="FAIL"/>
        <ContentTimeStamp>
            <MessageImprintDataFound Level="FAIL" />
            <MessageImprintDataIntact Level="FAIL" />
        </ContentTimeStamp>
        <!--<ContentType Level="FAIL">1.2.840.113549.1.7.1</ContentType>-->
    </MandatedSignedQProperties>
    <MandatedUnsignedQProperties>
        <CounterSignature>
            <ReferenceDataExistence Level="FAIL" />
            <ReferenceDataIntact Level="FAIL" />
```

```
                <SignatureIntact Level="FAIL" />
            </CounterSignature>
        </MandatedUnsignedQProperties>
    </MainSignature>
    <Timestamp>
        <TimestampDelay Unit="DAYS">0</TimestampDelay>
        <MessageImprintDataFound Level="FAIL"/>
        <MessageImprintDataIntact Level="FAIL"/>
        <RevocationTimeAgainstBestSignatureTime Level="FAIL"/>
        <BestSignatureTimeBeforeIssuanceDateOfSigningCertificate Level="FAIL"/>
        <SigningCertificateValidityAtBestSignatureTime Level="FAIL"/>
        <AlgorithmReliableAtBestSignatureTime Level="FAIL"/>
        <Coherence Level="WARN"/>
        <SigningCertificate>
            <Recognition Level="FAIL"/>
            <Signature Level="FAIL"/>
            <Expiration Level="FAIL"/>
            <RevocationDataAvailable Level="FAIL"/>
            <RevocationDataIsTrusted Level="FAIL"/>
            <RevocationDataFreshness Level="WARN"/>
            <ProspectiveCertificateChain Level="FAIL"/>
            <Revoked Level="FAIL"/>
            <OnHold Level="FAIL"/>
            <TSLStatus Level="FAIL"/>
            <Cryptographic Level="FAIL">
                <AcceptableEncryptionAlgo>
                    <Algo>RSA</Algo>
                    <Algo>DSA</Algo>
                    <Algo>ECDSA</Algo>
                </AcceptableEncryptionAlgo>
                <MiniPublicKeySize>
                    <Algo Size="128">DSA</Algo>
                    <Algo Size="1024">RSA</Algo>
                    <Algo Size="192">ECDSA</Algo>
                </MiniPublicKeySize>
                <AcceptableDigestAlgo>
                    <Algo>SHA1</Algo>
                    <Algo>SHA224</Algo>
                    <Algo>SHA256</Algo>
                    <Algo>SHA384</Algo>
                    <Algo>SHA512</Algo>
                    <Algo>RIPEMD160</Algo>
                </AcceptableDigestAlgo>
            </Cryptographic>
        </SigningCertificate>
        <CACertificate>
            <Signature Level="FAIL"/>
            <Expiration Level="FAIL"/>
```

```
            <RevocationDataAvailable Level="FAIL"/>
            <RevocationDataIsTrusted Level="FAIL"/>
            <RevocationDataFreshness Level="WARN"/>
            <Revoked Level="FAIL"/>
            <Cryptographic Level="FAIL">
                <AcceptableEncryptionAlgo>
                    <Algo>RSA</Algo>
                    <Algo>DSA</Algo>
                    <Algo>ECDSA</Algo>
                </AcceptableEncryptionAlgo>
                <MiniPublicKeySize>
                    <Algo Size="128">DSA</Algo>
                    <Algo Size="1024">RSA</Algo>
                    <Algo Size="192">ECDSA</Algo>
                </MiniPublicKeySize>
                <AcceptableDigestAlgo>
                    <Algo>SHA1</Algo>
                    <Algo>SHA224</Algo>
                    <Algo>SHA256</Algo>
                    <Algo>SHA384</Algo>
                    <Algo>SHA512</Algo>
                    <Algo>RIPEMD160</Algo>
                </AcceptableDigestAlgo>
            </Cryptographic>
        </CACertificate>
    </Timestamp>
    <Revocation>
        <RevocationFreshness Unit="DAYS">0</RevocationFreshness>
        <SigningCertificate>
            <Signature Level="FAIL"/>
            <Expiration Level="FAIL"/>
            <RevocationDataAvailable Level="FAIL"/>
            <RevocationDataIsTrusted Level="FAIL"/>
            <RevocationDataFreshness Level="WARN"/>
            <Revoked Level="FAIL"/>
            <Cryptographic Level="WARN">
                <AcceptableEncryptionAlgo>
                    <Algo>RSA</Algo>
                    <Algo>DSA</Algo>
                    <Algo>ECDSA</Algo>
                </AcceptableEncryptionAlgo>
                <MiniPublicKeySize>
                    <Algo Size="128">DSA</Algo>
                    <Algo Size="1024">RSA</Algo>
                    <Algo Size="192">ECDSA</Algo>
                </MiniPublicKeySize>
                <AcceptableDigestAlgo>
                    <Algo>SHA1</Algo>
```

```
                    <Algo>SHA224</Algo>
                    <Algo>SHA256</Algo>
                    <Algo>SHA384</Algo>
                    <Algo>SHA512</Algo>
                    <Algo>RIPEMD160</Algo>
                </AcceptableDigestAlgo>
            </Cryptographic>
        </SigningCertificate>
        <CACertificate>
            <Signature Level="FAIL"/>
            <Expiration Level="FAIL"/>
            <RevocationDataAvailable Level="FAIL"/>
            <RevocationDataIsTrusted Level="FAIL"/>
            <RevocationDataFreshness Level="WARN"/>
            <Revoked Level="FAIL"/>
            <Cryptographic Level="FAIL">
                <AcceptableEncryptionAlgo>
                    <Algo>RSA</Algo>
                    <Algo>DSA</Algo>
                    <Algo>ECDSA</Algo>
                </AcceptableEncryptionAlgo>
                <MiniPublicKeySize>
                    <Algo Size="128">DSA</Algo>
                    <Algo Size="1024">RSA</Algo>
                    <Algo Size="192">ECDSA</Algo>
                </MiniPublicKeySize>
                <AcceptableDigestAlgo>
                    <Algo>SHA1</Algo>
                    <Algo>SHA224</Algo>
                    <Algo>SHA256</Algo>
                    <Algo>SHA384</Algo>
                    <Algo>SHA512</Algo>
                    <Algo>RIPEMD160</Algo>
                </AcceptableDigestAlgo>
            </Cryptographic>
        </CACertificate>
    </Revocation>
    <Cryptographic />
<!--
    <Cryptographic>
            <AlgoExpirationDate Format="yyyy-MM-dd">
            <Algo Date="2017-02-24">SHA1</Algo>
            <Algo Date="2035-02-24">SHA224</Algo>
            <Algo Date="2035-02-24">SHA256</Algo>
            <Algo Date="2035-02-24">SHA384</Algo>
            <Algo Date="2035-02-24">SHA512</Algo>
            <Algo Date="2017-02-24">RIPEMD160</Algo>
            <Algo Date="2017-02-24">DSA128</Algo>
```

```
            <Algo Date="2015-02-24">RSA1024</Algo>
            <Algo Date="2015-02-24">RSA1536</Algo>
            <Algo Date="2020-02-24">RSA2048</Algo>
            <Algo Date="2020-02-24">RSA3072</Algo>
            <Algo Date="2035-02-24">RSA4096</Algo>
            <Algo Date="2035-02-24">ECDSA192</Algo>
            <Algo Date="2035-02-24">ECDSA256</Algo>
        </AlgoExpirationDate>
    </Cryptographic>
-->
</ConstraintsParameters>
```

CAdES signature and validation

To familiarise yourself with this type of signature it is advisable to read the following document:

- CAdES Specifications (cf. [R02])

To implement this form of signature you can use the XAdES examples. You only need to instantiate the CAdES object service and change the SignatureLevel parameter value. Below is an example of the CAdES-Baseline-B signature:

SignXmlCadesBTest.java

```
// Preparing parameters for the CAdES signature
CAdESSignatureParameters parameters = new CAdESSignatureParameters();
// We choose the level of the signature (-B, -T, -LT, -LTA).
parameters.setSignatureLevel(SignatureLevel.CAdES_BASELINE_B);
// We choose the type of the signature packaging (ENVELOPING, DETACHED).
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPING);
// We set the digest algorithm to use with the signature algorithm. You must use the
// same parameter when you invoke the method sign on the token. The default value is
// SHA256
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create CAdES xadesService for signature
CAdESService service = new CAdESService(commonCertificateVerifier);

// Get the SignedInfo segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
DigestAlgorithm digestAlgorithm = parameters.getDigestAlgorithm();
SignatureValue signatureValue = signingToken.sign(dataToSign, digestAlgorithm,
privateKey);

// We invoke the xadesService to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

PAdES signature and validation

The standard ISO 32000-1 (cf. [R05]) allows defining a file format for portable electronic documents. It is based on PDF 1.7 of Adobe Systems. Concerning the digital signature it supports three operations:

- Adding a digital signature to a document,
- Providing a placeholder field for signatures,
- Checking signatures for validity.

PAdES defines eight different profiles to be used with advanced electronic signature in the meaning of European Union Directive 1999/93/EC (cf. [R10]):

- PAdES Basic - PDF signature as specified in ISO 32000-1 (cf. [R05]). The profile is specified in TS 102 778-2 (cf. [R03]).
- PAdES-BES Profile - based upon CAdES-BES as specified in TS 101 733 (cf. [R02]) with the option of a signature time-stamp (CAdES-T).
- PAdES-EPES profile - based upon CAdES-EPES as specified in TS 101 733 (cf. [R02]). This profile is the same as the PAdES - BES with the addition of a signature policy identifier and optionally a commitment type indication.
- PAdES-LTV Profile - This profile supports the long term validation of PDF Signatures and can be used in conjunction with the above-mentioned profiles.
- Four other PAdES profiles for XML Content.

To familiarise yourself with this type of signature it is advisable to read the documents referenced above.

Below is an example of code to perform a PAdES-BASELINE-B type signature:

SignPdfPadesBTest.java

```
// Preparing parameters for the PAdES signature
PAdESSignatureParameters parameters = new PAdESSignatureParameters();
// We choose the level of the signature (-B, -T, -LT, -LTA).
parameters.setSignatureLevel(SignatureLevel.PAdES_BASELINE_B);
// We choose the type of the signature packaging (ENVELOPING, DETACHED).
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);
// We set the digest algorithm to use with the signature algorithm. You must use the
// same parameter when you invoke the method sign on the token. The default value is
// SHA256
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create PAdESService for signature
PAdESService service = new PAdESService(commonCertificateVerifier);

// Get the SignedInfo segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
DigestAlgorithm digestAlgorithm = parameters.getDigestAlgorithm();
SignatureValue signatureValue = signingToken.sign(dataToSign, digestAlgorithm,
privateKey);

// We invoke the xadesService to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

To add the timestamp to the signature (PAdES-T or LTA), please provide TSP source to the service.

To create PAdES-BASELINE-B level with additional options: signature policy identifier and optionally a commitment type indication, please observe the following example in code 5.

All these parameters are optional.

- **SignaturePolicyOID :** The string representation of the OID of the signature policy to use when signing.
- **SignaturePolicyHashValue :** The value of the hash of the signature policy, computed the same way

as in clause 5.8.1 of CAdES (TS 101 733 (cf. [R02])).

- **SignaturePolicyHashAlgorithm :** The hash function used to compute the value of the SignaturePolicyHashValue entry. Entries must be represented the same way as in table 257 of ISO 32000-1 (cf. [R05]).
- **SignaturePolicyCommitmentType :** If the SignaturePolicyOID is present, this array defines the commitment types that can be used within the signature policy. An empty string can be used to indicate the default commitment type.

If the SignaturePolicyOID is absent, the three other fields defined above will be ignored. If the SignaturePolicyOID is present but the SignaturePolicyCommitmentType is absent, all commitments defined by the signature policy will be used.

The extension of a signature of the level PAdES-BASELINE-B up to PAdES-BASELINE-LTA profile will add the following features:

- Addition of validation data to an existing PDF document which may be used to validate earlier signatures within the document (including PDF signatures and time-stamp signatures).
- Addition of a document time-stamp which protects the existing document and any validation data.
- Further validation data and document time-stamp may be added to a document over time to maintain its authenticity and integrity.

PAdES Visible Signature

The framework also allows to create PDF files with visible signature as specified in TS 102 778-6 (cf. [R03]). In the SignatureParameters object, there's a special attribute named ImageParameters. This parameter let you custom the visual signature (with text, with image or with image and text). Below is an example of code to perform a PADES-BASELINE-B type signature with a visible signature:

SignPdfPadesBVisibleTest.java

```
// Preparing parameters for the PAdES signature
PAdESSignatureParameters parameters = new PAdESSignatureParameters();
// We choose the level of the signature (-B, -T, -LT, -LTA).
parameters.setSignatureLevel(SignatureLevel.PAdES_BASELINE_B);
// We choose the type of the signature packaging (ENVELOPING, DETACHED).
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Initialize visual signature
SignatureImageParameters imageParameters = new SignatureImageParameters();
// the origin is the left and top corner of the page
imageParameters.setxAxis(200);
imageParameters.setyAxis(500);

// Initialize text to generate for visual signature
SignatureImageTextParameters textParameters = new SignatureImageTextParameters();
textParameters.setFont(new Font("serif", Font.PLAIN, 14));
textParameters.setTextColor(Color.BLUE);
textParameters.setText("My visual signature");
imageParameters.setTextParameters(textParameters);

parameters.setImageParameters(imageParameters);

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create PAdESService for signature
PAdESService service = new PAdESService(commonCertificateVerifier);

// Get the SignedInfo segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
DigestAlgorithm digestAlgorithm = parameters.getDigestAlgorithm();
SignatureValue signatureValue = signingToken.sign(dataToSign, digestAlgorithm,
privateKey);

// We invoke the xadesService to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

ASiC signature and validation

When creating a digital signature, the user must choose between different packaging elements, namely enveloping, enveloped or detached. This choice is not obvious, because in one case the signature will alter the signed document and in the other case it is possible to lose the association between the signed document and its signature. That's where the standard ETSI TS 102 918 (cf. [R04]) offers a standardized use of container forms to establish a common way for associating data objects with advanced signatures or time-stamp tokens.

A number of application environments use ZIP based container formats to package sets of files together with meta-information. ASiC technical specification is designed to operate with a range of such ZIP based application environments. Rather than enforcing a single packaging structure, ASiC describes how these package formats can be used to associate advanced electronic signatures with any data objects.

The standard defines two types of containers; the first (ASiC-S) allows you to associate one or more signatures with a single data element. In this case the structure of the signature can be based (in a general way) on a single CAdES signature or on multiple XAdES signatures or finally on a single TST; the second is an extended container (ASiC-E) that includes multiple data objects. Each data object may be signed by one or more signatures which structure is similar to ASiC-S. This second type of container is compatible with OCF, UCF and ODF formats.

For the moment the DSS framework uses only ASIC-S based on XAdES signature type of containers.

This is an example of the source code for signing a document using ASiCS-S based on XAdES-B:

SignPdfASiCSBTest.java

```
// Preparing parameters for the AsicS signature
ASiCSignatureParameters parameters = new ASiCSignatureParameters();
// We choose the level of the signature (-B, -T, -LT, LTA).
parameters.setSignatureLevel(SignatureLevel.ASiC_S_BASELINE_B);
// We choose the underlying format (XAdES or CAdES. XAdES is the default value)
parameters.aSiC().setUnderlyingForm(SignatureForm.XAdES);

// We set the digest algorithm to use with the signature algorithm. You must use the
// same parameter when you invoke the method sign on the token. The default value is
// SHA256
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create ASiCS service for signature
ASiCService service = new ASiCService(commonCertificateVerifier);

// Get the SignedInfo segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information using the
// private key and specified algorithm
DigestAlgorithm digestAlgorithm = parameters.getDigestAlgorithm();
SignatureValue signatureValue = signingToken.sign(dataToSign, digestAlgorithm,
privateKey);

// We invoke the xadesService to sign the document with the signature value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

This is another example of the source code for signing a document using ASiCS-E based on CAdES:

SignPdfASiCSEWithCAdESTest.java

```
// Preparing parameters for the AsicE signature
ASiCSignatureParameters parameters = new ASiCSignatureParameters();

// We choose the level of the signature (-B, -T, -LT or -LTA).
parameters.setSignatureLevel(SignatureLevel.ASiC_E_BASELINE_B);
// We choose CAdES as underlying form
parameters.aSiC().setUnderlyingForm(SignatureForm.CAdES);

// We set the digest algorithm to use with the signature algorithm. You
// must use the
// same parameter when you invoke the method sign on the token. The
// default value is
// SHA256
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

// We set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// We set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create ASiCS service for signature
ASiCService service = new ASiCService(commonCertificateVerifier);

// Get the SignedInfo segment that need to be signed.
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);

// This function obtains the signature value for signed information
// using the
// private key and specified algorithm
DigestAlgorithm digestAlgorithm = parameters.getDigestAlgorithm();
SignatureValue signatureValue = signingToken.sign(dataToSign, digestAlgorithm,
privateKey);

// We invoke the xadesService to sign the document with the signature
// value obtained in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

Please note that you need to pass only few parameters to the service. Other parameters, although are positioned, will be overwritten by the internal implementation of the service. Therefore, the obtained signature is always based on XAdES and of DETACHED packaging.

It is also possible with the framework DSS to make an extension of an ASICS signature to the level XAdES-BASELINE-T or -LT.

Management of signature tokens

The DSS framework is able to create signatures from PKCS#11, PKCS#12 and MS CAPI. Java 6 is inherently capable of communicating with these kinds of KeyStores. To be independent of the signing media, DSS framework uses an interface named SignatureTokenConnection to manage different implementations of the signing process. The base implementation is able to sign a stream of the data in one step. That means that all the data to be signed needs to be sent to the SSCD. This is the case for MS CAPI. As to the PKCS#11 and PKCS#12, which give to the developer a finer control in the signature operation, the DSS framework implements the AsyncSignatureTokenConnection abstract class that permits to execute the digest operation and signature operation in two different threads or even two different hardwares.

This design permits also other card providers/adopters to create own implementations. For example, this can be used for a direct connection to the Smartcard through Java 6 PC/SC.

PKCS#11

PKCS#11 is widely used to access smart cards and HSMs. Most commercial software uses PKCS#11 to access the signature key of the CA or to enrol user certificates. In the DSS framework, this standard is encapsulated in the class Pkcs11SignatureToken.

PKCS11Snippet.java

```
SignatureTokenConnection token = new Pkcs11SignatureToken("C:\\Windows\\System32
\\beidpkcs11.dll");

List<DSSPrivateKeyEntry> keys = token.getKeys();
for (DSSPrivateKeyEntry entry : keys) {
    System.out.println(entry.getCertificate().getCertificate());
}

ToBeSigned toBeSigned = new ToBeSigned("Hello world".getBytes());
SignatureValue signatureValue = token.sign(toBeSigned, DigestAlgorithm.SHA256, keys.get(
0));

System.out.println("Signature value : " + Base64.encodeBase64String(signatureValue
.getValue()));
```

PKCS#12

This standard defines a file format commonly used to store the private key and corresponding public

key certificate protecting them by password.

In order to use this format with the DSS framework you have to go through the class Pkcs12SignatureToken.

PKCS12Snippet.java

```
SignatureTokenConnection token = new Pkcs12SignatureToken("password",
"src/main/resources/user_a_rsa.p12");

List<DSSPrivateKeyEntry> keys = token.getKeys();
for (DSSPrivateKeyEntry entry : keys) {
    System.out.println(entry.getCertificate().getCertificate());
}

ToBeSigned toBeSigned = new ToBeSigned("Hello world".getBytes());
SignatureValue signatureValue = token.sign(toBeSigned, DigestAlgorithm.SHA256, keys.get(
0));

System.out.println("Signature value : " + Base64.encodeBase64String(signatureValue
.getValue()));
```

MS CAPI

If the middleware for communicating with an SSDC provides a CSP based on MS CAPI specification, then to sign the documents you can use MSCAPISignatureToken class.

MSCAPISnippet.java

```
SignatureTokenConnection token = new MSCAPISignatureToken();

List<DSSPrivateKeyEntry> keys = token.getKeys();
for (DSSPrivateKeyEntry entry : keys) {
    System.out.println(entry.getCertificate().getCertificate());
}

ToBeSigned toBeSigned = new ToBeSigned("Hello world".getBytes());
SignatureValue signatureValue = token.sign(toBeSigned, DigestAlgorithm.SHA256, keys.get(
0));

System.out.println("Signature value : " + Base64.encodeBase64String(signatureValue
.getValue()));
```

Other Implementations

As you can see, it is easy to add another implementation of the SignatureTokenConnection, thus enabling the framework to use other API than the provided three (PKCS#11, PKCS#12 and MS CAPI). For example, it is likely that in the future PC/SC will be the preferred way of accessing a Smartcard. Although PKCS#11 is currently the most used API, DSS framework is extensible and can use PC/SC. For our design example we propose to use PC/SC to communicate with the Smartcard.