Digital Signature Service

Table of Contents

1. Generic information

1.1. The Project

The **DSS (Digital Signature Service)** project is an open-source software library, aimed at providing implementation of the standards for Advanced Electronic Signature creation, augmentation and validation in line with European legislation and the eIDAS Regulation in particular.

This project is available in **Java** language.

1.2. 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 familiarize them with the code step by step.

1.3. Scope of the document

This document provides examples of code which allow easy handling of digital signatures. The examples are consistent with the Release 5.10 of DSS framework which can be downloaded via the webpage.

Three main features can be distinguished within the framework :

- The creation of a digital signature;
- The augmentation of a digital signature and;
- The validation of a digital signature.

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

- Forms of digital signatures: XAdES, CAdES, PAdES, JAdES and ASiC-S/ASiC-E;
- Formats of the signed documents: XML, JSON, PDF, DOC, TXT, ZIP, etc.;
- Packaging structures: enveloping, enveloped, detached and internally-detached;
- 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;
- Signature qualification;
- Validation reports (Simple, Detailed, ETSI Validation report);
- Management of signature tokens;
- Validation of the signing certificate;
- Timestamp creation;
- Timestamp validation and qualification;
- REST and SOAP webservices.

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.

The DSS framework is actively maintained and new features will be released in the future.

1.4. Available demonstrations

With the framework, some demonstrations are provided:

- DSS online demo web application;

- Ready to use demo web application build;
- JavaFX Standalone Application.

The demonstrations use a fake timestamp service (Mock) so that is not recommended for a production usage.

The requirements and build instructions for DSS demonstrations can be found in the section DSS Demonstrations.

1.5. License

For the DSS core: GNU Lesser General Public License version 2.1 (LGPL).

For the DSS demo: Dual license MOCCA EUPL and GNU Lesser General Public License version 2.1 (LGPL). For more information please see DSS demonstration LICENSE.

1.6. Abbreviations and Acronyms

Table 1. Abbreviations and Acronyms

Code	Description
AdES	Advanced Electronic Signature
API	Application Programming Interface
ASiC	Associated Signature Containers
BB	Building Block (DIGITAL)
BBB	Basic Building Block (cf. [R09])
CA	Certificate authority
CAdES	CMS Advanced Electronic Signatures
CMS	Cryptographic Message Syntax
CRL	Certificate Revocation List
CSP	Cryptographic Service Provider
DER	Distinguished Encoding Rules
DIGITAL	EC DIGITAL Building Block
DSA	Digital Signature Algorithm - an algorithm for public-key cryptography
DSS	Digital Signature Service
EC	European Commission
ESI	Electronic Signatures and Infrastructures
ETSI	European Telecommunications Standards Institute

EUPL	European Union Public License
HSM	Hardware Security Modules
HTTP	Hypertext Transfer Protocol
JAdES	JSON Advanced Electronic Signatures
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
JDBC	Java DataBase Connectivity
JWS	JSON Web Signatures
LGPL	Lesser General Public License
LOTL	List of Trusted List or List of the Lists
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
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
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	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

1.7. References

Table 2. References

Ref.	Title	Reference	Version
R01	ESI - XAdES digital signatures	ETSI EN 319 132 part 1-2	1.1.1
R02	ESI - CAdES digital signatures	ETSI EN 319 122 part 1-2	1.1.1
R03	ESI - PAdES digital signatures	ETSI EN 319 142 part 1-2	1.1.1
R04	ESI - Associated Signature Containers (ASiC)	ETSI EN 319 162 part 1-2	1.1.1
R05	ESI - JAdES digital signatures	ETSI TS 119 182 part 1	1.1.1

Ref.	Title	Reference	Version
R06	Document management - Portable document format - Part 1: PDF 1.7	ISO 32000-1	1
R07	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	
R08	Internet X.509 Public Key Infrastructure - Time-Stamp Protocol (TSP)	RFC 3161	
R09	ESI - Procedures for Creation and Validation of AdES Digital Signatures	ETSI EN 319 102-1	1.3.1
R10	ESI - Signature validation policy for European qualified electronic signatures/seals using trusted lists	ETSI TS 119 172-4	1.1.1
R11	ESI - Trusted Lists	ETSI TS 119 612	2.2.1
R12	eIDAS Regulation No 910/2014	910/2014/EU	
R13	ESI - Procedures for Creation and Validation of AdES Digital Signatures	ETSI TS 119 102-2	1.3.1
R14	ESI - Procedures for using and interpreting EU Member States national trusted lists	ETSI TS 119 615	1.1.1
R15	Internet RFC 2315 PKCS #7: Cryptographic Message Syntax	RFC 2315	Version 1.5

Ref.	Title	Reference	Version
R16	Commission implementing decision (EU) 2015/1506 of 8 September 2015	CID 2015/1506	
R17	ESI - Building blocks and table of contents for human readable signature policy documents	ETSI TS 119 172-1	1.1.1
R18	ESI - XML format for signature policies	ETSI TS 119 172-2	1.1.1
R19	ESI - ASN.1 format for signature policies	ETSI TS 119 172-3	1.1.1
R20	Internet RFC 7515: JSON Web Signature (JWS)	RFC 7515	
R21	Internet RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile	RFC 5280	
R22	Internet RFC 6960: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP	RFC 6960	
R23	COMMON PKI SPECIFICATIONS FOR INTEROPERABLE APPLICATIONS FROM T7 & TELETRUST	Common PKI v2.0	v2.0

1.8. Useful links

- Digital Building Block
- eSignature FAQ
- Trust Services Dashboard
- eSignature validation tests
- Trusted List Manager non-EU
- DSS source code (GitHub)

- DSS source code (EC Bitbucket)
- DSS-demonstrations source code (GitHub)
- DSS-demonstrations source code (EC Bitbucket)
- Report an issue (EC Jira)
- Old Jira

2. How to start with DSS

2.1. Integration instructions

The section explains the basic steps required to successfully integrate the DSS components to your project.

2.1.1. DSS Core

This section explains the usage and build requirements for DSS framework.

2.1.1.1. Requirements

The latest version of DSS framework has the following minimal requirements:

- Java 11 and higher (tested up to Java 17) for the build is required. For usage Java 8 is a minimum requirement;
- Maven 3.6 and higher;
- Memory and Disk: see minimal requirements for the used JVM. In general the higher available is better;
- Operating system: no specific requirements (tested on Windows and Linux).

We strongly recommend using the latest available version of JDK, in order to have the most recent security fixes and cryptographical algorithm updates.

Before processing the integration steps, please ensure you have successfully installed Maven and JVM with a required version.

2.1.1.2. Adding as Maven dependency

The simplest way to include DSS to your Maven project is to add a repository into the pom.xml file in the root directory of your project as following:

```
<repositories>
    ...

    <repository>
      <id>cefdigital</id>
      <name>cefdigital</name>

<url>https://ec.europa.eu/cefdigital/artifact/content/repositories/esignaturedss/</url
>
    </repository>
</repositories>
```

After that specify a list of dependencies required for your project. See Maven modules to get familiar with the available modules in DSS.

Refresh your project in order to download the dependency, and then you will be able to use all modules of the DSS framework. Your project needs to be refreshed every time you add new dependencies.

2.1.1.2.1. Integration with Bill of Materials (BOM) module

As DSS represents a multi-modules framework that benefits users from a more effective way of using the library (include only what you need), it has a downside that makes it difficult to keep versions of all modules up-to-date. The "bill of materials" (BOM) solution, represented by `dss-bom` module, helps other projects with the "version management".

The root `pom.xml` of `dss-bom` defines versions of all modules within DSS-library. Other projects that wish to benefit from the solution in DSS, should import `dss-bom` module using `dependencyManagement` and load other required modules without the need to define a version for each dependency:

```
 <dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>eu.europa.ec.joinup.sd-dss</groupId>
            <artifactId>dss-bom</artifactId>
            <version>5.10</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>eu.europa.ec.joinup.sd-dss</groupId>
        <artifactId>dss-utils-apache-commons</artifactId>
    </dependency>
    <dependency>
        <groupId>eu.europa.ec.joinup.sd-dss</groupId>
        <artifactId>dss-xades</artifactId>
    </dependency>
    ...
    <!-- add other required modules -->
</dependencies>
```

2.1.1.3. Maven build and profiles

In order to use a customized bundle of DSS, you may want to build the DSS Core framework modules.

If you have implemented a new feature or fixed a bug issue, your pull requests are welcome at our GitHub Repository

A simple build of the DSS Maven project can be done with the following command:

```
mvn clean install
```

All listed commands must be executed from the project directory via a Command Line Interface (CLI).

This installation will run all unit tests present in the modules, which can take more than one hour to do the complete build.

In addition to the general build, the framework provides a list of various profiles, allowing a customized behavior:

- **quick** - disables unit tests and java-doc validation, in order to process the build as quick as

possible (takes 1-2 minutes). **This profile cannot be used for a primary DSS build** (see below).

- **quick-init** - is similar to the `quick` profile. Disables java-doc validation for all modules and unit tests excluding some modules which have dependencies on their test classes. **Can be used for the primary build of DSS**.
- **slow-tests** - executes all tests, including time-consuming unit tests.
- **owasp** - runs validation of the project and using dependencies according to the National Vulnerability Database (NVD).
- **jdk19-plus** - executed automatically for JDK version 9 and higher. Provides a support of JDK 8 with newer versions.
- **spotless** - used to add a licence header into project files.

Some modules (e.g. `dss-utils`, `dss-crl-parser`, etc., see ch. Specific modules) have to be built completely, as other modules are dependent on their test classes. Therefore, for the first build of DSS, the profile `quick-init` should be chosen rather than `quick` profile.

In order to run a build with a specific profile, the following command must be executed:

```
mvn clean install -P *profile_name*
```

2.1.1.4. Documentation generation

In order to generate HTML and PDF documentation for the DSS project, the `dss-cookbook` module of the DSS Core must be built with the following command (please, ensure that you are located in the `/dss-cookbook` directory):

```
mvn clean install -P asciidoctor
```

2.1.1.5. Javadoc generation

In order to generate HTML Javadoc, you will need to build the DSS Core completely.

2.1.2. DSS Demonstrations

This section explains the build and use requirements for the DSS Demonstration Applications.

2.1.2.1. Requirements

The minimal requirements to build/run DSS Demonstrations:

- Java 8 and higher (tested up to Java 17) is required;
- Maven 3.6 and higher (if build required);
- Tomcat 8.5+ for Java 8 and Tomcat 9+ for Java 9 and higher (for Web-application);
- Memory and Disk: see minimal requirements for the used JVM. In general the highest available

is the best;

- Operating system: no specific requirements (tested on Windows and Linux).

2.1.2.2. Ready to use solutions

2.1.2.2.1. DSS Web Application

The ready to use webapp allows testing the different functionalities offered in DSS without needing to dive into the implementation.

The DSS demo is available online on the DIGITAL website.

The DSS demo is also available as a ready to use downloadable webapp. To use it, you need to complete the following steps:

1. Download the webapp as a ZIP folder.
2. Unzip the folder
3. Click on the Webapp-Startup.bat file
4. Wait until this message appears "Server startup in xxx ms"
5. Click on the DSS-Web internet shortcut

2.1.2.2.2. DSS Standalone Application

DSS provides a standalone application which uses JavaFX. The application does not require a server to publish the product. The application can be run locally on a client's machine.

Download links for the Standalone Application (Windows x64):

- Minimal ZIP (application + bat file);
- Complete ZIP (application + bat file + OpenJDK + JavaFX SDK).

2.1.2.3. Maven build instructions

The build of the project can be done similarly to the DSS Core framework build with the command `mvn clean install`.

Please ensure that you build modules that you really need. Ignore build failures for non-required modules.

2.1.2.3.1. DSS Web Application build

To build the DSS Web Application the following modules are required:

- `dss-mock-tsa`;
- `dss-demo-webapp`;
- `dss-demo-bundle`.

After a successful build, in the directory `/dss-demo-bundle/target/` you will be able to find two

containers: dss-demo-bundle.zip and dss-demo-bundle.tar.gz. Despite the different container type, the content of both containers is the same. After extracting the content, you will need to run the file Webapp-Startup.bat in order to launch the server and the file Webapp-Shutdown.bat to stop the server. After running the server, the web-application will be available at the address http://localhost:8080/.

If during TL/LOTL loading you experience problems with some particular Trusted Lists, please refer the Java Keystore Management chapter for a resolution.

The documentation and javadoc will be copied automatically from the built DSS Core and made available on the following addresses respectively:

- HTML documentation : http://localhost:8080/doc/dss-documentation.html;
- PDF documentation : http://localhost:8080/doc/dss-documentation.pdf;
- Javadoc : http://localhost:8080/apidocs/index.html.

In order to build a bundle for JDK 17, the following profile can be used from the dss-demo-bundle module:

```
mvn clean install -P java17
```

This will create a bundle with Tomcat 9.

2.1.2.3.2. Integration tests

The dss-demo-webapp module provides a collection of integration tests in order to test the behavior of REST/SOAP web-services. In order to run the tests, a web-server with the DSS Web Application shall be launched and the following profile needs to be executed from the module:

```
mvn clean install -P run-integration-test
```

2.1.2.3.3. DSS Standalone Application build

In order to build the standalone application, the following modules are required:

- dss-mock-tsa;
- dss-standalone-app;
- dss-standalone-package.

If the build is successful, you will be able to find out the following containers in the directory /dss-standalone-app-package/target/:

- dss-standalone-app-package-minimal.zip - contains the application code. Requires JDK ad JavaFX installed on a target machine in order to run the application;
- dss-standalone-app-package-complete.zip - contains the application code, as well as JDK and JavaFX library code. Can be run on a machine without pre-installed libraries.

In order to launch the application, you will need to extract the archive and run the file `dss-run.bat`.

2.2. DSS framework structure

DSS framework is a Maven multi-module project. See below the specifications about provided modules within the DSS core.

2.2.1. Maven modules

This chapter provides an overview on modules available within Source code of DSS Core.

2.2.1.1. Shared modules

dss-enumerations

Contains a list of all used enumerations in the DSS project.

dss-alerts

Allows configuration of triggers and handers for arbitrary defined events.

dss-jaxb-parsers

Contains a list of all classes used to transform JAXB objects/strings to Java objects and vice versa.

2.2.1.2. JAXB model modules

specs-xmldsig

W3C XSD schema for signatures http://www.w3.org/2000/09/xmldsig

specs-xades

ETSI EN 319 132-1 XSD schema for XAdES.

specs-trusted-list

ETSI TS 119 612 XSD schema for parsing Trusted Lists.

specs-validation-report

ETSI TS 119 102-2 XSD schema for the Validation report.

specs-asic-manifest

ETSI EN 319 162 schema for ASiCManifest.

specs-saml-assertion

OASIS schema for SAML Assertions.

---

dss-policy-jaxb

JAXB model of the validation policy.

dss-diagnostic-jaxb

JAXB model of the diagnostic data.

dss-detailed-report-jaxb

JAXB model of the detailed report.

dss-simple-report-jaxb

JAXB model of the simple report.

dss-simple-certificate-report-jaxb

JAXB model of the simple report for certificates.

2.2.1.3. JSON validation modules

specs-jws

JSON Schemas based on the RFC 7515 specifications ([R05]).

specs-jades

ETSI TS 119 182-1 JSON Schemas for JAdES ([R20]).

2.2.1.4. Utils modules

dss-utils

API with utility methods for String, Collection, I/O,…

dss-utils-apache-commons

Implementation of dss-utils with Apache Commons libraries.

dss-utils-google-guava

Implementation of dss-utils with Google Guava.

2.2.1.5. i18n

dss-i18n

a module allowing internationalization of generated reports.

2.2.1.6. Core modules

dss-model

Data model used in almost every module.

dss-crl-parser

API to validate CRLs and retrieve revocation data

dss-crl-parser-stream

Implementation of dss-crl-parser which streams the CRL.

dss-crl-parser-x509crl

Implementation of dss-crl-parser which uses the java object X509CRL.

dss-spi

Interfaces and util classes to process ASN.1 structure, compute digests, etc.

dss-document

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

dss-service

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

dss-token

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

validation-policy

Business of the signature's validation (ETSI EN 319 102 / TS 119 172-4).

dss-xades

Implementation of the XAdES signature, augmentation and validation.

dss-cades

Implementation of the CAdES signature, augmentation and validation.

dss-jades

Implementation of the JAdES signature, augmentation and validation.

dss-pades

Common code which is shared between dss-pades-pdfbox and dss-pades-openpdf.

dss-pades-pdfbox

Implementation of the PAdES signature, augmentation and validation with PDFBox.

dss-pades-openpdf

Implementation of the PAdES signature, augmentation and validation with OpenPDF (fork of iText).

dss-asic-common

Common code which is shared between dss-asic-xades and dss-asic-cades.

dss-asic-cades

Implementation of the ASiC-S and ASiC-E signature, augmentation and validation based on CAdES signatures.

dss-asic-xades

Implementation of the ASiC-S and ASiC-E signature, augmentation and validation based on XAdES signatures.

dss-tsl-validation

Module which allows loading / parsing / validating of LOTL and TSLs.

2.2.1.7. WebServices

dss-common-remote-dto

Common classes between all remote services (REST and SOAP).

dss-common-remote-converter

Classes which convert the DTO to DSS Objects.

---

dss-signature-dto

Data Transfer Objects used for signature creation/augmentation (REST and SOAP).

dss-signature-remote

Common classes between dss-signature-rest and dss-signature-soap.

dss-signature-rest-client

Client for the REST webservices.

dss-signature-rest

REST webservices to sign (getDataToSign, signDocument methods), counter-sign and augment a signature.

dss-signature-soap-client

Client for the SOAP webservices.

dss-signature-soap

SOAP webservices to sign (getDataToSign, signDocument methods), counter-sign and augment a signature.

---

dss-server-signing-dto

Data Transfer Objects used for the server signing module (REST and SOAP).

dss-server-signing-common

Common classes for server signing.

dss-server-signing-rest

REST webservice for server signing.

dss-server-signing-rest-client

REST client for server signing (sign method).

dss-server-signing-soap

SOAP webservice for server signing.

dss-server-signing-soap-client

SOAP client for server signing (sign method).

dss-validation-dto

Data Transfer Objects used for signature validation (REST and SOAP).

dss-validation-common

Common classes between dss-validation-rest and dss-validation-soap.

dss-validation-rest-client

Client for the REST signature-validation webservices.

dss-validation-soap-client

Client for the SOAP signature-validation webservices.

dss-validation-rest

REST webservices to validate a signature.

dss-validation-soap

SOAP webservices to validate a signature.

---

dss-certificate-validation-dto

Data Transfer Objects used for certificate validation (REST and SOAP).

dss-certificate-validation-common

Common classes between dss-certificate-validation-rest and dss-certificate-validation-soap.

dss-certificate-validation-rest-client

Client for the REST certificate-validation webservice.

dss-certificate-validation-soap-client

Client for the SOAP certificate-validation webservice.

dss-certificate-validation-rest

REST webservice to validate a certificate.

dss-certificate-validation-soap

SOAP webservice to validate a certificate.

---

dss-timestamp-dto

Data Transfer Objects used for timestamp creation.

dss-timestamp-remote-common

Common classes between dss-timestamp-remote-rest and dss-timestamp-remote-soap.

dss-timestamp-remote-rest-client

Client for the REST timestamp webservice.

dss-timestamp-remote-soap-client

Client for the SOAP timestamp webservice.

dss-timestamp-remote-rest

REST webservice to create a timestamp.

dss-timestamp-remote-soap

SOAP webservice to create a timestamp.

2.2.1.8. Other modules

dss-test

Mock and util classes for unit tests.

dss-cookbook

Samples and documentation of DSS used to generate this documentation.

dss-jacoco-coverage

Module which is used to collect a test coverage for all modules.

dss-bom

Module which helps the integration with all DSS modules and the version.

2.2.2. Specific modules

Some modules of the DSS framework have a specific behavior and has to be handled accordingly.

DSS contains a bundle of JAXB-based modules, generating Java classes at runtime based on XSD-schema. When any change is made in the XSD, the classes of the module are being re-generated according to the change. The following modules present this behavior:

- specs-xmldsig;
- specs-xades;
- specs-trusted-list;
- specs-validation-report;
- specs-asic-manifest;
- specs-saml-assertion;
- dss-policy-jaxb;
- dss-diagnostic-jaxb;
- dss-detailed-report-jaxb;
- dss-simple-report-jaxb;
- dss-simple-certificate-report-jaxb.

Specific modules with JWS and JAdES specifications exist. These modules allow to validate the generated JSON against the related JSON Schema :

- specs-jws;
- specs-jades.

Also, as it was explained in the previous section, some modules are required to be built completely in order for their dependent modules to be built when using a quick profile, namely:

- dss-utils;
- dss-crl-parser;
- dss-test;
- dss-pades;
- dss-asic-common.

The modules contain common interfaces, used in other DSS modules, as well as unit tests to ensure the same behavior between their implementations.

2.2.3. DSS-demonstration modules

This chapter provides an overview on modules available within demonstrations project.

dss-mock-tsa	Timestamping source which generates fake timestamps from a self-signed certificate.
sscd-mocca-adapter	Adapter for the MOCCA connection.
dss-standalone-app	Standalone application which allows signing 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 basic DSS functionalities.
dss-demo-bundle	Packaging module for dss-demo-webapp.

3. Electronic signatures and DSS

3.1. EU legislation

In the European Union the following legislation have had a considerable impact on the topic of electronic and digital signatures:

- the Directive 1999/93/EC of the European Parliament and of the Council of 13 December 1999 on a Community framework for electronic signatures (cf. [R07]);
- Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on

electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC (cf. [R12]).

The eIDAS Regulation repealed Directive 1999 and became official on July 1, 2016. A Regulation is a law that applies across all EU Member States (MS). eIDAS aims for interoperability between the EU MS, among others in the field of the electronic signature, by building compatible trust service frameworks.

One of the main aspects of the eIDAS Regulation, is that where the Directive mainly covered Certificate Service Providers, the eIDAS Regulation expands on that concept and introduces the new concepts of trust services and trust service providers which is detailed in the next subsection.

3.1.1. Trust Service Provider

A Trust Service Provider (TSP) is a natural or legal person who provides one or more trust services. A trust service is an electronic service related, among others, to the creation, validation and preservation of electronic signatures, timestamps, and certificates.

Given that a TSP can provide a combination of trust services, a TSP can take one or more of the following roles

- a certificate issuer (CA);
- a time-stamp issuer (TSA);
- a signature verifier (VA);
- …

A TSP can be either a qualified or non-qualified trust service provider. All TSPs no matter if qualified or not have the following obligations and requirements

- Processing of personal data;
- Notification of security and personal data breaches;
- Keeping an up-to-date termination plan;
- Meeting requirements on employed staff and subcontractors (e.g. trainings);
- Keeping sufficient financial resources and/or liability insurance;
- Recording and keeping activities related to data accessible;
- …

This ensures the validity and security of the trust services that TSPs provide, such as the integrity of the data that was used for certificate and signature creation as well as the security of the signing keys.

A qualified trust service provider (QTSP) is a TSP that provides one or more qualified trust services and is included in a Trusted List (cf. Trusted Lists).

Some aspects are specific to QTSPs and follow from the requirements of eIDAS

- Undergoing a pre-authorization scheme;

- Being actively supervised;
- Undergoing regular audits;
- Presumption of intention or negligence in case of damage due to failure to comply to the law;
- Providing a high level of security;
- Providing legal certainty;
- Presumption of the integrity of the data;
- ...

3.2. Electronic and digital signatures

The terms “Electronic Signature” and “Digital Signature” are often used interchangeably however they are very disctinct concetps as "electronic signature" is a legal concept, whereas "digital signature" is a technical concept that is used to provide a concrete instance of electronic signatures.

In the eIDAS Regulation, and electronic signature is defined (legally) as "data in electronic form which is attached to or logically associated with other data in electronic form and which is used by the signatory to sign".

An electronic signature does not necessarily guarantee that the signature process is secure nor that it is possible to track the changes that have been brought to the content of a document after it was signed. This depends on the category of the electronic signature. Indeed, beyond the concept of "simple" electronic signatures (SES) the Regulation further defines Advanced Electronic Signatures (AdES) and Qualified Electronic Signatures (QES).

A **Simple Electronic Signature** can cover a very broad range of data, such as a name written at the end of an email or an image added to a document.

An **Advanced Electronic Signature** is an electronic signature that has the following properties:

- It is uniquely linked to the signatory.
- It is capable of identifying the signatory.
- The signatory has the sole control over the data used for the creating signatures.
- It can detect whether the signed data has been modified since the signature.

A **Qualified Electronic Signature** is an AdES that is based on a qualified certificate for electronic signatures (cf. Digital certificate) and that has been generated by a qualified signature creation device (QSCD). QES have the same legal value as handwritten signatures. When an electronic signature is a QES, there is a reversal of the burden of proof. There is a presumption that a person has signed until a proof is given that the person did not sign.

A **digital signature** is a technical concept that is based on a Public Key Infrastructure (PKI, cf.Simplified PKI model) and involves, among others, public key cryptography and public key certificates (cf. Digital certificate).

Digital signatures can be used to ensure the unique identification of the signer, the authenticity of the signature and the integrity of the data. The identification of the signer as well as the

authenticity of the signature are guaranteed by decrypting the digital signature value using a public key attested by a public key certificate (cf. Digital certificate). The component of the digital signature that allows detecting whether signed data has been tampered with is a cryptographic function called a hash function.

"AdES digital signatures" are digital signature formats that have been developped by ETSI to support the eIDAS Regulation and provide a way to create digital signatures that can meet the legal requirements for AdES and QES.

3.3. Digital signatures concepts

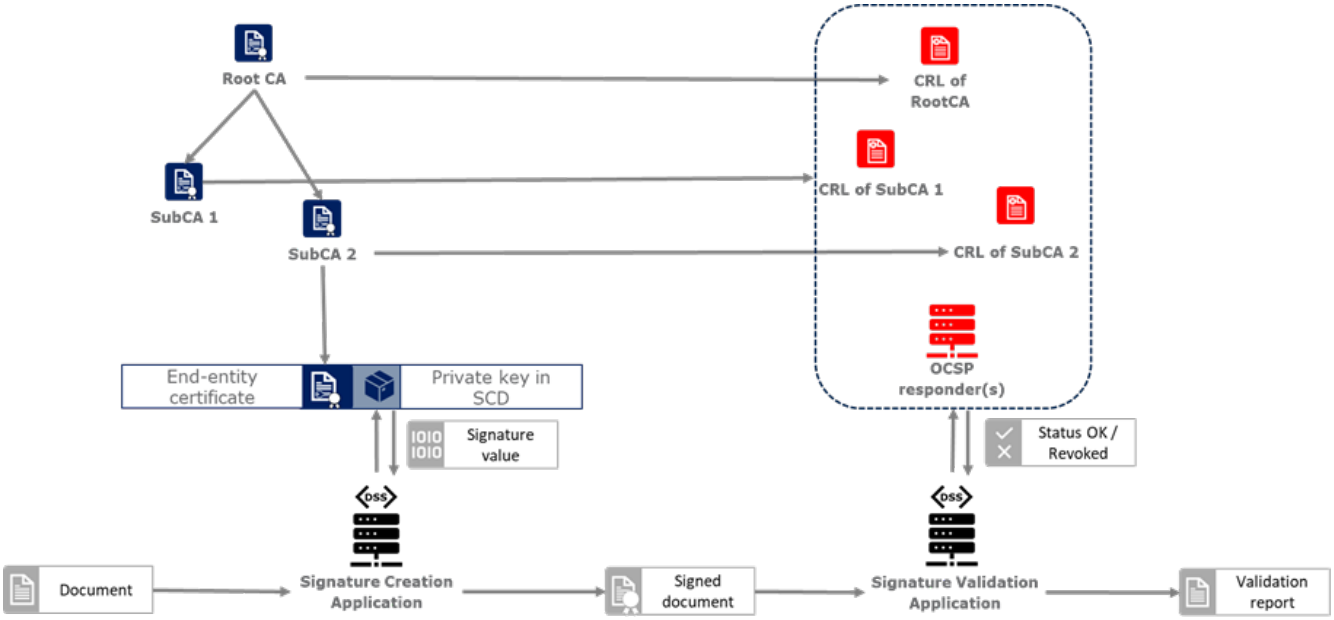
This section aims to briefly introduce PKI-based digital signature concepts, more specifically concepts related to digital signatures supported by X.509 digital certificates issued by Certification Authorities (CA), and making use of asymmetric cryptography. Such signatures are the kind of signatures that are handled in DSS.

For the rest of this section, the creation of a digital signature value is assumed to be the encryption of the digest of a data object using a private key for which there exists a corresponding X.509 public key certificate issued by a CA.

For the purpose of introducing those concepts, we will first provide a simplified description of the PKI model in which digital signatures are created. The goal of this model is not to provide an accurate and exhaustive description and definition of a PKI but to provide a basis for introducing the main PKI concepts that are useful to DSS users. Suggestions for improvement are welcomed and can be proposed via PRs in the DSS github.

3.3.1. Simplified PKI model

A (simplified) description of the PKI model and where DSS is involved in that model is given in the figure below.



In this simplified model, a PKI is composed of:

- **Certificates**;
- **Certification Authorities (CA)** issuing the certificates;
- **Certificate Revocation Lists (CRL)** issued by CAs; and
- **OCSP responders** providing information on the status of certificates.

In turn, DSS within that model, can be used to implement Signature creation applications (SCA) and/or Signature Validation Applications (SVA)

Each of those concepts are further detailed in the next sections.

3.3.2. Digital certificate

As mentioned before, in the present context, digital signatures are supported by public key certificates. **Public key certificates** are data structures that binds an entity to a public key and that are signed by a third party, they provide a proof of authenticity of the public key.

The ITU-T X.509 Recommendation is a standard describing (among others) such a data structure, and public key certificates structured as per the specifications provided in that standard are commonly referred to as “X.509 public key certificates”.

Furthermore, the IETF published the RFC 5280 ([R21]) which specifies a profile for X.509 public key certificates (and certificate revocation lists). For the remainder of this document, X.509 public key certificates are assumed to be profiled as per RFC 5280.

Certificates can be end-entity certificates or CA certificates:

- **End-entity certificates** are certificates issued to entities that are not authorized to issue certificates, for instance a natural person;
- **CA certificates** are certificates issued to entities authorized to issue certificates, also known as Certification Authorities (CA).

Certificates have a defined validity period during which the CA having issued the certificate guarantees the correctness of its content. During that validity period, they may however be revoked or suspended, for instance when the entity to which the certificate has been issued has lost control of the corresponding private key.

A certificate contains among other things information on:

- The entity to which the certificate has been issued, also referred to as the Subject;
- The public key which is bound to the Subject;
- The entity having issued the certificate (the CA), also referred to as the Issuer;
- The validity period of the certificate;
- The location where information on the revocation status of the certificate can be found;
- Restriction applying to the usage of the public key contained in the certificate;
- A digital signature created by the issuer of the certificate;
- …

3.3.3. CRLs and OCSP

As previously mentionned, a certificate can be revoked or suspended. This information is usually provided in the form of a Certificate Revocation List (CRL), or through the Online Certificate Status Protocol (OCSP).

A CRL is a list of revoked (and/or suspended) certificates that is digitally signed and published by a CRL issuer. This issuer can be the CA having issued the certificates listed in the CRL, or it can be another CA in which case the CRL is called an “indirect CRL”. RFC 5280 ([R21]) provides a profile for X.509 CRLs.

The OCSP is a protocol defined in RFC 6960 ([R22]) that enables the determination of the (revocation) status of a certificate without the use of a CRL. An OCSP request, containing (among other things) information on the certificate for which the (revocation) status is requested, is sent to a server and a response, containing information of that (revocation) status, is provided by an OCSP responder. OCSP responses are signed by the OCSP responder, and the OCSP responder can be the CA having issued the certificate or another CA in which case the OCSP responder is called a “delegated OCSP responder”.

RFC 5280 section 6.3 describes an algorithm for the validation of CRLs, while Common PKI v2.0 part 5 section 2.3 ([R23]) describes an algorithm for checking the revocation status of a certificate using CRLs and OCSP responses.

3.3.3.1. Certificate Authority

Certification Authorities are entities issuing certificates and guaranteeing the correctness of their content. They manage the whole lifecycle of the certificates they issue, including the revocation services. Throughout this document, they will be denominated as:

- Issuing CA for the CAs that issue end-entity certificates:
- Intermediate CA for CAs that issue certificates to other CAs and are not root CAs;
- Root CA for the CAs that have at least one self-signed certificate.

3.3.3.2. Trust Anchors and Trust Stores

Without going into the details and inner workings of the hierarchical trust model (this document does not intend to discuss the soundness of this model, the soundness of transitivity of trust, etc.), when a user is looking to validate a certificate, that is the user needs to decide whether or not it can trust the binding between the public key and the subject of that certificate, it will make use of so called “trust anchors”.

A trust anchor, in the context of certificate validation, is a CA that is trusted by the user in such a way that if there exists a valid chain of certificate from that CA to a certificate, the user trusts the correctness of the information contained in that certificate taking into consideration the (revocation) status of that certificate.

The wording “valid chain of certificate” used above is voluntarily informal, but it can be more formally defined as meaning that there exists a prospective certification path such that the output of the certification validation path algorithm (see Certificate Chain and Certification Path

Validation) provided with, as inputs, that prospective certification path, the trust anchor information and possibly other inputs, is a success indication.

Trust anchor information can be, and is often, provided as a (potentially self-signed) public key certificate.

A trust store is, in turn, a list of trust anchor information that can be, and is often, a list of directly trusted public key certificates.

3.3.4. Trusted List (TL)

3.3.4.1. EU MS Trusted List

Trusted lists, as they are used in the EU/EEA, are a legal instrument used to provide, among other things, information on the qualified status of trust services.

Technically, they take the form of an XML structure formatted as specified in the standard ETSI TS 119 612 ([R11]).

Trusted lists can be used in a similar way to trust stores in that one can use, for instance, the public key certificates that are listed as the digital identity of qualified trust services issuing qualified certificates as trust anchors for the purpose of validating certificates, however there are significant differences between the usage of trusted lists and the usage of classic trust stores. Below is a non-exhaustive list of such differences:

- Trusted lists can be used to determine/confirm the legal type of a certificate i.e. verifying that a certificate is a certificate for electronic signature, for electronic seal or for website authentication, whereas trust store typically do not allow such determination.
- Trusted list can be used to determine/confirm the qualified status of a certificate;
- Trusted lists contain the status history of trust services, meaning that they allow the determination/confirmation of whether a certificate was qualified and of a particular type at a time in the past. Trust service entries are never removed from a trusted list whereas compromise of a trust anchor is usually reflected by the removal of the corresponding trust anchor information from a trust store (in a trusted list, this would be reflected by changing the current status of the corresponding trust service, while keeping the status history);
- Trusted lists frequently (one might argue ‘mostly’) identify trust services issuing certificates through the certificates of issuing CAs, whereas trust store usually contain mostly root CAs.

3.3.4.2. List of Trusted Lists (LOTL)

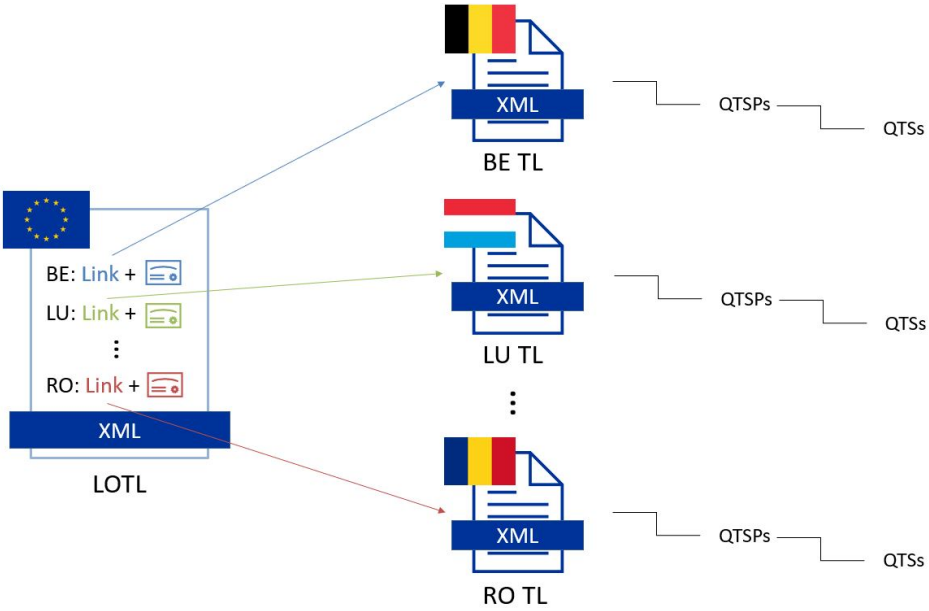
A List of Trusted Lists (LOTL) is a list that contains:

- links towards all the published EU MS Trusted Lists;
- the certificates used to verify the signatures of these trusted lists.

In the EU/EEA context, a LOTL is published by the European Commission at a secure location that is made publicly available on the Official Journal of the European Commission (OJEU). It is available in an XML format which is suitable for automated processing. This format of the LOTL is digitally

signed/sealed, which allows to assure authenticity and integrity of the LOTL. The signing certificates of the LOTL are also made publicly available in the OJEU.

The LOTL is used to authenticate EU MS Trusted Lists and to provide an easy and trustworthy way to access these TLs.



When the LOTL-signing certificates or the location of the LOTL changes, the modification needs to be published by the Commission. The update is done in the form of a “pivot LOTL”, which is a specific instance of a LOTL. Each new modification will create a new pivot LOTL. The pivot LOTLs are grouped in the current LOTL itself, under the < SchemeInformationURI> field. Consulting all the pivot LOTL from the most recent to the oldest gives a trace of all the signing certificates and locations of the LOTL back to the initial ones.

3.3.5. Certificate Chain and Certification Path Validation

The certificate path validation is an algorithm that seeks to verify the binding between the public key and the subject of a certificate, using trust anchor information. The complete processing is described in RFC 5280 section 6.1, and as stated there, it verifies among other things that a prospective certification path (a sequence of n certificates) satisfies the following conditions:

a. for all x in {1, ..., n-1}, the subject of certificate x is the issuer of certificate x+1;

b. certificate 1 is issued by the trust anchor;

c. certificate n is the certificate to be validated (i.e., the target certificate); and

d. for all x in {1, ..., n}, the certificate was valid at the time in question.

Although RFC 5280 states that procedures performed to obtain the sequence of certificate that is provided to the certification path validation is outside its scope, Common PKI v2.0 part 5 section 2.1 ([R23]) provides one such possible procedure.

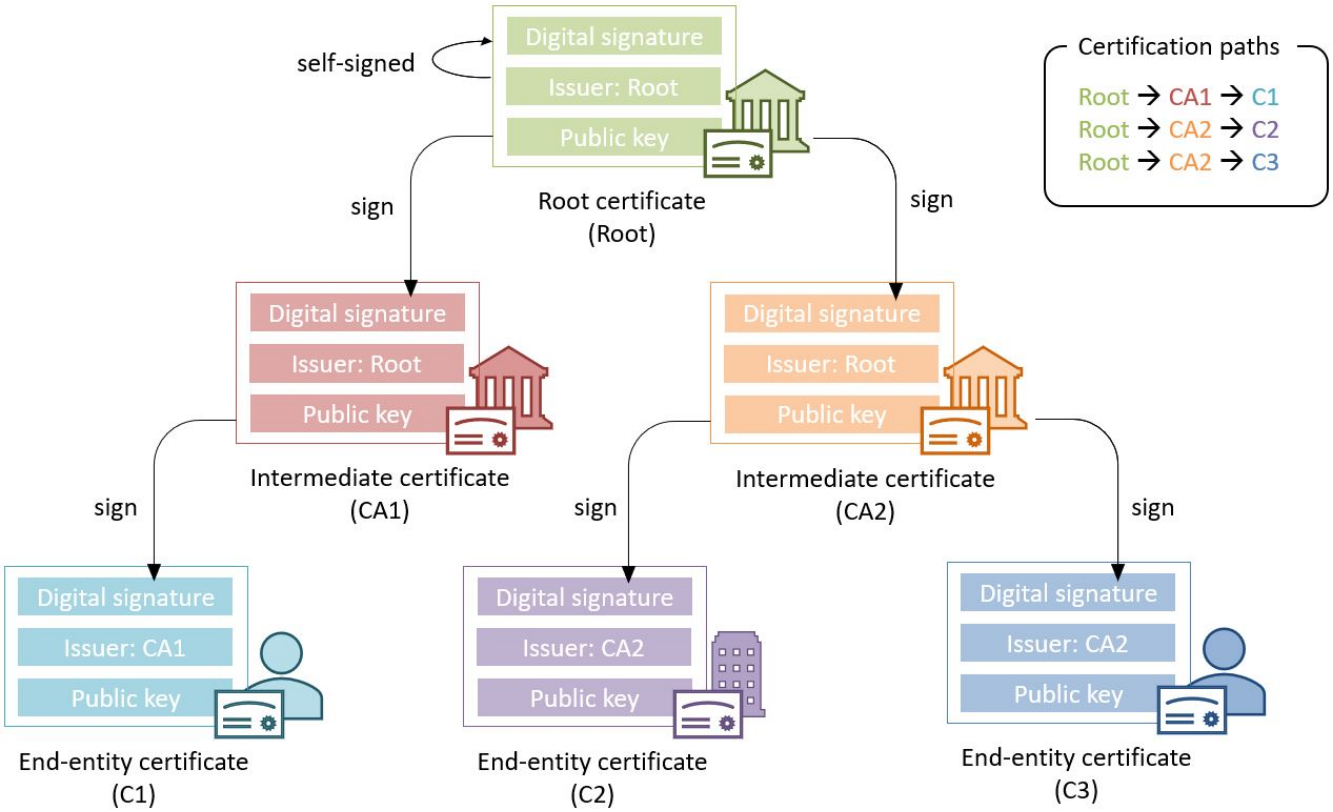
An intuitive approach to build a prospective certification path is to start by looking at the “Authority Information Access” (AIA) extension of the target certificate (see RFC 5280 section 4.2.2.1) which, if present, frequently includes information on how to retrieve the certificate of the issuer of that certificate. Repeating this action on the certificate retrieved can then allow to build a

prospective certification path.

The wording "certificate chain" is often used interchangeably with "certification path".

In ETSI EN 319 102-1 ([R09]) however, a prospective certificate chain is defined as a sequence of certificate that satisfies the conditions a. to c. above and for which the trust anchor is trusted according the validation policy in use.

An illustration of different certificate chains/certification paths is provided in the figure below.



3.3.6. Signature creation

3.3.6.1. Signature creation process

Although other schemes exist, we assume here that creating a digital signature value consists in the encryption of a hash computed on the signed data.

The standard ETSI EN 319 102-1 clause 4 ([R09]) provides a complete conceptual model for the creation of “AdES digital signatures”, but for the sake of simplicity we can extract from this model the following steps:

- Receiving a (set of) document(s) or a (set of) hash(es) representing those documents, together with other inputs (such as so-called “signed attribute” values e.g. signer’s location, and constraints driving the creation of the signature such as the cryptographic algorithms to be used for the creation of the signature value);
- Composing the “data to be signed” (DTBS) which is the data object that will be covered by the signature value (including thus the document(s) and attributes to be signed), and the associated “data to be signed formatted” (DTBSF) which can be taken as the format-specific byte-stream on which the signature value will be computed;

- Creating the “data to be signed representation” (DTBSR) by applying the appropriate hash algorithm on the DTBSF obtained in the previous step;
- Computing the signature value by encrypting the DTBSR using the appropriate algorithm (this is usually done by activating the private key within a “Signature creation device” (SCDev), that will perform the operation);
- Formatting the result into a “signed data object” (SDO) complying with the desired signature format (e.g. XAdES, PAdES, etc).

As mentionned above, the activation of the private key and the operation of creating the signature value is assumed to be performed by a specific device. It is in general desirable that this device is a secure (e.g. temper proof) device that require authentication for the activation of the key (e.g. using PIN codes).

When the private key contained in that device is controlled by an end-entity, this device is usually called “signature creation device” or **SCDev**. This can be a local SCDev such as a smartcard, but it can also be a remote SCDev managed by a CA or TSP.

When the private key is used by a CA for signing certificates, this device is usually called a “hardware security module” or **HSM**.

Frequently, when the private key is under the control of a legal entity (such as when the key is used to create electronic seals) the device is also called an HSM.

3.3.7. Signature validation (introduction)

Taking a very (or over) simplified model, validating a digital signature can be seen as:

- On one hand, verifying the cryptographic validity of the digital signature value (part of it consisting in decrypting the digital signature value and comparing the decrypted value with the hash of the signed data).
- On the other hand, verifying the validity of the signing certificate (see certification path validation).

We’ll see that even such a simplified model is useful for the purpose of introducing common concepts in digital signature validation.

Let’s imagine that we want to validate a digital signature and the time when this validation occur is denoted as T_{val}.

If the signing certificate successfully passes the certification path validation at T_{val}, and the digital signature value is cryptographically valid, one can then say that the digital signature is valid at T_{val}.

Now, if computing the hash of the signed data does not yield the same value as the decryption of the signature value, one can then say that the digital signature is invalid.

Beyond valid and invalid digital signature however, there are a lot of cases when one cannot determine the validity of a digital signature. Below are some examples where one cannot conclude that a digital signature is valid or invalid, in which case the validity status of the signature is indeterminate.

Let's imagine that at T_{val}, when we are trying to access the certification status information, that information is unavailable (e.g. the CRL cannot be downloaded, the OCSP responder is unavailable). Then it is not possible, at T_{val}, to determine whether the signing certificate is valid or not because at that time we are lacking information to conclude on that validity status. Because the validity of the signing certificate cannot be determined, the validity of the overall signature cannot be determined either and the validity of the signature is indeterminate. However, this status is only indeterminate because we do not have the information that would allow us to conclude, retrying to validate the signature with more information (e.g. at a time when the CRLs can be downloaded) could result in a definite valid or invalid status.

A more complex example is when, at T_{val}, revocation information indicates that the signing certificate is revoked since a time indicated as T_{rev} (which is thus < T_{val}).

Then at T_{val}, we can only conclude that the signing certificate is revoked and thus the signature cannot be determined as valid at T_{val}. However, this does not mean necessarily that the signature was created when the signing certificate was revoked, it may very well be that the signature was created at a time prior to T_{rev} and that, should we have validated the signature at that time, the validation would have been successful. Therefore we cannot conclude that the signature is invalid because we do not know in a definite manner if the signature was created before the revocation of the signing certificate.

For instance, if we had a proof that the signature existed before T_{rev}, such as a signature timestamp indicating a time $T_{poe} < T_{rev}$, then using that proof of existence (POE) we can conclude that the signature was created before the signing certificate was revoked and this could allow us to produce a definite conclusion.

On the other hand, if we had a proof that the signature could not have existed before T_{rev}, such as a content timestamp indicating a time $T_{cnt} > T_{rev}$ (a content timestamp is necessarily created before the digital signature value), then we could definitely conclude that the signing certificate was revoked when the digital signature was created and thus that the digital signature is invalid.

Another issue that can be illustrated here is when one creates a digital signature using cryptographic algorithms that are not considered secure: In such a case, it may be possible for an malicious actor to create counterfeited signed documents.

When validating a signature, it is therefore necessary to verify that the signature was created using cryptographic algorithms and parameters that are considered as secure. This is usually done by comparing a POE of the digital signature value with a sunset date for the cryptographic algorithms and parameters involved. A sunset date for a cryptographic algorithm and/or parameter is called a cryptographic constraint, and the application validating the signature usually keeps a set of such dates and cryptographic algorithms and parameters; this set is what is called the set of cryptographic constraints.

In general, the validation of a signature is made against a set of constraints, which the cryptographic constraints are a part of, that is also sometimes referred to as a signature validation policy.

The standard ETSI EN 319 102-1 specifies a complete validation model and procedures for the validation of "AdES digital signatures", which are implemented in DSS. The result of a validation

process performed according to those procedures is a validation report and an indication which can be:

- `TOTAL-PASSED` indicating that the signature has passed verification and it complies with the signature validation policy.
- `INDETERMINATE` indicating that the format and digital signature verifications have not failed but there is insufficient information to determine if the electronic signature is valid.
- `TOTAL_FAILED` indicating that either the signature format is incorrect or that the digital signature value fails the verification.

For each of the validation checks/constraint (e.g. signature format, signing certificate validity), 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. This allows the user to determine whether the signature validation has succeeded or not and it helps him find out why.

The following table presents the indications and sub-indications that can be encountered at completion of a signature validation process. For a detailed description of their meaning, refer to ETSI EN 319 102-1 ([R09]).

Table 3. Signature validation indications and sub-indications

Indication	Sub-indication
TOTAL-PASSED	-
TOTAL-FAILED	FORMAT_FAILURE
	HASH_FAILURE
	SIG_CRYPTO_FAILURE
	REVOKED
	EXPIRED
	NOT_YET_VALID

Indication	Sub-indication
INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	CHAIN_CONSTRAINTS_FAILURE
	CERTIFICATE_CHAIN_GENERAL_FAILURE
	CRYPTO_CONSTRAINTS_FAILURE
	POLICY_PROCESSING_ERROR
	SIGNATURE_POLICY_NOT_AVAILABLE
	TIMESTAMP_ORDER_FAILURE
	NO_SIGNING_CERTIFICATE_FOUND
	NO_CERTIFICATE_CHAIN_FOUND
	REVOKED_NO_POE
	REVOKED_CA_NO_POE
	OUT_OF_BOUNDS_NOT_REVOKED
	OUT_OF_BOUNDS_NO_POE
	REVOCATION_OUT_OF_BOUNDS_NO_POE
	CRYPTO_CONSTRAINTS_FAILURE_NO_POE
	NO_POE
	TRY_LATER
	SIGNED_DATA_NOT_FOUND
	CUSTOM

3.3.8. Timestamping

As illustrated in Signature validation (introduction), validating a signature sometimes require a proof of existence of that signature at a given time.

Such proof of existence can be given in the form of a **timestamp**.

A digital timestamp is an assertion of proof that a data object existed at particular time. This usually takes the form of a binding between a hash of a data object and a date and time issued and signed by a trustworthy timestamping authority.

When signing digitally, a date and time can be already included into the signature, but it corresponds to the signer computer's local time. The latter can easily be modified prior to signing so that the time of signing is not the actual one. Thus, this signing time cannot be trusted. A trustworthy digital timestamp shall be used to prove existence of the signature (and its associated data) at a certain point in time.

This principle exists for handwritten signatures too. When a document is signed manually, it is done in the presence of a trustworthy notary, who verifies not only the identity of the signer but also the date and time of the signature.

Before explaining the timestamping process, let us define some concepts that are involved in this process

- A Timestamp Authority (TSA) is a Trust Service Provider (cf. Trust Service Provider) that creates timestamp tokens using one or more Timestamping Units. The TSA must comply with the IETF RFC 3161 specifications (cf. [R08]).
- A Timestamping Unit (TU) is a set of hardware and software that contains a single signing key used by a TSA.

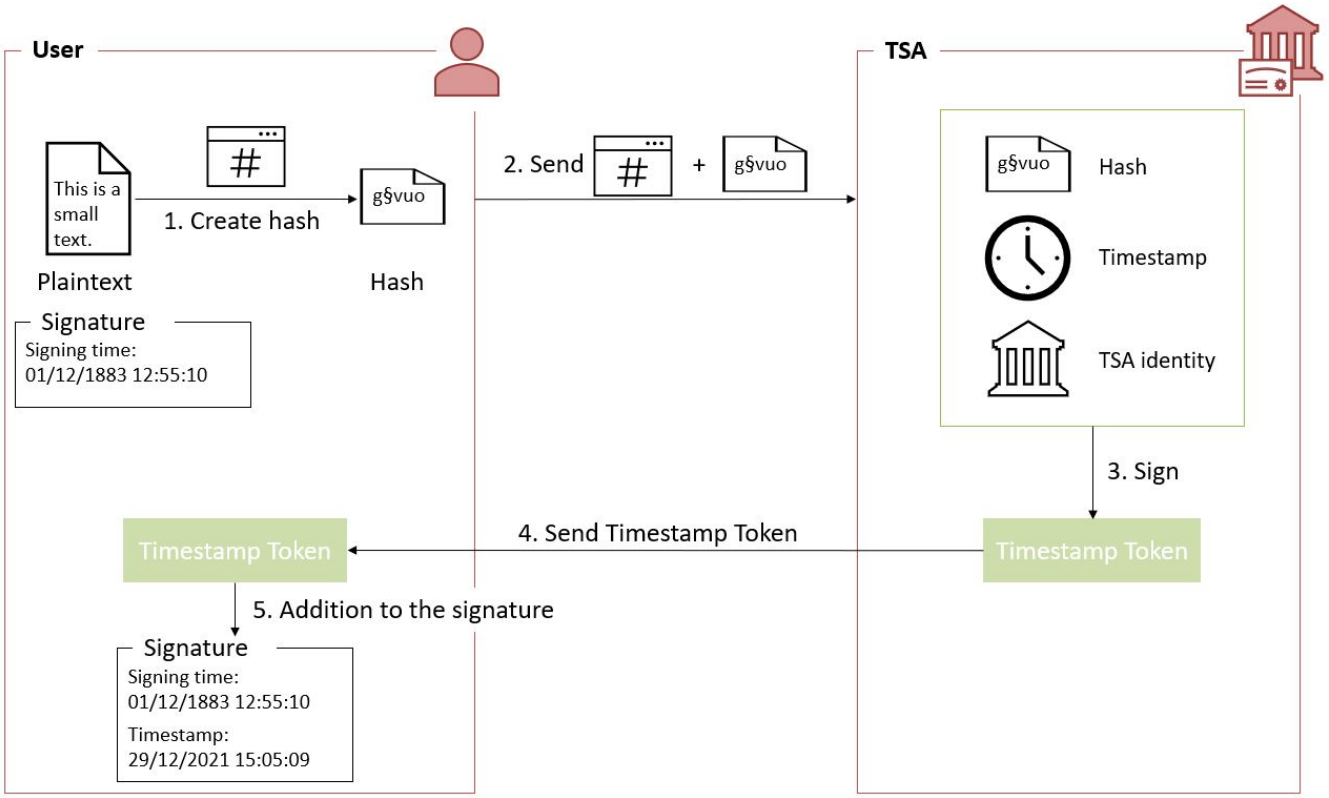
Furthermore, in the context of digital signatures, we usually distinguish timestamps depending on the data for which they provide a proof of existence:

- A content timestamp is a timestamp that is computed on the original data that is signed by a signature. It provides a proof of existence of the original data but not of the signature.
- A signature timestamp is a timestamp that is computed on the digital signature value (in some case on the whole signed data object). It provides a proof of existence of the signature value.
- An archive timestamp is a timestamp that is computed on the validation material of a signature (that is, the data necessary to validate a signature such as CRLs, OCSP responses, certificate chain, etc). They at least provide a proof of existence of that validation material, but as they are frequently in fact computed on the whole signed data object in which that validation material has been added, they often provide a proof of existence of the original data, signature value, signature timestamp, validation material, and possible other archive timestamps that are covered by them

Timestamping, the process of adding a timestamp to a signature, can be broken down into the following steps:

1. The user creates a hash of the data for which a timestamp assertion is required (e.g. signature value for a signature timestamp).
2. The user sends the hash and the digest algorithm to a TSA.
3. The TSA groups the hash, the time of stamping (current date and time) and the identity of the TSA and signs it with a private key contained in a TU.
4. The timestamp token resulting from the previous step is returned to the client.
5. The timestamp token is added to the signature of the data that was sent as a hash in the first step.

An illustration of that process for the creation of a signature timestamp is provided below:



The timestamp token created by a TSA can be considered as trustworthy because

- the TSA is independent from the signing process;
- the clock of the TSA is synchronized with an authoritative time source;
- the timestamp is digitally signed by the TSA;
- the TSA shall follow strict specifications.

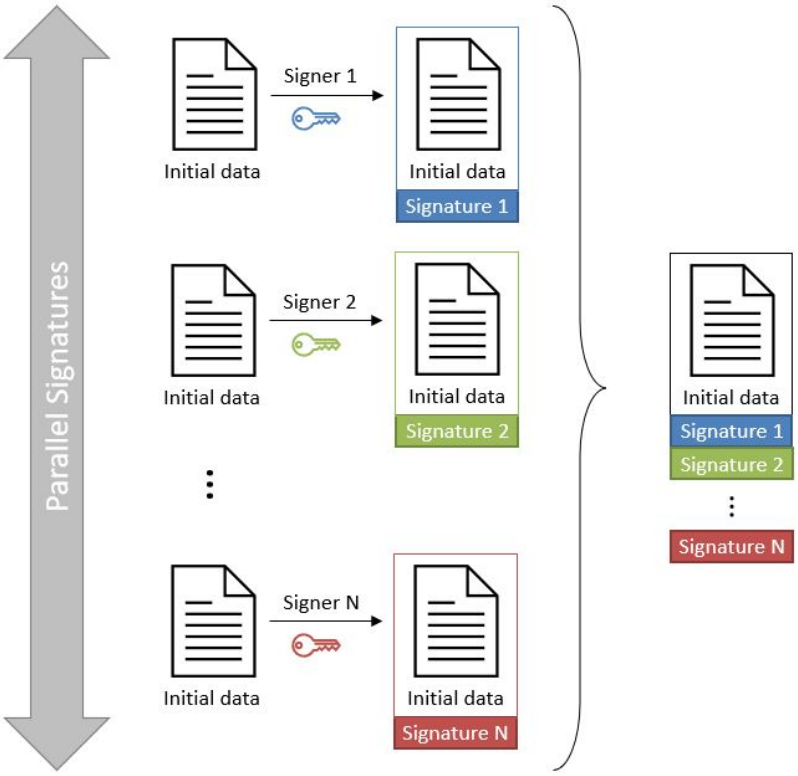
3.3.9. Multiple signatures

Up until now, only creation of a single signature have been covered. However, in most cases multiple signatures need to be created (e.g. a contract signing by multiple parties). In such cases, it is useful to note that multiple signatures can be created in parallel or in a sequential order.

3.3.9.1. Parallel signatures

Parallel signatures are stand-alone, mutually independent signatures where the ordering of the signatures is not important. All the involved parties can receive the data at the same time and sign in any order. The computation of these signatures is performed on exactly the same hash data but using different private keys associated to the different signers. Parallel signatures can be validated independently to verify whether the associated data is validly signed.

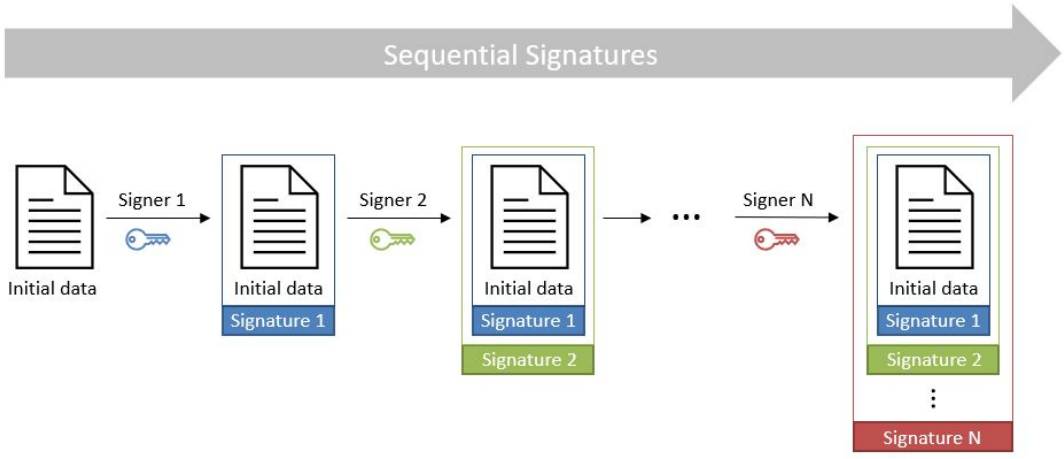
The following schema illustrates the creation of parallel signatures:



3.3.9.2. Sequential signatures

Sequential signatures are mutually dependent signatures where the ordering of the signatures is important. A fixed signing order is defined and the next signer in the chain shall not sign before the preceding signers have signed the data. The computation of these signatures is not performed on the same data. A signer that is further in the signing chain will sign the initial data previously signed by the signers preceding him in the chain. Each signer uses his own private key to sign.

The following schema illustrates the creation of sequential signatures:



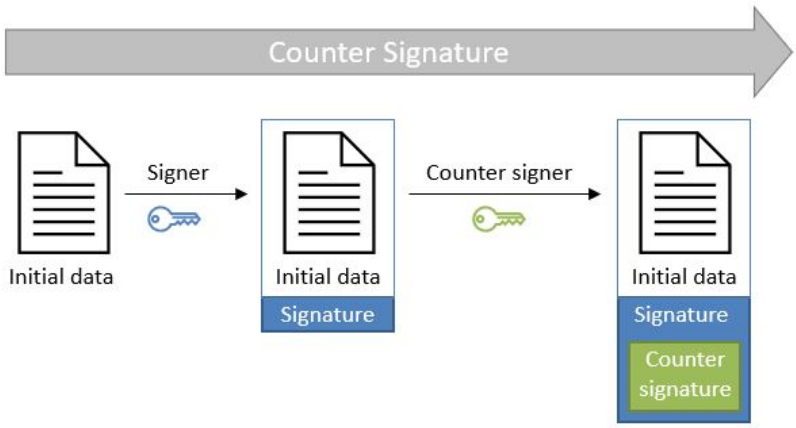
3.3.9.3. Counter signatures

A counter signature is an additional signature applied on data that has already been signed previously. This type of signature is used to show approval of the data and signature, to confirm the authenticity of the data. The computation of a counter signature is performed on the signed data and it is added to the signature as an unsigned attribute, i.e. after initial signature creation.

Counter signatures are often created by trustworthy entities such as notaries, doctors or attorneys.

Possible use cases are rental and mortgage applications, health documents, passports and visas.

The following schema illustrates the creation of counter signatures:



3.3.10. Signature Applicability Rules / Signature Policy

The term "signature policy" is often used to refer to "Signature Applicability Rules", that is, a set of **rules** for the creation, validation and long-term management of one (or more) electronic signature(s).

A Signature Policy, in that meaning, **contains** general information such as:

- the identifier of the signature policy;
- the name of the signature policy issuer;
- the date of issuance of the signature policy;
- the signing period;
- the field of application;
- ...

A Signature Policy is composed of **three main parts** that define technical and procedural requirements:

1. Signature Creation Policy: requirements for the signer in creating a signature;
2. Signature Validation Policy: requirements for the verifier when validating a signature;
3. Signature (LTV) Management Policy: requirements for the long term management and preservation of a signature.

A signature policy is a way of **expressing**:

- who may sign;
- in what capacity an entity may sign;
- what data is being signed;
- in what circumstances the data is signed;
- why the data is being signed (i.e. what are the consequences);

- the purpose for the signature;
- the context in which the signature will be used;
- the means for the creation , verification and long-term management of an electronic signature;
- the means for reproducing the formalities of signing;
- the requirements imposed on or committing the involved actors.

The exact information contained in a signature policy will depend on the use cases of the signature and on the involved parties as the signature policy can be negotiated between them. Therefore, it is not possible to define a single template policy to cover all use cases.

Having a signature policy and thus all the above-mentioned information, available in a signature, has several **advantages**:

- It allows keeping a trace of the decisions that were made during the analysis of the signatures that will need to be created.
- It allows a signature to be legally enforceable in any Member State
- It makes the signature workflow transparent to all involved parties. This enhances trust in electronic signatures that comply with a signature policy.

Parties involved in a signature policy are:

- The Signature policy issuer: a legal/natural entity that sets the rules that compose the signature policy.
- Signature policy users: natural persons that can be one of the two following types of entities:
  1. Signer: creates an electronic signature.
  2. Verifier: ensures the authenticity of the policy and decides whether the signed data is valid or not.
- Trust Service Provider(s).

ETSI ESI has developped several standards to express signature applicability rules or "signature policy" in two **forms**:

- In a human readable form: It can be assessed to meet the requirements of the legal and contractual context in which it is being applied (cf. ETSI TS 119 172-1 [R17]).
- In a machine processable form (XML or ASN.1): To facilitate its automatic processing using the electronic rules (cf. ETSI TS 119 172-2 [R18] and ETSI TS 119 172-3 [R19]).

3.3.10.1. Signature policy at creation and validation

During signature **creation**, a signature creation policy can be added to the signature as a signed attributes of the signature. Signed attributes are information that can only be included upon signature creation and that cannot be added, modified or removed at a later point in the life of the signature. The signature creation policy can be added to the signature indirectly as a reference which is composed of the hash value of the policy and the hash algorithm that was used to hash the policy, or directly when it is in a machine processable form.

During signature **validation**, a mapping between acceptable signature creation policies and their corresponding signature validation policies can be provided to the signature validation application (SVA). If the signature contains one signature creation policy identifier, which is part of the list of mappings, the SVA can then apply the corresponding validation policy during validation.

3.4. Resources

Certain resources have been developed to improve the adoption of the eIDAS Regulation as well as improve information sharing about the eIDAS Regulation and related concepts.

The EU Trust Services Dashboard (EU TSD) is such a resource. It "proposes a centralized platform that enables interested parties and Digital Single Market players to easily and transparently access information and tools related to the trust services chapter of eIDAS".

It contains among others a Trusted List Browser to browse through the trusted lists of the different EU Member States.

eIDAS implementing acts have been issued and adopted by the Commission:

- Commission Implementing Decision (EU) 2015/296: procedural arrangements for cooperation between Member States on electronic identification.
- Commission Implementing Decision (EU) 2015/1501: on the interoperability framework.
- Commission Implementing Decision (EU) 2015/1502: on setting out minimum technical specifications and procedures for assurance levels for electronic identification means.
- Commission Implementing Decision (EU) 2015/1984: circumstances, formats and procedures of notification.
- Commission Implementing Regulation (EU) 2015/806: specifications relating to the form of the EU trust mark for qualified trust services.
- Commission Implementing Decision (EU) 2015/1505: technical specifications and formats relating to trusted lists.
- Commission Implementing Decision (EU) 2015/1506: specifications relating to formats of advanced electronic signatures and advanced seals to be recognised by public sector bodies.
- Commission Implementing Decision (EU) 2016/650: standards for the security assessment of qualified signature and seal creation devices.

ETSI has developed standards that can be followed to be compliant with the eIDAS Regulation.

3.5. Digital signatures in DSS

3.5.1. Tokens in DSS

The Token class is the base class for the different types of tokens used in the process of signature validation which are certificates, OCSPs, CRLs and timestamps. These tokens can be described as follows:

- **CertificateToken:** Whenever the signature validation process encounters an X509Certificate a

certificateToken is created. This class encapsulates some frequently used information: a certificate comes from a certain context (Trusted List, CertStore, Signature), has revocation data, etc. To expedite the processing of such information, they are kept in cache.

- **RevocationToken:** Represents a revocation data token. It can be a CRLToken or an OCSPToken:
  - **CRLToken:** Represents a CRL and provides the information about its validity.
  - **OCSPToken:** OCSP Signed Token which encapsulate BasicOCSPResp (BC).
- **TimestampToken:** SignedToken containing a TimeStamp.
  - **PdfTimestampToken:** Specific class for a PDF Document TimestampToken.

3.5.2. Compliance to ETSI standards

DSS implements the following ETSI standards for various signature forms:

- XAdES digital signatures are compliant with ETSI EN 319 132 part 1-2 ([R01]);
- CAdES digital signatures are compliant with ETSI EN 319 122 part 1-2 ([R02]);
- PAdES digital signatures are compliant with ETSI EN 319 142 part 1-2 ([R03]);
- JAdES digital signatures are compliant with ETSI TS 119 182 part 1 ([R05]);
- ASiC signature containers are compliant with ETSI EN 319 162 part 1-2 ([R04]).

but also claims:

- Creation and validation of AdES digital signatures are compliant with ETSI EN 319 102-1 ([R09]) and ETSI TS 119 102-2 ([R13]).
- The determination of the certificate qualification is compliant with ETSI TS 119 172-4 ([R10]).
- Trusted lists processes are compliant with ETSI TS 119 612 ([R11]).
- Procedures for using and interpreting EU Member States national trusted lists, such as determining the qualified status of a timestamp or of an SSL certificate, are compliant with ETSI TS 119 615 ([R14]).

3.5.3. Out of the EU context

DSS is not limited to EU contexts. It can be used in non-EU contexts with all its basic functions, i.e. signing, augmentation, validation, etc.

An example would be the configuration of trust anchors (see section Trust anchor configuration from a certificate store). The certificate sources can be configured from a TrustStore (kind of keystore which only contains certificates), a trusted list and/or a list of trusted lists. In case of an EU context you could use any of these three trust anchors. For a non-EU context you could use a trust store or a non-EU trusted list. However, non-EU TLs are supported by DSS only if they have the same XML structure as EU TLs, i.e. if they are compliant with the XSD schema. Another constraint is that there is no guarantee for a proper qualification determination as the non-EU TL shall also be compliant with EU regulations.

4. Signature creation

4.1. AdES specificities

4.1.1. Existing formats

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** - for XML Advanced Electronic Signatures (cf. [R01]);
- **CAdES** - for CMS Advanced Electronic Signatures (cf. [R02]);
- **PAdES** - for PDF Advanced Electronic Signatures (cf. [R03]);
- **JAdES** - for JSON Advanced Electronic Signatures (cf. [R05]);
- **ASIC** - for Associated Signature Containers (cf. [R04]). XAdES and CAdES combinations are possible.

4.1.2. Signature profiles

To ensure interoperability of electronic signatures in the EU, the eIDAS Regulation has declared through the Commission Implementing Decision (EU) 2015/1506 (cf. [R16]) that the advanced electronic signatures in the EU shall comply with the baseline standards defined by the following ETSI technical specifications:

- XAdES Baseline Profile: ETSI TS 103171 v.2.1.1;
- CAdES Baseline Profile: ETSI TS 103173 v.2.2.1;
- PAdES Baseline Profile: ETSI TS 103172 v.2.2.2;
- ASiC Baseline Profile: ETSI TS 103174 v.2.2.1.

The baseline profile for a certain signature format provides the basic features required to assure interoperability in the EU for that format. Each baseline profile defines four different levels that correspond to the four signature classes described in the ETSI standard EN 319 102-1 (cf. [R09]) and which are described in the following section.

Before the introduction of the baseline profiles, old extended profiles were used. Below is a comparative table of old extended profiles and new baseline profiles for each signature format:

Table 4. Signature supported profiles

XAdES		CAdES		PAdES		JAdES
EXTENDED	**BASELINE**	**EXTENDED**	**BASELINE**	**EXTENDED**	**BASELINE**	**BASELINE**
XAdES-E-BES	XAdES-B-B	CAdES-E-BES	CAdES-B-B	PAdES-E-BES	PAdES-B-B	JAdES-B-B
XAdES-E-EPES		CAdES-E-EPES		PAdES-E-EPES		

XAdES		CAdES		PAdES		JAdES
XAdES-E-T	XAdES-B-T	CAdES-E-T	CAdES-B-T		PAdES-B-T	JAdES-B-T
XAdES-E-C		CAdES-E-C				
XAdES-E-X		CAdES-E-X				
XAdES-E-X-L	XAdES-B-LT	CAdES-E-X-L	CAdES-B-LT		PAdES-B-LT	JAdES-B-LT
XAdES-E-A	XAdES-B-LTA	CAdES-E-A	CAdES-B-LTA	PAdES-E-LTV	PAdES-B-LTA	JAdES-B-LTA

The DSS framework is compatible with the baseline profiles. However, it is not possible to use the extended profiles for signing purpose except for XAdES-E-A/C/X/XL. The validation of the signature has a basic support of old profiles.

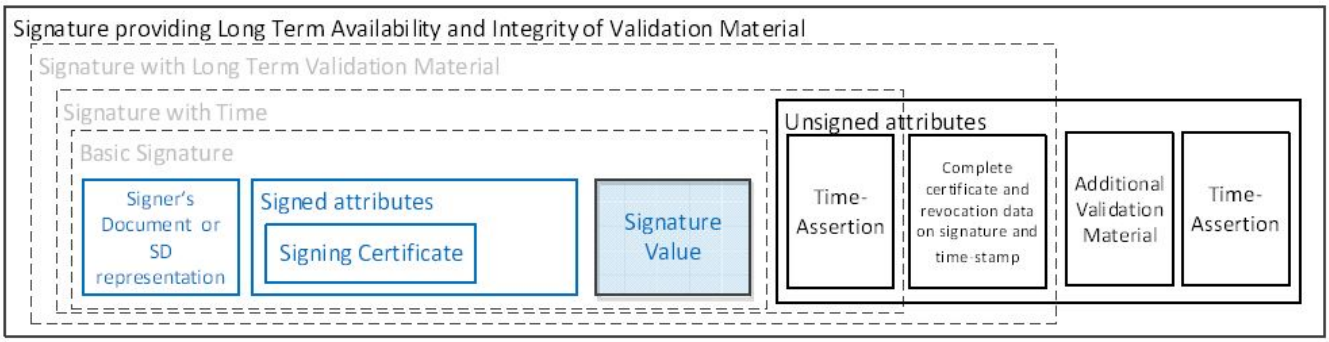
4.1.3. Signature classes/levels

The ETSI standard EN 319 102-1 (cf. [R09]) defines four conformance classes to address the need to protect the validity of the signature in time. Henceforth, to denote the class of the signature the word "level" will be used. Follows the list of profiles implementing the four classes defined in the standard:

- AdES-BASELINE-**B**: Profile implementing the signature class *Basic Signature*
  The lowest and simplest version containing at least a signature value, a reference to or a copy of the signing certificate as a signed attribute, and optionally other signed or unsigned attributes.
- AdES-BASELINE-**T**: Profile implementing the signature class *Signature with time*
  A timestamp regarding the time of signing is added to protect against repudiation.
- AdES-BASELINE-**LT**: Profile implementing the signature class *Signature with Long-Term Validation Material*
  All the material or references to material required for validating the signature are embedded to allow verification in future even if their original source is not available. For example, 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".
- AdES-BASELINE-**LTA**: Profile implementing the signature class *Signature providing Long Term Availability and Integrity of Validation Material*
  By using periodical timestamping (e.g. each year), the availability or integrity of the validation data is maintained. The validity could be limited due to cryptographic obsolescence of the algorithms, keys and parameters used, or due to expiration or revocation of the validation material. The `AdES-BASELINE-LTA` augmentation 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 valid at the time when the used cryptographic algorithms were considered safe. Additional validation data can be included too.

The use of extended profiles on signature creation, such as -E-C, -E-X, -E-XL, -E-A, is supported only for XAdES.

The following schema from the ETSI standard EN 319 102-1 (cf. [R09]) illustrates the different classes.



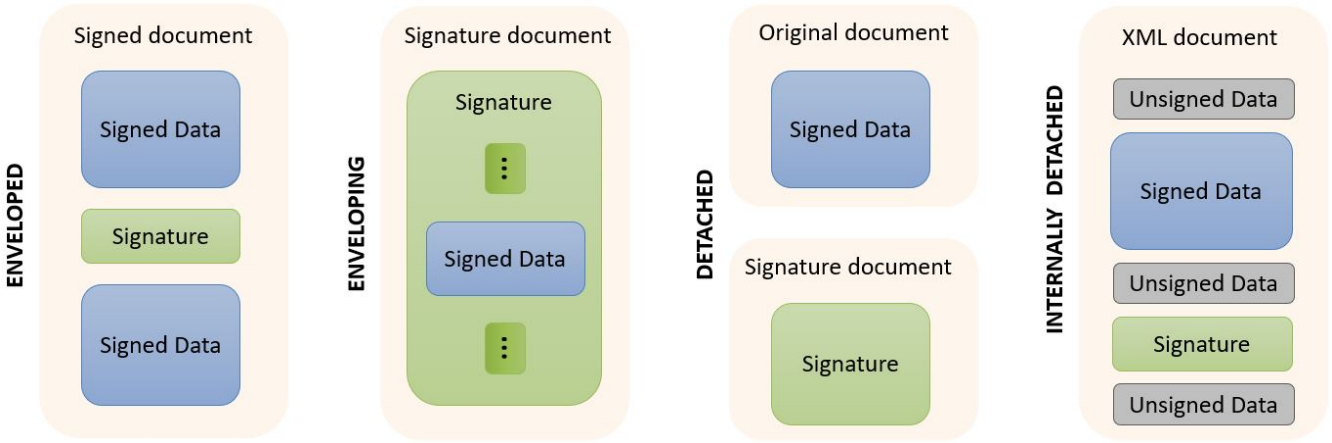
4.1.4. Signature packaging

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.

The 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.

The following types of packaging can be defined for various signature formats:

- **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 resource(s) separated from it;
- **INTERNALLY-DETACHED :** when the signature and the related signed data are both included in a parent element (only XML).



More information about packaging use in combination with available formats can be found in the Signature profile guide.

4.1.4.1. Signature profile guide

Below you can find a table specifying various signature possibilities with available in DSS signature's profiles/formats. The vertical column specifies available signature profiles and their extensions. The horizontal row specifies types of documents to be signed with the formats.

Table 5. File formats and Signature types conformance

Signature profiles			XML	JSON	PDF	Binary	Digest	Multiple files	Multiple signatures	Counter signature	Stand-alone timestamp
XAdES	Enveloping	Base64 encoded	✓	✓	✓	✓	✗	✓	✗	✓	✗
		Embed XML	✓	✗	✗	✗	✗	XML only	✗	✓	✗
		Manifest	✓	✓	✓	✓	✓	✓	✗	✓	✗
		Canonicalization	✓	✗	✗	✗	✗	XML only	✗	✓	✗
	Enveloped	enveloped transformation	✓	✗	✗	✗	✗	✗	✗	✓	✗
		based on XPath	✓	✗	✗	✗	✗	✗	✓	✓	✗
		based on Filter2	✓	✗	✗	✗	✗	✗	✓	✓	✗
		Canonicalization	✓	✗	✗	✗	✗	XML only	✓	✓	✗
	Detached		✓	✓	✓	✓	✓	✓	✗	✓	✗
	Internally Detached		✓	✗	✗	✗	✗	XML only	✓	✓	✗
CAdES	Enveloping		✓	✓	✓	✓	✗	✗	✓	✓	✗
	Detached		✓	✓	✓	✓	✓	✗	✓	✓	✗
PAdES	Enveloped		✗	✗	✓	✗	✗	✗	✓	✗	✓

Signature profiles			XML	JSON	PDF	Binary	Digest	Multiple files	Multiple signatures	Counter signature	Stand-alone timestamp
JAdES	Enveloping	Compact Serialization	✓	✓	✓	✓	✗	✗	✗	✗	✗
		Flattened JSON Serialization	✓	✓	✓	✓	✗	✗	✗	✓	✗
		JSON Serialization	✓	✓	✓	✓	✗	✗	✓	✓	✗
	Detached	Compact Serialization	✓	✓	✓	✓	✓	SigD only	✗	✗	✗
		Flattened JSON Serialization	✓	✓	✓	✓	✓	SigD only	✗	✓	✗
		JSON Serialization	✓	✓	✓	✓	✓	SigD only	✓	✓	✗
ASiC	ASiCS	CAdES / XAdES	✓	✓	✓	✓	✗	✓	✓	✓	✓
	ASiCE	CAdES / XAdES	✓	✓	✓	✓	✗	✓	✓	✓	✓

4.1.5. Signature algorithms

DSS supports several signature algorithms (combination of an encryption algorithm and a digest algorithm). Below, you can find the supported combinations. The support of the algorithms depends on the registered OID (ASN1) or URI (XML).

In the next table, XAdES also applies to ASiC with embedded XAdES signatures and CAdES also concerns PAdES and ASiC with embedded CAdES signatures.

SmartCards/HSMs don't allow signing with all digest algorithms. Please refer to your SmartCard/HSM provider.

Table 6. Supported algorithms

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512	SHA3-224	SHA3-256	SHA3-384	SHA3-512	MD2	MD5	RIPEMD160
RSA												
XAdES	✓	✓	✓	✓	✓						✓	✓
CAdES	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
JAdES			✓	✓	✓							
RSA-PSS												
XAdES	✓	✓	✓	✓	✓	✓	✓	✓	✓			
CAdES	✓	✓	✓	✓	✓	✓	✓	✓	✓			
JAdES			✓	✓	✓							
ECDSA												
XAdES	✓	✓	✓	✓	✓							✓
CAdES	✓	✓	✓	✓	✓	✓	✓	✓	✓			
JAdES			✓	✓	✓							
Ed25519												
CAdES					✓							
DSA												
XAdES	✓		✓									
CAdES	✓	✓	✓	✓	✓	✓	✓	✓	✓			
HMAC												
XAdES	✓	✓	✓	✓	✓							✓
CAdES	✓	✓	✓	✓	✓	✓	✓	✓	✓			
JAdES			✓	✓	✓							

4.2. Representation of documents in DSS

DSS allows creation of different kinds of DSSDocument :

- `InMemoryDocument` : fully loads the document in memory. This type of `DSSDocument` can be instantiated with an array of bytes or an InputStream.
- `FileDocument` : created from an existing local file.
- `DigestDocument` : only contains pre-computed digest values for a given document. That allows a user to avoid sending the full document (detached signatures).
- `CMSSignedDocument` : an internal implementation of a `DSSDocument`, loading a CMS Signed Data object.
- `HTTPHeader` : represents an HTTP Header used as a signed object within a JAdES implementation (see JAdES Detached Packaging).
- `HTTPHeaderDigest` : represents an HTTP body message's digest to be used as a signed HTTP Header within JAdES Detached Packaging. The object is built from another `DSSDocument` (e.g. from binaries, digest document, etc.).

InMemoryDocument

```
// We can instantiate an InMemoryDocument from binaries
DSSDocument binaryInMemoryDocument = new InMemoryDocument("Hello World".getBytes());

// Or from InputStream
DSSDocument isInMemoryDocument;
try (InputStream is = new FileInputStream("src/main/resources/xml_example.xml")) {
    isInMemoryDocument = new InMemoryDocument(is);
}
```

FileDocument

```
// Instantiate a FileDocument from a File
DSSDocument fileDocument = new FileDocument(new
File("src/main/resources/xml_example.xml"));

// Or from file path directly
DSSDocument filePathDocument = new FileDocument("src/main/resources/xml_example.xml");
```

DigestDocument

```
// Firstly, we load a basic DSSDocument (FileDocument or InMemoryDocument)
DSSDocument fileDocument = new FileDocument("src/main/resources/xml_example.xml");

// After that, we create a DigestDocument
DigestDocument digestDocument = new DigestDocument(DigestAlgorithm.SHA1,
fileDocument.getDigest(DigestAlgorithm.SHA1));
digestDocument.setName(fileDocument.getName());

// We can add additional digest values when required. Eg : for a SHA-256 based
signature
digestDocument.addDigest(DigestAlgorithm.SHA256,
fileDocument.getDigest(DigestAlgorithm.SHA256));

// Or incorporate digest value as a String directly
digestDocument.addDigest(DigestAlgorithm.SHA512,
"T1h8Ss0fiK0pfo1chVoLumIhyIgR9I0g8IvPhJPxwnR5dPFhLDEMU5kpt3AE4xnU2dagh6JaMz1INaCkO0LIt
g==");
```

HTTPHeader

```
// An HTTPHeader shall be defined with a header name and a value
DSSDocument httpHeader = new HTTPHeader("content-type", "application/json");

// An `digest` HTTP Header can be created from a HTTP body message, using a
DSSDocument and a desired DigestAlgorithm
DSSDocument httpBodyMessage = new InMemoryDocument("Hello World!".getBytes());
DSSDocument httpHeaderDigest = new HTTPHeaderDigest(httpBodyMessage,
DigestAlgorithm.SHA256);
```

4.3. Signature tokens

The DSS framework is able to create signatures using different keystores: PKCS#11, PKCS#12, JKS, MS CAPI and Apple Keystore. To be independent of the signing media, the DSS framework uses an interface named `SignatureTokenConnection` to manage different implementations of the signing process. All token implementations provided within the framework extend the `AbstractSignatureTokenConnection` abstract class that allows executing complete signature operation (digest and encryption on the token) or raw signature operation (external digest and encryption on the token).

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

4.3.1. 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. It requires some installed drivers (dll, sso, etc.).

Pkcs11SignatureToken usage

```
try (Pkcs11SignatureToken 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 : " +
Utils.toBase64(signatureValue.getValue()));
}
```

4.3.2. PKCS#12

This standard defines a file format commonly used to store the private key and corresponding public key certificate protecting them with a password. It allows signing with a PKCS#12 keystore (.p12 file). In order to use this format with the DSS framework you have to go through the class Pkcs12SignatureToken.

Pkcs12SignatureToken usage

```
try (Pkcs12SignatureToken token = new
Pkcs12SignatureToken("src/main/resources/user_a_rsa.p12", new
PasswordProtection("password".toCharArray()))) {

    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 : " +
Utils.toBase64(signatureValue.getValue()));
}
```

4.3.3. MS CAPI

If the middleware for communicating with an SCDev provides a CSP based on MS CAPI (the Microsoft interface to communicate with SmartCards) specification, then you can use the `MSCAPISignatureToken` class to sign the documents.

MSCAPISignatureToken usage

```
try (MSCAPISignatureToken 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 : " +
Utils.toBase64(signatureValue.getValue()));
}
```

4.3.4. Apple Keystore

Since DSS 5.10 a new class `AppleSignatureToken` is provided within the source code allowing to access a Keychain store in a MacOS environment.

The API does not access keys from a smartcard, as opposite to the MS CAPI implementation.

AppleSignatureToken usage

```
try (AppleSignatureToken token = new AppleSignatureToken()) {

    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 : " +
Utils.toBase64(signatureValue.getValue()));
}
```

4.3.5. Java Key Store

The `JKSSignatureToken` class allows signing with a Java Key Store (.jks file).

JKSSignatureToken usage

```
try (InputStream is = new FileInputStream("src/main/resources/keystore.jks");
     JKSSignatureToken token = new JKSSignatureToken(is, new PasswordProtection("dss-
password".toCharArray()))) {

    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 : " +
Utils.toBase64(signatureValue.getValue()));
}
```

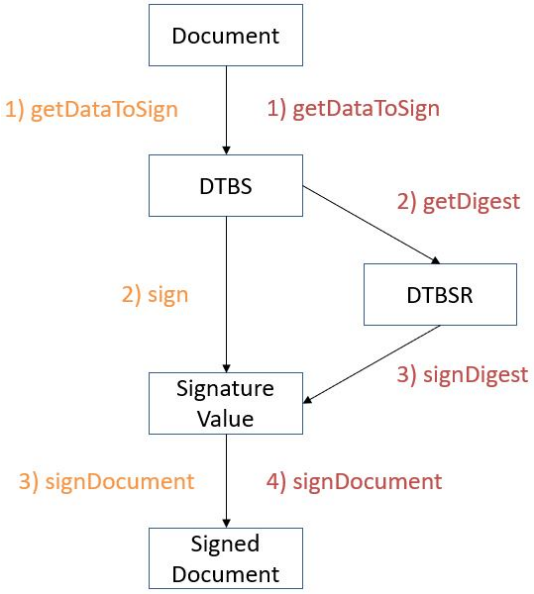
4.3.6. Other Implementations

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.

As you can see, it is easy to add another implementation of the `SignatureTokenConnection`, thus enabling the framework to use other API than provided within the framework. 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.

4.4. Signature creation in DSS

In the DSS framework, there are two alternatives for signing a document which are illustrated in the following schema.



The first path consists in 3 atomic steps:

1. Compute the data to be signed (DTBS);
2. Sign the DTBS to obtain the signature value;
3. Sign the document (add the signature value).

The alternative is to use the following 4 steps:

1. Compute the data to be signed (DTBS);
2. Compute the digest of the DTBS to obtain the data to be signed representation (DTBSR);
3. Sign the DTBSR to obtain the signature value;
4. Sign the document (add the signature value).

4.4.1. Signature creation in 3 stateless methods

Usually, the signature creation is done using the 3 stateless methods approach.

DSS fully manages the first and last steps of the document signature process. The signature operation (signing the DTBS) needs to be specified. DSS offers some implementations in the `dss-token` module. During this step, the signing keys are not retrieved. Instead, a communication channel that allows sending the DTBS to the keys is opened.

The following code presents the three stateless steps.

Three stateless steps

```
// step 1: generate ToBeSigned data
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, signatureParameters);

// step 2: sign ToBeSigned data using a private key
SignatureValue signatureValue = signingToken.sign(dataToSign, digestAlgorithm,
privateKey);

// step 3: sign document using a SignatureValue obtained on the previous step
DSSDocument signedDocument = service.signDocument(toSignDocument, signatureParameters,
signatureValue);
```

The first step uses the `getDataToSign()` method, which receives the following arguments:

- the document to be signed;
- the previously selected settings.

This step returns the DTBS. In the case of a XAdES, it corresponds to the SignedInfo XMLDSig element. Usually the DTBS is composed of the digest of the original document and the signed attributes (i.e. XAdES or CAdES).

The second 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:

- Array of bytes that must be signed. It is obtained by the previous method invocation.
- Algorithm used to create the digest. There is the choice between, for example, SHA256 and SHA512. This list is not exhaustive. For an exhaustive list see this class.
- Private key entry.

The third and 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 done using the method `signDocument()` on the service. It requires three parameters:

- Document to sign;
- Signature parameters (the same as the ones used in the first step);
- 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.

4.4.2. Signature creation in 4 stateless methods

The main difference in this approach is that the digest of DTBS (i.e. DTBSR) is computed separately from a SignatureValue creation.

The following code presents the alternative with four stateless steps.

Four stateless steps

```
// step 1: generate ToBeSigned data
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, signatureParameters);

// step 2: compute the digest of the ToBeSigned data
Digest digest = new Digest(signatureParameters.getDigestAlgorithm(),
DSSUtils.digest(signatureParameters.getDigestAlgorithm(), dataToSign.getBytes()));

// step 3: sign the digested data using a private key
SignatureValue signatureValue = signingToken.signDigest(digest, privateKey);

// step 4: sign document using a SignatureValue obtained on the previous step
DSSDocument signedDocument = service.signDocument(toSignDocument, signatureParameters,
signatureValue);
```

The first and last steps, which compute the DTBS and sign the document, are the same as for the alternative with three steps.

The second step uses the `digest` method, which takes the following arguments

- the digest algorithm;
- the DTBS computed during the first step.

It returns the data to be signed representation (DTBSR) of a DTBS, i.e. it computes the digest.

The step that signs the digest with a raw signature (eg: NONEwithECDSA) and returns the signature value uses the method `signDigest`, which takes the following arguments:

- the DTBSR.
- the private key entry.

4.5. Creating a Baseline B-level signature

The simplest way to address the digital signature passes through the XAdES format. Indeed, it allows visualization of the signature content 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]).

The referenced specifications define the following:

- To digitally sign a document, a signing certificate (that proves the signer's identity) and the access to its associated private key is needed.
- To digitally 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, associated to the private key used to create the signature, must also be provided.

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

xml_example.xml

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

To instantiate a document from a file in DSS, refer to the section about DSSDocuments.

Since this is an XML document, we will use the XAdES signature and more particularly the XAdES-BASELINE-B level. For our example, we will use the ENVELOPED packaging.

To write our Java code, we still need to specify the type of KeyStore to use for signing our document. In other words, we need to specify where the private key can be found. To know more about the use of the different signature tokens, please consult the Signature tokens section.

In this example the class `Pkcs12SignatureToken` will be used. A file in PKCS#12 format must be provided to the constructor of the class. It contains a private key accompanying the X.509 public key certificate and protected by a password. The certificate 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 and http://dss.nowina.lu/pki-factory/keystore/list to access dummy certificates and their chains.

This is the complete code example that allows you to sign an XML document:

Create a XAdES-BASELINE-B signature

```
// 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);
```

To summarize, signing a document in DSS requires to:

- Create an object based on the `SerializableSignatureParameters` class. Generally, the number of specified parameters depends on the format, profile and level of the signature. This object also defines some default parameters.
- Choose the level, packaging and signature digest algorithm.
- Indicate the private key entry as well as the associated signing certificate and certificate chain.
- Instantiate the adequate signature service based on the format of the signature. In our case it is an XML file, so we will instantiate a XAdES service.
- Carry out the signature process (three or four steps). in our case, the process of signing takes place in three stages.

The encryption algorithm is determined by the private key and therefore shall not be compelled by the setter of the signature parameters object. It would 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.

Setting the signing certificate and the certificate chain should always be done. They can be set even if the private key is only known to the signature value creation process and there is no access to the private key at DTBS preparation. However, of course, the signing certificate shall be associated to the private key that will be used at the signature value creation step. Integrating the certificate chain in the signature simplifies the build of a prospective certificate chain during the validation process.

When the specific service is instantiated a certificate verifier must be set. It is recommended to read section CertificateVerifier configuration for information on the configuration of a `CertificateVerifier`.

The code above produces the signature that can be found here.

4.6. Configuration of attributes in DSS

4.6.1. Signed and unsigned attributes

A signature is composed of signed and unsigned attributes.

Signed attributes shall be included in a signature. These are BASELINE-B attributes that are set upon signature creation. They cannot be changed after the signature has been created.

Unsigned attributes are not obligatory and can be added after the signature creation during signature augmentation.

4.6.1.1. Table with all attributes per format and class

Table 7. File formats and Signature types conformance

		DSS classes and methods	XAdES	CAdES	PAdES	JAdES
B-Level	Signed attributes	*class BLevelParameters* **#setSigningDate**	SigningTime	signing-time	/M	sigT
		class BLevelParameters **#setClaimedSignerRole**	SignerRoleV2	signer-attributes-v2	signer-attributes-v2	srAts
		class BLevelParameters **#setSignedAssertions**	SignedAssertions	signedAssertions	signedAssertions	signedAssertions
		class BLevelParameters **#setCommitmentTypeIndications**	CommitmentTypeIndication	commitment-type-indication	commitment-type-indication	srCms
		class BLevelParameters **#setSignerLocation**	SignatureProductionPlaceV2	signer-location	/Location	sigPl
		class BLevelParameters **#setSignaturePolicy**	SignaturePolicyIdentifier	signature-policy-identifier	signature-policy-identifier	sigPId

		DSS classes and methods	XAdES	CAdES	PAdES	JAdES
B-Level	Signed attributes	*class* **AdESSignatureParameters* **#setSigningCertificate**	SigningCertificate SigningCertificateV2	signing-certificate-v2	signing-certificate-v2	x5t#256 x5t#o
		class **AdESSignatureParameters* **#setCertificateChain**	KeyInfo/X509Data	SignedData.certificates	SignedData.certificates	x5c
		class **AdESSignatureParameters* **#setEncryptionAlgorithm #setDigestAlgorithm**	SignatureMethod	/	/	alg
		class **AdESSignatureParameters* **#setContentTimestamps**	AllDataObjectsTimeStamp IndividualDataObjectsTimeStamp	content-time-stamp	content-time-stamp	adoTst
		class *XAdESSignatureParameters* **#setReferences**	SignedInfo/Reference	/	/	/

		DSS classes and methods	XAdES	CAdES	PAdES	JAdES
B-Level	Signed attributes	*class XAdESSignatureParameters* **#setSignedAdESObject**	Object	/	/	/
		class CAdESSignatureParameters **#setContentHintsType #setContentHintsDescription**	/	content-hints	/	/
		class CAdESSignatureParameters **#setContentIdentifierPrefix #setContentIdentifierSuffix**	/	content-identifier	/	/
		class PAdESSignatureParameters **#setReason**	/	/	/Reason	/
		class PAdESSignatureParameters **#setContentSize**	/	/	/Contents (length)	/

		DSS classes and methods	XAdES	CAdES	PAdES	JAdES
B-Level	Signed attributes	*class PAdESSignatureParameters* **#setFilter**	/	/	/Filter	/
		class PAdESSignatureParameters **#setSubFilter**	/	/	/SubFilter	/
		class PAdESSignatureParameters **#setSignerName**	/	/	/Name	/
		class PAdESSignatureParameters **#setImageParameters**	/	/	Visual representation	/
		class PAdESSignatureParameters **#setPermission**	/	/	/P (CertificationPermission)	/

		DSS classes and methods	XAdES	CAdES	PAdES	JAdES
B-Level	Signed attributes	*class JAdESSignatureParameters* **#setIncludeCertificateChain**	/	/	/	x5c
		class JAdESSignatureParameters **#setIncludeSignatureType**	/	/	/	typ
		class JAdESSignatureParameters **#setIncludeKeyIdentifier**	/	/	/	kid
		class JAdESSignatureParameters **#setSigningCertificateDigestMethod (DigestAlgorithm.SHA256)**	/	/	/	x5t#S256
		class JAdESSignatureParameters **#setSigDMechanism**	/	/	/	sigD

		DSS classes and methods	XAdES	CAdES	PAdES	JAdES
B-Level	Signed attributes	*class JAdESSignatureParameters* **#setBase64UrlEncodedPayload**	/	/	/	b64
		class DSSDocument (for signer document) **#setMimeType**	DataObjectFormat/ MimeType	mime-type	/	cty
	Unsigned attributes	*class *AdESService* **#getDataToBeCounterSigned #counterSignSignature**	CounterSignature	countersignature	/	cSig
		*class *AdESService* **#addSignaturePolicyStore**	SignaturePolicyStore	signature-policy-store	/	sigPSt
		class JAdESSignatureParameters **#setBase64UrlEncodedEtsiUComponents**	/	/	/	etsiU (items encoding)

		DSS classes and methods	XAdES	CAdES	PAdES	JAdES
T-Level	Unsigned attributes	*class* ***AdESSignatureParameters*** **#setSignatureLevel(*AdES_BASELINE_T)** **#setSignatureTimestampParameters**	SignatureTimeStamp	signature-time-stamp	signature-time-stamp	sigTst
LT-Level	Unsigned attributes	*class* ***AdESSignatureParameters*** **#setSignatureLevel(*AdES_BASELINE_LT)** **#setSignatureLevel(XAdES_XL)**	CertificateValues RevocationValues	SignedData.certificates SignedData.crls	/DSS /VRI	xVals rVals
LTA-Level	Unsigned attributes	*class* ***AdESSignatureParameters*** **#setSignatureLevel(*AdES_BASELINE_LTA)** **#setSignatureLevel(XAdES_A)** **#setArchiveTimestampParameters**	ArchiveTimeStamp	archive-time-stamp-v3	document-time-stamp	arcTst

		DSS classes and methods	XAdES	CAdES	PAdES	JAdES
C-Level *(xades only)*	Unsigned attributes	*class XAdESSignatureParameters* **#setSignatureLevel(XAdES_C)**	CompleteCertificateRefs CompleteRevocationRefs	*(not supported)*	*(not supported)*	*(not supported)*
X-Level *(xades only)*	Unsigned attributes	*class XAdESSignatureParameters* **#setSignatureLevel(XAdES_X)**	SigAndRefsTimeStamp	*(not supported)*	*(not supported)*	*(not supported)*

4.6.2. Signed attributes

Additional signed attributes can be added to the basic signature configuration as presented in the following example of a XAdES-BASELINE-B signature.

XAdES signature with additional signed attributes

```
XAdESSignatureParameters parameters = new XAdESSignatureParameters();

// Basic signature configuration
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPING);
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_B);
parameters.setDigestAlgorithm(DigestAlgorithm.SHA512);
parameters.setSigningCertificate(privateKey.getCertificate());
parameters.setCertificateChain(privateKey.getCertificateChain());

// Configuration of several signed attributes like ...
BLevelParameters bLevelParameters = parameters.bLevel();

// Contains claimed roles assumed by the signer when creating the signature
bLevelParameters.setClaimedSignerRoles(Arrays.asList("Manager"));

// signer location
SignerLocation signerLocation = new SignerLocation();
signerLocation.setCountry("BE");
signerLocation.setStateOrProvince("Luxembourg");
signerLocation.setPostalCode("1234");
signerLocation.setLocality("SimCity");
// Contains the indication of the purported place where the signer claims to have
produced the signature
bLevelParameters.setSignerLocation(signerLocation);

// Identifies the commitment undertaken by the signer in signing (a) signed data
object(s)
// in the context of the selected signature policy
List<CommitmentType> commitmentTypeIndications = new ArrayList<>();
commitmentTypeIndications.add(CommitmentTypeEnum.ProofOfOrigin);
commitmentTypeIndications.add(CommitmentTypeEnum.ProofOfApproval);
// NOTE: CommitmentType supports also IDQualifier and documentationReferences.
// To use it, you need to have a custom implementation of the interface.
bLevelParameters.setCommitmentTypeIndications(commitmentTypeIndications);

CommonCertificateVerifier verifier = new CommonCertificateVerifier();
XAdESService service = new XAdESService(verifier);
service.setTspSource(getOnlineTSPSource());

// Allows setting of content-timestamp (part of the signed attributes)
TimestampToken contentTimestamp = service.getContentTimestamp(toSignDocument,
parameters);
parameters.setContentTimestamps(Arrays.asList(contentTimestamp));
```

```
// Signature process with its 3 stateless steps
ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);
SignatureValue signatureValue = signingToken.sign(dataToSign,
parameters.getDigestAlgorithm(), privateKey);
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

In the XAdES format the following types of Content Timestamp can be used:

- **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.

By default, the AllDataObjectsTimeStamp is used. The IndividualDataObjectsTimeStamp can be configured manually.

To set the Content Timestamp, a TSP source needs to be set. The method `getOnlineTSPSource()` is not a core feature of DSS and shall be implemented by the user.

Concerning the signing date, the framework uses the current date time by default. In the case where it is necessary to indicate a different time it is possible to use the setter `setSigningDate(Date)`.

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

The code above produces the signature that can be found here.

4.6.2.1. Signature Policy

The signature policy is a BASELINE-B signed attribute that is set upon signature creation. Thus, this attribute cannot be changed after the signature has been created.

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. For more information on the signature policy refer to section Signature Applicability Rules / Signature Policy.

The signer may reference the policy either implicitly or explicitly.

- An implied policy means the signer follows the rules of a 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 the `SignaturePolicyIdentifier` element will be empty.
- When the policy is explicit, the signature contains an `ObjectIdentifier` 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 content of the policy document.

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

XAdES with implicit policy

```
BLevelParameters bLevelParameters = parameters.bLevel();

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

bLevelParameters.setSignaturePolicy(policy);
```

The following XML segment will be added to the signature's qualifying and signed properties (<QualifyingProperties><SignedProperties>):

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

The next example demonstrates the use of an explicit policy identifier. This is obtained by setting the BASELINE-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 up to the developer to proceed with the computation. 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.

XAdES with explicit policy

```
BLevelParameters bLevelParameters = parameters.bLevel();

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

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

bLevelParameters.setSignaturePolicy(policy);
```

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

XAdES Signature Policy element

```
<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>
```

4.6.2.2. Signature Policy Store

DSS provides a possibility of incorporation of a Signature Policy Store element as an unsigned property to the existing signature file.

The following signature formats support the Signature Policy Store addition:

- XAdES (as well as ASiC with XAdES);
- CAdES (as well as ASiC with CAdES);
- JAdES.

Being an unsigned component, the Signature Policy Store is not protected by a digital signature, unlike for example a Signature Policy Identifier incorporated into the signed properties.

Before incorporating of a Signature Policy Store, you need to ensure the target signature contains the matching Signature Policy Identifier element (see section Signature Policy).

An example of a Signature Policy Store creation is available below:

Add SignaturePolicyStore

```
// Create the SignaturePolicyStore object
SignaturePolicyStore signaturePolicyStore = new SignaturePolicyStore();
// Provide the policy content referenced within the Signature Policy Identifier
signaturePolicyStore.setSignaturePolicyContent(policyContent);
// Define the Id of the policy
SpDocSpecification spDocSpec = new SpDocSpecification();
spDocSpec.setId(signaturePolicyId);
signaturePolicyStore.setSpDocSpecification(spDocSpec);

// Add the SignaturePolicyStore
XAdESService xadesService = new XAdESService(commonCertificateVerifier);
DSSDocument signedDocumentWithSignaturePolicyStore =
xadesService.addSignaturePolicyStore(signedDocument, signaturePolicyStore);
```

4.6.3. Trust Anchor Inclusion Policy

In DSS, it is possible to indicate whether to include or not the certificate related to the trust anchor in the signature. Refer to section Trust Anchors and Trust Stores for information on trust anchors.

By default, when a `BASELINE-B` signature is constructed the trust anchor is not included, only the certificates previous to the trust anchor are included. When a `BASELINE-LT` signature is constructed the trust anchor is included.

It is possible to indicate to the framework that the certificate related to the trust anchor should be included to the signature even at the `BASELINE-B` level. The setter `setTrustAnchorBPPolicy(trustAnchorBPPolicy)` of the `BLevelParameters` class should be used for this purpose.

By default, the argument `trustAnchorBPPolicy` is set to true so that only the certificates previous to the trust anchor are included. It should be set to false in order to include all certificates, including the trust anchor.

Use of Trust Anchor Inclusion Policy parameter

```
// Enforce inclusion of trust anchors into the signature
parameters.bLevel().setTrustAnchorBPPolicy(false);
```

4.6.4. Other parameters

4.6.4.1. XAdES

4.6.4.1.1. Canonicalization parameters

Before computing digests on an XML element, a canonicalization should be performed.

DSS allows defining canonicalization algorithms to be used on signature or timestamp creation:

Use of canonicalization algorithm parameters

```
// Sets canonicalization algorithm to be used for digest computation for the
ds:Reference referencing
// xades:SingedProperties element
parameters.setSignedPropertiesCanonicalizationMethod(CanonicalizationMethod.EXCLUSIVE)
;

// Sets canonicalization algorithm to be used for digest computation for the
ds:SignedInfo element
parameters.setSignedInfoCanonicalizationMethod(CanonicalizationMethod.EXCLUSIVE);

// Defines canonicalization algorithm to be used for formatting ds:KeyInfo element
// NOTE: ds:KeyInfo shall be a signed property in order for the method to take effect
parameters.setKeyInfoCanonicalizationMethod(CanonicalizationMethod.EXCLUSIVE);
// To be used to enforce signing of ds:KeyInfo element
parameters.setSignKeyInfo(true);

// It is also possible to define canonicalization algorithm for a timestamp
XAdESTimestampParameters timestampParameters = new XAdESTimestampParameters();
// ...
timestampParameters.setCanonicalizationMethod(CanonicalizationMethod.EXCLUSIVE);
// Set timestamp parameters to the signature parameters, e.g. for archival timestamp:
parameters.setArchiveTimestampParameters(timestampParameters);
```

4.6.4.1.2. References

In order to compute digest for original document(s) in a custom way, a class `DSSReference` can be used to define a `ds:Reference` element's content:

DSSReference definition

```
List<DSSReference> references = new ArrayList<>();
// Initialize and configure ds:Reference based on the provided signer document
DSSReference dssReference = new DSSReference();
dssReference.setContents(toSignDocument);
dssReference.setId("r-" + toSignDocument.getName());
dssReference.setTransforms(transforms);
// set empty URI to cover the whole document
dssReference.setUri("");
dssReference.setDigestMethodAlgorithm(DigestAlgorithm.SHA256);
references.add(dssReference);
// set references
parameters.setReferences(references);
```

For more information about `DSSReference` configuration, please refer the section Reference Transformations.

4.6.4.2. PAdES

The framework allows creation of PDF files with visible signature as specified in ETSI EN 319 142 (cf. [R03]) and allows the insertion of a visible signature to an existing field. For more information on the possible parameters see section PAdES Visible Signature.

4.7. Multiple signatures

4.7.1. Parallel signatures

Parallel signatures, described in section Parallel signatures, can be created in DSS according to the corresponding AdES formats.

Parallel signatures creation

```
// Load the user token to create the first signature
try (SignatureTokenConnection goodUserToken = getPkcs12Token()) {

    // Preparing parameters for the XAdES signature
    XAdESSignatureParameters parameters = initSignatureParameters();

    // ENVELOPED SignaturePackaging should be used for a parallel signature creation
    parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);

    // Set the signing certificate and a certificate chain for the used token
    DSSPrivateKeyEntry privateKey = goodUserToken.getKeys().get(0);
    parameters.setSigningCertificate(privateKey.getCertificate());
    parameters.setCertificateChain(privateKey.getCertificateChain());

    // Sign in three steps
    ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);
    SignatureValue signatureValue = goodUserToken.sign(dataToSign,
parameters.getDigestAlgorithm(), privateKey);
    signedDocument = service.signDocument(toSignDocument, parameters, signatureValue);
}

signingAlias = RSA_SHA3_USER;
// Load the second user token
try (SignatureTokenConnection rsaUserToken = getPkcs12Token()) {

    // Preparing parameters for the XAdES signature
    XAdESSignatureParameters parameters = initSignatureParameters();

    // ENVELOPED SignaturePackaging should be used for a parallel signature creation
    parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);

    // Set the signing certificate and a certificate chain for the used token
    DSSPrivateKeyEntry privateKey = rsaUserToken.getKeys().get(0);
    parameters.setSigningCertificate(privateKey.getCertificate());
    parameters.setCertificateChain(privateKey.getCertificateChain());

    // Sign in three steps using the document obtained after the first signature
    ToBeSigned dataToSign = service.getDataToSign(signedDocument, parameters);
    SignatureValue signatureValue = rsaUserToken.sign(dataToSign,
parameters.getDigestAlgorithm(), privateKey);
    doubleSignedDocument = service.signDocument(signedDocument, parameters,
signatureValue);

}
```

4.7.2. Sequential signatures

DSS allows creation of sequential signatures according to the PAdES formats (cf. Sequential

signatures).

Sequential signatures creation

```
// Load the user token to create the first signature
try (SignatureTokenConnection goodUserToken = getPkcs12Token()) {

    // Preparing parameters for the PAdES signature
    PAdESSignatureParameters parameters = initSignatureParameters();

    // Set the signing certificate and a certificate chain for the used token
    DSSPrivateKeyEntry privateKey = goodUserToken.getKeys().get(0);
    parameters.setSigningCertificate(privateKey.getCertificate());
    parameters.setCertificateChain(privateKey.getCertificateChain());

    // Sign in three steps
    ToBeSigned dataToSign = service.getDataToSign(toSignDocument, parameters);
    SignatureValue signatureValue = goodUserToken.sign(dataToSign,
parameters.getDigestAlgorithm(), privateKey);
    signedDocument = service.signDocument(toSignDocument, parameters, signatureValue);
}

signingAlias = RSA_SHA3_USER;
// Load the second user token
try (SignatureTokenConnection rsaUserToken = getPkcs12Token()) {

    // Preparing parameters for the PAdES signature
    PAdESSignatureParameters parameters = initSignatureParameters();

    // Set the signing certificate and a certificate chain for the used token
    DSSPrivateKeyEntry privateKey = rsaUserToken.getKeys().get(0);
    parameters.setSigningCertificate(privateKey.getCertificate());
    parameters.setCertificateChain(privateKey.getCertificateChain());

    // Sign in three steps using the document obtained after the first signature
    ToBeSigned dataToSign = service.getDataToSign(signedDocument, parameters);
    SignatureValue signatureValue = rsaUserToken.sign(dataToSign,
parameters.getDigestAlgorithm(), privateKey);
    doubleSignedDocument = service.signDocument(signedDocument, parameters,
signatureValue);

}
```

4.8. Counter signatures

DSS allows creation of counter signatures in accordance with corresponding AdES formats (cf. Counter signatures).

A counter signature does not provide a Proof Of Existence for a signed signature! Use signature augmentation / timestamping for this purpose.

The following formats are supported for the counter signature creation:

- `XAdES` - multiple, nested and augmented counter signatures (up to LTA level) are allowed;
- `CAdES` - B-level counter signatures are allowed, as well as multiple counter signatures. This limitation comes from the cryptography library (bouncyCastle) and not from the standard;
- `JAdES` - multiple, nested and augmented signatures (up to LTA level) are allowed;
- `ASiC` - counter signatures are allowed according to the used format (XAdES or CAdES).

In order to create a counter signature, the DSS Identifier (or XML Id for XAdES) of the target signature you want to sign shall be provided within the parameters. The example below presents a counter signature creation:

Counter signature creation

```
// Initialize counter signature parameters
XAdESCounterSignatureParameters counterSignatureParameters = new
XAdESCounterSignatureParameters();
// Set signing certificate parameters
counterSignatureParameters.setSigningCertificate(privateKey.getCertificate());
counterSignatureParameters.setCertificateChain(privateKey.getCertificateChain());
// Set target level of the counter signature
counterSignatureParameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_B);

// Next step is to extract and set the Id of a signature to be counter signed

// Initialize a validator over the signedDocument in order to extract the master
signature Id
DocumentValidator validator = SignedDocumentValidator.fromDocument(signedDocument);
// Get list of signatures
List<AdvancedSignature> signatures = validator.getSignatures();
// Get Id of the target signature
AdvancedSignature signature = signatures.iterator().next();
String signatureId = signature.getId();
// For XAdES, the XML Id can be used
signatureId = signature.getDAIdentifier();
// Set the Id to parameters
counterSignatureParameters.setSignatureIdToCounterSign(signatureId);

// Initialize a new service for the counter signature creation
// The counter signature will be created in three steps, similarly as a normal
signature
XAdESService service = new XAdESService(commonCertificateVerifier);
// First step is to get toBeSigned, which represents a SignatureValue of the master
signature
ToBeSigned dataToBeCounterSigned = service.getDataToBeCounterSigned(signedDocument,
counterSignatureParameters);
// Second step is to compute the signatureValue on the dataToBeCounterSigned
SignatureValue signatureValue = signingToken.sign(dataToBeCounterSigned,
counterSignatureParameters.getDigestAlgorithm(), privateKey);
// Third step is to create the counter signed signature document
DSSDocument counterSignedSignature = service.counterSignSignature(signedDocument,
counterSignatureParameters, signatureValue);
```

4.9. Extract the original document from a signature

DSS is able to retrieve the original data from a valid signature.

Retrieve the original data from a signed document

```
// We have our signed document, we want to retrieve the original/signed data
DSSDocument signedDocument = new FileDocument("src/test/resources/signature-
pool/signedXmlXadesB.xml");

// We create an instance of DocumentValidator. DSS automatically selects the validator
depending on the
// signature file
SignedDocumentValidator documentValidator =
SignedDocumentValidator.fromDocument(signedDocument);

// We set a certificate verifier. It handles the certificate pool, allows to check the
certificate status,...
documentValidator.setCertificateVerifier(new CommonCertificateVerifier());

// We retrieve the found signatures
List<AdvancedSignature> signatures = documentValidator.getSignatures();

// We select the wanted signature (the first one in our current case)
AdvancedSignature advancedSignature = signatures.get(0);

// We call get original document with the related signature id (DSS unique ID)
List<DSSDocument> originalDocuments =
documentValidator.getOriginalDocuments(advancedSignature.getId());

// We can have one or more original documents depending on the signature (ASiC,
PDF,...)
DSSDocument original = originalDocuments.get(0);

// Save the extracted original document if needed
original.save(targetPath);
```

5. Specificities of signature creation in different signature formats

5.1. XAdES (XML)

5.1.1. Versions support

DSS supports the following XAdES formats :

Table 8. Supported XAdES versions

	B-level	T-level	LT-level	LTA-level
XAdES 1.1.1	✓	✓	✓	✗
XAdES 1.2.2	✓	✓	✓	✗

	B-level	T-level	LT-level	LTA-level
XAdES 1.3.2	✓	✓	✓	✓
XAdES 1.4.1	The format contains qualifying properties for XAdES 1.3.2 LTA level			

The XAdES Profile, as well as a customizable prefixes can be set with following methods :

XAdES formats and prefixes

```
// Allows setting of a XAdES namespace (changes a XAdES format)
// Default : XAdESNamespaces.XADES_132 (produces XAdES 1.3.2)
parameters.setXadesNamespace(XAdESNamespaces.XADES_132);

// Defines an XmlDSig prefix
// Default : XAdESNamespaces.XMLDSIG
parameters.setXmldsigNamespace(new DSSNamespace(XMLSignature.XMLNS, "myPrefix"));

// Defines a XAdES 1.4.1 format prefix
// Default : XAdESNamespaces.XADES_141
parameters.setXades141Namespace(XAdESNamespaces.XADES_141);
```

5.1.2. Reference Transformations

In case of 'Enveloping', 'Enveloped' and 'Internally Detached' signatures, it is possible to apply custom transformations for signing references in order to compute a proper digest result. You can find an example of definition reference transformations below:

Custom transformations definition

```
// Prepare transformations in the proper order
List<DSSTransform> transforms = new ArrayList<>();
DSSTransform envelopedTransform = new EnvelopedSignatureTransform();
transforms.add(envelopedTransform);
DSSTransform canonicalization = new
CanonicalizationTransform(CanonicalizationMethod.EXCLUSIVE_WITH_COMMENTS);
transforms.add(canonicalization);

// Initialize signature parameters
XAdESSignatureParameters parameters = new XAdESSignatureParameters();
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPED);
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_B);

List<DSSReference> references = new ArrayList<>();
// Initialize and configure ds:Reference based on the provided signer document
DSSReference dssReference = new DSSReference();
dssReference.setContents(toSignDocument);
dssReference.setId("r-" + toSignDocument.getName());
dssReference.setTransforms(transforms);
// set empty URI to cover the whole document
dssReference.setUri("");
dssReference.setDigestMethodAlgorithm(DigestAlgorithm.SHA256);
references.add(dssReference);
// set references
parameters.setReferences(references);
```

Current version of DSS supports the following transformations:

- `EnvelopedSignatureTransform` - removes the current `Signature` element from the digest calculation of the reference.

Enveloped Signature Transform does not support parallel signatures!

```
DSSTransform envelopedTransform = new EnvelopedSignatureTransform();
```

- `CanonicalizationTransform` - any canonicalization algorithm that can be used for 'CanonicalizationMethod' can be used as a transform:

```
DSSTransform canonicalization = new
CanonicalizationTransform(CanonicalizationMethod.EXCLUSIVE_WITH_COMMENTS);
```

- `Base64Transform` - this transform is used if the application needs to sign RAW data (binaries, images, audio or other formats). The 'Base64 Transform' is not compatible with the following:
  - Other reference transformations. The reference shall not contain other transforms, the `Base64Transform` shall be the sole element of the reference transformation;

- setEmbedXML(true) - embedded setting cannot be used;
- setManifestSignature(true) - As is apparent from the previous point, Manifest cannot be used with the Base64 Transform as well since it shall also be embedded to the signature;
- ENVELOPED, DETACHED or INTERNALLY DETACHED packaging.

```
DSSTransform base64Transform = new Base64Transform();
```

- `XPathTransform` - allows signing a custom node in a signature or embedded document. DSS contains an additional class `XPathEnvelopedSignatureTransform` allowing to exclude signatures from the digested content (used for Enveloped signatures by default). Additional information about the 'XPath Transform' can be found by the link.

```
DSSTransform envelopedTransform = new XPathTransform("not(ancestor-or-
self::ds:Signature)");
```

- `XPath2FilterTransform` - an alternative to 'XPath Transform'. Additional information about the 'XPath2Filter Transform' can be found by the link. DSS contains an additional class `XPath2FilterEnvelopedSignatureTransform` allowing to exclude signatures from the digest calculation.

In DSS, the XPath-2-Filter transform is used by default for ENVELOPED signature packaging.

```
DSSTransform envelopedTransform = new
XPath2FilterTransform("descendant::ds:Signature", "subtract");
```

- `XsltTransform` - This transform requires a 'org.w3.dom.Document' as an input, compatible with the normative XSLT Specification. Must be a sole transform.

All transformations, except Base64, can be applied only to XML objects.

5.1.3. Specific schema version

Some signatures may have been created with an older version of the XAdES format using different schema definition. To take into account the validation of such signatures the `interface eu.europa.esig.dss.xades.definition.XAdESPaths` was created. This interface allows providing the different needed XPath expressions which are used to explore the elements of the signature. The DSS framework proposes 3 implementations :

- XAdES132Paths (XAdES 1.3.2 / 1.4.1)
- XAdES122Paths (XAdES 1.2.2)
- XAdES111Paths (XAdES 1.1.1)

By default, all XAdES are supported and DSS loads/parses all versions of XAdES. It is possible to

restrict to only one version of XAdES with the following code :

Customize the supported XAdES version(s) at the validation

```
XMLDocumentValidator xmlDocumentValidator = new XMLDocumentValidator(xmlDocument);
xmlDocumentValidator.setCertificateVerifier(certificateVerifier);

// Restrict the current XMLDocumentValidator to XAdES 1.3.2 (and 1.4.1 for
// archival timestamps)
List<XAdESPaths> xadesPathsHolders = xmlDocumentValidator.getXAdESPathsHolder();
xadesPathsHolders.clear();
xadesPathsHolders.add(new XAdES132Paths());

Reports reports = xmlDocumentValidator.validateDocument();
```

5.2. CAdES signature (CMS)

To familiarize 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. For an extensive example see section CAdES of the Annex.

5.3. PAdES signature (PDF)

The standard ISO 32000-1 (cf. [R06]) allows defining a file format for portable electronic documents (PDF). The standard defines the PDF version 1.7 of Adobe Systems. Concerning the advanced electronic signature, the PAdES standard is used (cf. [R03]).

The DSS implementation supports main operations for PAdES signatures:

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

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

An example of code to perform a `PAdES-BASELINE-B` type signature can be found in section PAdES of the Annex.

5.3.1. PAdES Visible Signature

The framework also allows creation of PDF files with visible signatures as specified in ETSI EN 319 142 (cf. [R03]). In the `PAdESSignatureParameters` object, there is a special attribute named `SignatureImageParameters`. This parameter allows you to customize the visual signature (with text,

with image or with image and text). An example of code to perform a PAdES-BASELINE-B type signature with a visible signature can be found in section PAdES Visible Signature of the Annex.

Additionally, DSS also allows you to insert a visible signature to an existing field. This is illustrated in section PAdES Visible Signature of the Annex.

In case of placing an image or text to an existing field, the visible signature will fill out the whole available area of the field.

5.3.1.1. Visible signature parameters (image and text)

This chapter introduces existing parameters for creation of visible signatures with DSS. DSS has three implementations for visible signature drawing:

- **OpenPDF (iText)** - supports separate image and text drawing;
- **PDFBox Default** - supports separate image and text drawing, as well as a joint drawing of image and text together. It transforms text to an image;
- **PDFBox Native** - supports separate image and text drawing, as well as a joint drawing of image and text together. It prints text in a native way, that increases quality of the produced signature.

5.3.1.1.1. Positioning

DSS provides a set of functions allowing to place the signature field on a specific place in the PDF page. For an illustrative example see section Positioning in the Annex.

5.3.1.1.2. Dimensions

DSS framework provides a set of functions to manage the signature field size. See section Dimensions in the Annex for a concrete example.

5.3.1.1.3. Text Parameters

The available implementation allows placing of a visible text in a signature field. See section Text Parameters in the Annex for a concrete example.

5.3.1.1.4. Text and image combination

DSS provides a set of functions to align a text with respect to an image. The parameters must be applied to a 'SignatureImageTextParameters' object. See section Text and image combination in the Annex for a concrete example.

5.3.1.2. Fonts usage

To customize a text representation a custom font can be defined. The font must be added as an instance of the `DSSFont` interface to a `SignatureImageTextParameters` object.

DSS provides the following common implementations for the `DSSFont` interface:

- `DSSFileFont` for use of physical fonts, which must be embedded to the produced PDF document. To create an instance of the class, you must pass to a `DSSFileFont` constructor an object of `DSSDocument` type or an InputStream of the font file;

- `DSSJavaFont` for use of logical fonts (default Java fonts). The logical Java fonts allow you to significantly reduce the document size, because these fonts cannot be embedded to the final PDF document. Be aware that, because of the latter fact, use of logical fonts does not allow producing PDF documents satisfying the PDF/A standard. To create an instance of this class, you should pass as an input a `java.awt.Font` object or target font parameters (name, style, size).

Logical fonts may have different implementations depending on the PAdES Visible signature service or Operating System (OS) used. Keep this in mind when switching from an implementation or system environment to another.

Additionally, there are implementation-dependent classes:

- `ITextNativeFont` to be used with `ITextSignatureDrawerFactory`;
- `PdfBoxNativeFont` to be used with `PdfBoxNativeObjectFactory`.

You can find an example of how to create a custom font for a physical font, for a logical font and for a native font in section Fonts usage.

By default, DSS uses a Google font : 'PT Serif Regular' (its physical implementation).

The 'Native PDFBox Drawer' implementation supports only one of the following fonts: SERIF, SANS-SERIF, MONOSPACED, DIALOG and DIALOG_INPUT.

5.4. ISO 32000-1 PDF signature (PKCS#7)

The ISO 32000-1 standard gives a specification for the creation of electronic signatures in the PDF format. However, these signatures are not in the AdES format defined by ETSI.

The Public Key Cryptographic Standard Number 7 (PKCS#7) is a common format of a signature included in a PDF which has been created by Adobe PDF Reader. The PKCS#7 signature format is described in the ISO 32000-1 standard (cf. [R06]) and must conform to the RFC 2315 (cf. [R15]). To familiarize yourself with this type of signature you can read these documents.

DSS only supports AdES formats (e.g. PAdES for PDF) and thus does not support signature creation for PKCS#7. DSS only provides a limited support for the augmentation and validation of PKCS#7 signatures.

There is no specific validation standard for PKCS#7 nor any augmentation formats in the ISO 32000-1 standard. Thus, DSS validates and augments PKCS#7 according to rules defined in ETSI standards. It validates PKCS#7 signatures according to the AdES validation algorithm, and it adds the same components as when augmenting PADES-BASELINE-B to PAdES-BASELINE-T.

5.5. JAdES signature (JWS)

DSS includes a possibility of creation and validation of JSON Advanced signatures (JAdES).

The JSON format for AdES Signatures (cf. [R05]) represents an extension of JSON Web Signatures (JWS) as specified in IETF RFC 7515.

A typical example of a JAdES signature creation can be found in section JAdES of the Annex.

The specific parameters for JAdES signature are described in the next sections.

5.5.1. JWS Serialization type

A JWS signature can be represented in different forms which are supported by the JAdES standard as well:

- `COMPACT_SERIALIZATION` represents a compact, URL-safe serialization. It has no JWS Unprotected Header, therefore only `JAdES-BASELINE-B` level is possible with this format.
- `JSON_SERIALIZATION` represents a JSON object with a collection of signatures inside the `'signatures'` header that allows parallel signing. It allows `JAdES-BASELINE-T/-LT/-LTA` signature augmentation levels.
- `FLATTENED_JSON_SERIALIZATION` represents a JSON object with a single signature container. It allows `JAdES-BASELINE-T/-LT/-LTA` signature augmentation levels.

JWS Serialization type usage

```
// Choose the form of the signature (COMPACT_SERIALIZATION, JSON_SERIALIZATION,
FLATTENED_JSON_SERIALIZATION)
parameters.setJwsSerializationType(JWSSerializationType.COMPACT_SERIALIZATION);
```

5.5.2. SigD header parameter

JAdES signatures allow two types of JWS Payload (signed data) inclusion: `ENVELOPING` and `DETACHED`.

5.5.2.1. Enveloping packaging

With `ENVELOPING` packaging the JWS Payload is enveloped into the JAdES Signature. The type only allows signing one document.

5.5.2.2. Detached packaging

A simple JWS signature allows a `DETACHED` packaging by omitting the JWS Payload in the created signature. For the validation process the detached content shall be provided and it is treated in the same way as if it were attached.

To create such a signature, the parameter `SigDMechanism.NO_SIG_D` shall be set. The solution allows signing of only one document.

The JAdES standard [R05] provides a possibility for signing multiple documents within one signature in a detached way.

The following mechanisms are possible:

- `HTTP_HEADERS` is used to sign an HTTP request. The signature may explicitly sign several HTTP headers (represented by the class `HTTPHeader`), as well as the HTTP message body (see the `HTTPHeaderDigest` class).

Configuration for signing with detached mechanism HttpHeaders

```
// Set Detached packaging
parameters.setSignaturePackaging(SignaturePackaging.DETACHED);
// Set Mechanism HttpHeaders for 'sigD' header
parameters.setSigDMechanism(SigDMechanism.HTTP_HEADERS);
// The HttpHeaders mechanism shall be used with unencoded JWS payload ("b64"="false")
parameters.setBase64UrlEncodedPayload(false);
// Create a list of headers to be signed
List<DSSDocument> documentsToSign = new ArrayList<>();
documentsToSign.add(new HTTPHeader("content-type", "application/json"));
documentsToSign.add(new HTTPHeader("x-example", "HTTP Headers Example"));
documentsToSign.add(new HTTPHeader("x-example", "Duplicated Header"));
// Add a document representing the HTTP message body (optional)
// Requires the message body content + digest algorithm to compute the hash to be
signed
documentsToSign.add(new HTTPHeaderDigest(toSignDocument, DigestAlgorithm.SHA1));
```

- `OBJECT_ID_BY_URI` can be used for signing of multiple documents. The signed files are dereferenced by URIs and their content is concatenated for generation of the JWS Payload.
- `OBJECT_ID_BY_URI_HASH` similarly provides a possibility to sign multiple documents, by signing the computed digests of the original documents. The JWS Payload for this format stays empty.

Configuration for signing with detached mechanism ObjectIdByURIHash

```
parameters.setSignaturePackaging(SignaturePackaging.DETACHED);
parameters.setSigDMechanism(SigDMechanism.OBJECT_ID_BY_URI_HASH);
// Prepare the documents to be signed
documentsToBeSigned = new ArrayList<>();
documentsToBeSigned.add(new FileDocument("src/main/resources/hello-world.pdf"));
documentsToBeSigned.add(new FileDocument("src/main/resources/xml_example.xml"));
```

5.5.3. Base64Url encoding

The `Base64Url` represents a Base64 encoded format with URI safe alphabet (see RFC 4648).

JAdES signatures (as well as JWS) force some values to be Base64Url-encoded, while providing a possibility to customize the format for some of them.

DSS provides options to configure encoding for the following elements:

- JWS Payload can be represented as Base64Url encoded octets (by default), and can be present in its initial form (with the protected header `b64` set to `false`).

Use unencoded JWS Payload

```
parameters.setBase64UrlEncodedPayload(false);
```

- The components of the unsigned header `'etsiU'` can occur either as Base64Url encoded strings

(by default), or as clear JSON objects.

All components inside the `'etsiU'` header shall be present in the same form (Base64Url encoded or as clear JSON).

The current version of DSS does not allow `JAdES-BASELINE-LTA` level creation for `'etsiU'` components in their clear JSON representation.

Represent EtsiU components as clear JSON instances

```
parameters.setBase64UrlEncodedEtsiUComponents(false);
```

5.6. ASiC signature (containers)

The ETSI EN 319 162 standard (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. It is an alternative to the packaging types presented in section Signature packaging.

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:

- **ASiC-S** is a simple container that allows you to associate one or more signatures with a single data element (zip-container). 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;
- **ASiC-E** is an extended container that includes multiple data objects. Each data object may be signed by one or more signatures whose structure is similar to ASiC-S. This second type of container is compatible with OCF, UCF and ODF formats.

DSS framework has some restrictions on the containers you can generate, depending on the input file. If the input file is already an ASiC container, the output container must be the same type of container based on the same type of signature. If the input is any other file, the output does not have any restriction.

Table 9. ASiC containers

Input	Output
ASiC-S CAdES	ASiC-S CAdES
ASiC-S XAdES	ASiC-S XAdES
ASiC-E CAdES	ASiC-E CAdES
ASiC-E XAdES	ASiC-E XAdES

Input	Output
Binary	ASiC-S CAdES, ASiC-S XAdES, ASiC-E CAdES, ASiC-E XAdES

Typical examples of the source code for signing a document using `ASiC-S` based on `XAdES-BASELINE-B` and `ASiC-E` based on `XAdES-BASELINE-B` can be found in sections ASiC-S and ASiC-E respectively.

You need to pass only few parameters to the service. Other parameters, although they are positioned, will be overwritten by the internal implementation of the service. Therefore, the obtained signature is always based on the DETACHED packaging no matter the packaging that was specified.

It is also possible with the DSS framework to make an augmentation of an ASiC container to the levels `BASELINE-T`, `BASELINE-LT` or `BASELINE-LTA`, respectively for XAdES and CAdES formats.

6. Revocation data management

Revocation data management is an essential part in the lifecycle of a digital certificate and thus of a digital signature too.

6.1. Tokens and sources

DSS provides utilities for processing and validation of both CRL and OCSP tokens, containing a revocation information about a certificate (see CRLs and OCSP for more information on CRLs and OCSP)

For every certificate, the validity has to be checked via CRL or OCSP responses. The information may originate from different CRL or OCSP sources. For easing the usage of such sources, DSS implements a `CRLSource` and `OCSPSource` interfaces (which inherit from `RevocationSource`), which offer a generic and uniform way of accessing CRL and OCSP sources, respectively. Furthermore, a caching mechanism can be easily attached to those sources, optimizing the access time to revocation information by reducing network connections to online servers.

The interface `CRLSource` defines the method which returns a `CRLToken` for the given certificate/issuer certificate couple:

CRLSource usage

```
CRLToken crlToken = crlSource.getRevocationToken(certificateToken,
issuerCertificateToken);
```

The interface `OCSPSource` defines the method which returns `OCSPToken` for the given certificate/issuer certificate couple:

OCSPSource usage

```
OCSPToken ocspToken = ocspSource.getRevocationToken(certificateToken,
issuerCertificateToken);
```

We use these classes during the certificate validation process through `validationContext` object (based on `ValidationContext` class) which is a "cache" for a validation request that contains every object required for a complete validation process. This object in turn instantiates a "verifier" based on `RevocationDataLoadingStrategy` class whose role is to fetch revocation data by querying an OCSP or CRL source in the defined order and return the succeeded result (see Revocation data loading strategy for more details).

In general, we can distinguish three main sources:

- Offline sources (`OfflineRevocationSource`);
- Online sources (`OnlineRevocationSource`);
- Sources with the cache mechanism.

As well as a list of sources (`ListRevocationSource`) with a collection of several sources.

6.2. Caching

The above-mentioned class allows caching of CRL and OCSP responses to a user-chosen source. By default, DSS provides a JDBC based implementation for this class, but other implementations also can be created. The class contains a complete set of functions to save revocation data to a database, extract, update and remove it.
Furthermore, the `RepositoryRevocationSource` allows the implementer to define a backup revocation source, for the case if the database does not contain the certificate's revocation data yet.

List of cached Revocation sources implemented in DSS:

- `JdbcRevocationSource`
  - `JdbcCacheCRLSource`
  - `JdbcCacheOCSPSource`

A database table shall be initialized before you start working with the cached revocation repository.

6.2.1. CRL

An example for JdbcCacheCRLSource :

JdbcCacheCRLSource usage

```
// Creates an instance of JdbcCacheCRLSource
JdbcCacheCRLSource cacheCRLSource = new JdbcCacheCRLSource();

// Initialize the JdbcCacheConnector
JdbcCacheConnector jdbcCacheConnector = new JdbcCacheConnector(dataSource);

// Set the JdbcCacheConnector
cacheCRLSource.setJdbcCacheConnector(jdbcCacheConnector);

// Allows definition of an alternative dataLoader to be used to access a revocation
// from online sources if a requested revocation is not present in the repository or
has been expired (see below).
cacheCRLSource.setProxySource(onlineCRLSource);

// All setters accept values in seconds
Long oneWeek = (long) (60 * 60 * 24 * 7); // seconds * minutes * hours * days

// If "nextUpdate" field is not defined for a revocation token, the value of
"defaultNextUpdateDelay"
// will be used in order to determine when a new revocation data should be requested.
// If the current time is not beyond the "thisUpdate" time + "defaultNextUpdateDelay",

// then a revocation data will be retrieved from the repository source, otherwise a
new revocation data
// will be requested from a proxiedSource.
// Default : null (a new revocation data will be requested of "nestUpdate" field is
not defined).
cacheCRLSource.setDefaultNextUpdateDelay(oneWeek);

// Defines a custom maximum possible nextUpdate delay. Allows limiting of a time
interval
// from "thisUpdate" to "nextUpdate" defined in a revocation data.
// Default : null (not specified, the "nextUpdate" value provided in a revocation is
used).
cacheCRLSource.setMaxNextUpdateDelay(oneWeek); // force refresh every week (eg : ARL)

// Defines if a revocation should be removed on its expiration.
// Default : true (removes revocation from a repository if expired).
cacheCRLSource.setRemoveExpired(true);

// Creates an SQL table
cacheCRLSource.initTable();

// Extract CRL for a certificate
CRLToken crlRevocationToken = cacheCRLSource.getRevocationToken(certificateToken,
issuerCertificateToken);
```

6.2.2. OCSP

An example for JdbcCacheOCSPSource :

JdbcCacheOCSPSource usage

```
// Creates an instance of JdbcCacheOCSPSource
JdbcCacheOCSPSource cacheOCSPSource = new JdbcCacheOCSPSource();

// Initialize the JdbcCacheConnector
JdbcCacheConnector jdbcCacheConnector = new JdbcCacheConnector(dataSource);

// Set the JdbcCacheConnector
cacheOCSPSource.setJdbcCacheConnector(jdbcCacheConnector);

// Allows definition of an alternative dataLoader to be used to access a revocation
// from online sources if a requested revocation is not present in the repository or
has been expired (see below).
cacheOCSPSource.setProxySource(onlineOCSPSource);

// All setters accept values in seconds
Long threeMinutes = (long) (60 * 3); // seconds * minutes

// If "nextUpdate" field is not defined for a revocation token, the value of
"defaultNextUpdateDelay"
// will be used in order to determine when a new revocation data should be requested.
// If the current time is not beyond the "thisUpdate" time + "defaultNextUpdateDelay",
// then a revocation data will be retrieved from the repository source, otherwise a
new revocation data
// will be requested from a proxiedSource.
// Default : null (a new revocation data will be requested of "nestUpdate" field is
not defined).
cacheOCSPSource.setDefaultNextUpdateDelay(threeMinutes);

// Creates an SQL table
cacheOCSPSource.initTable();

// Extract OCSP for a certificate
OCSPToken ocspRevocationToken = cacheOCSPSource.getRevocationToken(certificateToken,
issuerCertificateToken);
```

6.3. Online fetching

DSS provides utilities for online fetching of revocation data from remote sources, during the signature augmentation and validation processes.

By default, revocation data are not fetched from untrusted sources. In other words, revocation data are not fetched when the prospective certificate chain does not contain a trust anchor.

6.3.1. CRL

This is a representation of an Online CRL repository. This implementation will download the CRL using the protocol referenced in the certificate (e.g. HTTP, LDAP). The URIs of CRL server will be extracted from this property (OID value: 1.3.6.1.5.5.7.48.1.3).

It allows the following configuration :

OnlineCRLSource usage

```
// Instantiates a new OnlineCRLSource
OnlineCRLSource onlineCRLSource = new OnlineCRLSource();

// Allows setting an implementation of `DataLoader` interface,
// processing a querying of a remote revocation server.
// `CommonsDataLoader` instance is used by default.
onlineCRLSource.setDataLoader(new CommonsDataLoader());

// Sets a preferred protocol that will be used for obtaining a CRL.
// E.g. for a list of urls with protocols HTTP, LDAP and FTP, with a defined preferred
protocol as FTP,
// the FTP url will be called first, and in case of an unsuccessful result other url
calls will follow.
// Default : null (urls will be called in a provided order).
onlineCRLSource.setPreferredProtocol(Protocol.FTP);
```

6.3.2. OCSP

This is a representation of an Online OCSP repository. This implementation will access the OCSP responder using the access point defined in the certificate. Note that the certificate's Authority Information Access (AIA) extension is used to find issuer's resources location like Online Certificate Status Protocol (OCSP). The URIs of OCSP server will be extracted from this property (OID value: 1.3.6.1.5.5.7.48.1).

It allows the following configuration :

OnlineOCSPSource usage

```
// Instantiates a new OnlineOCSPSource object
OnlineOCSPSource onlineOCSPSource = new OnlineOCSPSource();

// Allows setting an implementation of the `DataLoader` interface,
// processing a querying of a remote revocation server.
// `CommonsDataLoader` instance is used by default.
onlineOCSPSource.setDataLoader(new OCSPDataLoader());

// Defines an arbitrary integer used in OCSP source querying in order to prevent a
replay attack.
// Default : null (not used by default).
onlineOCSPSource.setNonceSource(new SecureRandomNonceSource());

// Defines a DigestAlgorithm being used to generate a CertificateID in order to
complete an OCSP request.
// OCSP servers supporting multiple hash functions may produce a revocation response
// with a digest algorithm depending on the provided CertificateID's algorithm.
// Default : SHA1 (as a mandatory requirement to be implemented by OCSP servers. See
RFC 5019).
onlineOCSPSource.setCertIDDigestAlgorithm(DigestAlgorithm.SHA1);
```

6.4. Other implementations of CRL and OCSP Sources

Other revocation sources find the status of a certificate either from a list stored locally or using the information contained in the advanced signature or online way. Here is the list of sources already implemented in the DSS framework:

- CRL sources:
  - `OfflineCRLSource` : This class implements the `OfflineRevocationSource` and retrieves the revocation data from extracted information. The code is common for all signature formats and CRL contents are injected by its sub-classes:
    - `CMSCRLSource` : Extracts CRLs and CRL references from a CMS Signed Data:
      - `CAdESCRLSource` : Sub-class of `CMSCRLSource` for a CAdES Signature;
      - `TimestampCRLSource`: Sub-class of `CMSCRLSource` for a Timestamp token (RFC 3161);
    - `PAdESCRLSource` : Extracts CRLs and CRL references from a PAdES signature.
    - `XAdESCRLSource` : Extracts CRLs and CRL references from a XAdES signature.
    - `ExternalResourcesCRLSource` : A class that can instantiate a list of certificate revocation lists from a directory where the individual lists should be.
  - `OnlineCRLSource` : Retrieves CRL files from online sources with the CRL Distribution Points information from the certificate.
  - `JdbcCacheCrlSource` : Implementation of the `JdbcRevocationSource`. This implementation allows storage of valid CRL entries to a defined `DataSource'` and retrieve them locally.

- OCSP sources:
  - `OfflineOCSPSource` : This class implements the `OfflineRevocationSource` and retrieves the revocation data from extracted information. The code is common for all signature formats and OCSP responses are injected by its sub-classes:
    - `CMSOCSPSource` : Extracts OCSP responses and OCSP references from a CMS Signed Data:
      - `CAdESOCSPSource` : Sub-class of `CMSOCSPSource` for a CAdES Signature;
      - `TimestampOCSPSource`: Sub-class of `CMSOCSPSource` for a Timestamp token (RFC 3161);
    - `PAdESOCSPSource` : Extracts OCSP responses and OCSP references from a PAdES signature.
    - `XAdESOCSPSource` : Extracts OCSP responses and OCSP references from a XAdES signature.
      - `ExternalResourcesOCSPSource` : A class that can instantiate a list of OCSPToken from a directory where should be the individual DER Encoded X509 certificates files.
  - `OnlineOCSPSource` : Retrieves OCSP responses from online source.
  - `JdbcCacheOcspSource` : Implementation of the `JdbcRevocationSource`. This implementation allows storage of valid OCSP entries to a defined `DataSource` and retrieve them locally.

6.5. Revocation data loading strategy

Since version 5.9, DSS allows the use of a `RevocationDataLoadingStrategy`. The latter defines logic for loading OCSP or CRL data. Two strategies are available in the core package of DSS:

- `OCSPFirstRevocationDataLoadingStrategy`: loads OCSP first, if not available or the response is invalid, then tries to load CRL and returns the first succeeded result. This strategy is used by default for revocation retrieving.
- `CRLFirstRevocationDataLoadingStrategy`: fetches firstly CRL response, if not available, tries OCSP and returns the first succeeded result.

See section CertificateVerifier configuration for an example of how to customize a used `RevocationDataLoadingStrategy`.

Using an OCSP first (i.e. `OCSPFirstRevocationDataLoadingStrategy`) has some advantages:

1. There is a potential benefit of freshness compared to using CRLs: in case the OSCP sends queries to a database that is updated in real time or every x hours, the information fetched by the OCSP is more recent (thisUpdate field) than the information contained in the CRL, given that the CRL is built from that database. In the worst case, the OCSP uses the data contained in the CRL and they both have the same thisUpdate field. The information fetched by an OCSP will never be older than the one obtained from a CRL.
2. The certificate that signed the OCSP response might contain an OCSPNoCheck extension. This extension indicates that the revocation data of the certificate that contains the public key linked to the private key that was used to sign the OCSP response does not need to be checked.
3. Getting a response takes less time and less memory space for OCSP than with CRLs. A CRL can take a lot of space and a long time to be downloaded due its big size. There is also no obligation to sort CRLs according to serial number which means that the whole list needs to be browsed until finding the searched serial number.

7. Signature Validation

7.1. Validation of a certificate

The signature validation starts from a validation of a certificate chain (cf. Certificate Chain and Certification Path Validation). For a given certificate, the framework builds a certificate path until a known trust anchor (trusted list, keystore,...), validates each found certificate (OCSP / CRL) and determines its European "qualification".

To determine the certificate qualification, DSS follows the standard ETSI TS 119 615 ([R14]). It analyses the certificate properties (QCStatements, Certificate Policies, etc.) and applies possible overrules from the related trusted list ("catched" qualifiers from a trust service). More information about qualifiers can be found in the standard ETSI TS 119 612 ([R11]).

DSS always computes the status at 2 different times: certificate issuance and signing/validation time. The certificate qualification can evolve in time, its status is not immutable (e.g.: a trust service provider can lose the granted status). The eIDAS regulation ([R12]) clearly defines these different times in the Article 32 and related Annex I.

Validate a certificate and retrieve its qualification level

```
// Firstly, we load the certificate to be validated
CertificateToken token = DSSUtils.loadCertificate(new
File("src/main/resources/keystore/ec.europa.eu.1.cer"));

// We need a certificate verifier and configure it  (see specific chapter about the
CertificateVerifier configuration)
CertificateVerifier cv = new CommonCertificateVerifier();

// We create an instance of the CertificateValidator with the certificate
CertificateValidator validator = CertificateValidator.fromCertificate(token);
validator.setCertificateVerifier(cv);

// Allows specifying which tokens need to be extracted in the diagnostic data
(Base64).
// Default : NONE)
validator.setTokenExtractionStrategy(TokenExtractionStrategy.EXTRACT_CERTIFICATES_AND_
REVOCATION_DATA);

// We execute the validation
CertificateReports certificateReports = validator.validate();

// We have 3 reports
// The diagnostic data which contains all used and static data
DiagnosticData diagnosticData = certificateReports.getDiagnosticData();

// The detailed report which is the result of the process of the diagnostic data and
the validation policy
DetailedReport detailedReport = certificateReports.getDetailedReport();

// The simple report is a summary of the detailed report or diagnostic data (more
user-friendly)
SimpleCertificateReport simpleReport = certificateReports.getSimpleReport();
```

7.1.1. Trust anchor configuration from a certificate store

Trust anchors are an essential part of the validation process of a signature as described in section Trust Anchors and Trust Stores. DSS allows configuring of various trusted certificate source(s). These sources can be defined from a TrustStore (kind of keystore which only contains certificates), a trusted list or a list of trusted lists.

7.1.1.1. Trust store initialization

If you have a collection of certificates to trust, the easier way to provide them to DSS it to use a KeyStore / TrustStore (cf. Trust Stores).

Trust anchor initialization from a Trust Store

```
public CertificateSource trustStoreSource() throws IOException {
    KeyStoreCertificateSource keystore = new KeyStoreCertificateSource(new
File("src/main/resources/keystore.p12"), "PKCS12", getPassword());

    CommonTrustedCertificateSource trustedCertificateSource = new
CommonTrustedCertificateSource();
    trustedCertificateSource.importAsTrusted(keystore);

    // Optionally, certificates can also be directly added

trustedCertificateSource.addCertificate(DSSUtils.loadCertificateFromBase64EncodedStrin
g(

"MIIC9TCCAd2gAwIBAgIBAjANBgkqhkiG9w0BAQUFADArMQswCQYDVQQGEwJBQTEMMAoGA1UEChMDRFNTMQ4wD
AYDVQQDEwVJQ0EgQTAeFw0xMzEyMDIxNzMzMTBaFw0xNTEyMDIxNzMzMTBaMDAxCzAJBgNVBAYTAkFBMQwwCgY
DVQQKEwNEU1MxEzARBgNVBAMTCnVzZXIgQSBSU0EwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAJUHHAphm
SDdQ1t62tppK+dLTANsE2nAj+HCpasS3ohlBsrhteRsvTAbrDyIzCmTYWu/nVI4TGvbzBESwV/QitlkoMLpYFw
32MIBf2DLmECzGJ3vm5haw6u8S9quR1h8Vu7QWd+5KMabZuR+j91RiSuoY0xS2ZQxJw1vhvW9hRYjAgMBAAGjg
aIwgZ8wCQYDVR0TBAIwADAdBgNVHQ4EFgQU9ESnTWfwg13c3LQZzqqwibY5WVYwUwYDVR0jBEwwSoAUIO1CDsB
SUcEoFZxKaWf1PAL1U+uhL6QtMCsxDDAKBgNVBAoTA0RTUzELMAkGA1UEBhMCQUExDjAMBgNVBAMTBVJDQSBBg
gEBMAsGA1UdDwQEAwIHgDARBgNVHSAECjAIMAYGBFUdIAAwDQYJKoZIhvcNAQEFBQADggEBAGnhhnoyVUhDnr/
BSbZ/uWfSuwzFPG+2V9K6WxdIaaXOORFGIdFwGlAwA/Qzpq9snfBxuTkAykxq0uEDhHTj0qXxWRjQ+Dop/Drmc
coF/zDvgGusyY1YXaABd/kc3IYt7ns7z3tpiqIz4A7a/UHplBRXfqjyaZurZuJQRaSdxh6CNhdEUiUBxkbb1Sd
MjuOgjzSDjcDjcegjvDquMKdDetvtu2Qh4ConBBo3fUImwiFRWnbudS5H2HE18ikC7gY/QIuNr7USf1PNyUgcG
2g31cMtemj7UTBHZ2V/jPf7ZXqwfnVSaYkNvM3weAI6R3PI0STjdxN6a9qjt9xld40YEdw="));

    return trustedCertificateSource;
}
```

To generate the trust store, there's an utility class CreateKeyStoreApp in the `dss-cookbook` module.

7.1.1.2. Trusted List Certificate Source

In several countries, a list of Trust Service Providers (TSP) is published. This list is usually published in a machine processable format (XML) and sometimes in a human-readable format (PDF). A standard (ETSI TS 119 612 [R11]) exists with the specifications for the XML format.

DSS contains all needed resources to download, parse, validate and interpret the trusted list contents. In DSS, it is possible to configure one or more independent trusted list(s) (aka not linked to a list of trusted lists) and/or one or more list of trusted lists.

If you want to collect your trusted certificates from trusted list(s), the `TrustedListsCertificateSource` is required. The trusted list(s) loading can require some time (connection time-out, xml parsing, xml validation, etc.). This process is usually executed in background. An instance of `TrustedListsCertificateSource` needs to be created. That will be synchronized with the `TLValidationJob`.

Trusted List Certificate Source

```
public CertificateSource trustedListSource() {
    return new TrustedListsCertificateSource();
}
```

7.1.1.3. Multiple Trusted Sources usage

DSS provides a possibility to use multiple trusted certificate sources at one time. An example of the configuration is provided below:

Multiple trusted certificate sources usage

```
CertificateVerifier cv = new CommonCertificateVerifier();
cv.setTrustedCertSources(trustStoreSource(), trustedListSource());
```

7.1.2. Certificate chain in DSS

The validation of a certificate requires the access to some other certificates from multiple sources like trusted lists, trust store, the signature itself: certificates can be contained inside any other source. Within the framework, an X509 certificate is wrapped through the class:

- `eu.europa.esig.dss.model.x509.CertificateToken`

This encapsulation helps make certificate handling more suited to the needs of the validation in the context of trust. The framework associates two internal identifiers to the certificate: the DSS Id based on the certificate binary (unique for each certificate) and the Entity Id based on its public key (common to cross-signed certificates).

Certificate tokens are grouped into sources. A certificate token can be declared in several sources. The class that models a source is called:

- `eu.europa.esig.dss.spi.x509.CertificateSource`

This class stores all extracted/injected certificates for a specific source (Signature, OCSP Response, Trust store, Trusted-list, etc.). All source types are specified in the enumeration:

- `eu.europa.esig.dss.enumerations.CertificateSourceType`

This information is used, for example, to distinguish between the certificate from a trusted source and the others. A source has one and only one type, but a certificate token can be found in multiple sources. The DSS framework supplies some standard implementations, but also gives the possibility to implement custom solutions. Among the standard solutions you can find:

- `eu.europa.esig.dss.spi.x509.CommonCertificateSource`

This is the superclass of almost of the certificate sources. It stores the extracted certificates and implements the common methods from the `CertificateSource` to retrieve certificate(s) by subject, public key, subject key identifier (ski), etc.

It also exposes the method `CommonCertificateSource#addCertificate` which gives the possibility to add manually any `CertificateToken` as a part of this source.

- `eu.europa.esig.dss.spi.x509.CommonTrustedCertificateSource`

The `CommonTrustedCertificateSource` is a certificate source for trusted certificates. All added certificates are marked as trust anchors and no revocation data are required for these certificates.

- `eu.europa.esig.dss.validation.SignatureCertificateSource`

This class and its sub-classes are used to extract and collect certificates from signatures / timestamps. It also has methods to retrieve certificates / certificate references by their origin (e.g. SigningCertificate attribute, DSS Dictionary, etc.).

- `eu.europa.esig.dss.spi.tsl.TrustedListsCertificateSource`

Certificates coming from the list of Trusted Lists. This class inherits of `CommonTrustedCertificateSource` and gives the mechanism to define the set of trusted certificates (trust anchors). They are used in the validation process to decide if the prospective certificate chain has a trust anchor. See section Configuration of TL validation job to get more information about trusted lists loading (e.g. EU Trusted List).

- `eu.europa.esig.dss.spi.x509.ListCertificateSource`

This class follows the composite design pattern with a list of CertificateSources. That is used in the validation to retrieve all sources from the signatures / timestamps / revocation data / trusted lists / etc. It contains some methods which check over all sources to retrieve certificates or verify if a certificate is trusted.

7.1.2.1. Retrieving certificates by AIA

In case when intermediate certificates are not present within a signature document, nor in trusted/adjunct sources, a certificate chain can still be built using AIA URL obtained from a certificate (See Certificate Chain and Certification Path Validation).

To use AIA URLs DSS provides the interface `AIASource` with the following implementations:

- `DefaultAIASource` - the default implementation used in DSS, allowing retrieving of certificates by AIA URL from online sources. The class allows configuring a list of accepted protocols to be used for remote requests.
- `JdbcCacheAIASource` - a cache AIA Source, allowing storing and accessing of certificates from a JDBC database.

An example of `DefaultAIASource` configuration can be found below:

DefaultAIASource usage

```
Set<CertificateToken> certificates = aiaSource.getCertificatesByAIA(certificateToken);
```

7.1.3. Revocation data handling

For information on how revocation of data is handled, see chapter Revocation data management.

7.1.3.1. Revocation freshness

The revocation freshness constraint (RFC) is a time interval indicating that the validation accepts CRLs that were emitted at a point in time after the validation time minus the RFC: `valTime - RFC < CRL.thisUpdate`.

If the RFC is respected by a CRL then that CRL can be used. Otherwise, the CRL shall be rejected and shall not be used to determine whether the certificate is revoked or not. Another CRL can be searched online. If no CRL respecting the RFC is found, then it cannot be determined whether the certificate is valid, and it is thus not possible to determine whether the signature is valid.

In case of a signature with a `BASELINE-T` level, the validation time can be replaced by the best-signature-time when checking the constraint. Revocation data should be issued after the best-signature-time, provided by a signature timestamp.

In case of a `BASELINE-B` level, there is no timestamp among the unsigned attributes. If the RFC is equal to `0` then the validation time needs to be smaller than the `CRL.thisUpdate`. This means that the revocation data needs to have been issued after the validation process is concluded which is not possible.

According to the ETSI TS 119 172-4 (cf. [R10]) standard, the RFC shall be set to `0` (zero). If DSS had had an RFC equal to `0` then it would invalidate all B-level signatures without a signature timestamp. Therefore, revocation freshness is not checked in DSS by default. The validation level of the check is set to `IGNORE`, meaning users are shown that the check exists, but it is not executed in the validation process.

As DSS allows using a custom validation policy (see AdES validation constraints/policy), it is possible to change the validation level of the check and to define a revocation freshness constraint. The validation level and time interval are defined within the `<RevocationFreshness />` constraint.

For example applying of `<RevocationFreshness />` constraint to a signing-certificate of a signature:

```
<SignatureConstraints>
    ...
    <BasicSignatureConstraints>
        ...
        <SigningCertificate>
            ...
            <RevocationFreshness Level="FAIL" Unit="DAYS" Value="2" />
            ...
        </SigningCertificate>
        ...
    </BasicSignatureConstraints>
    ...
</SignatureConstraints>
```

With the following policy, the `RevocationFreshness` check of the signing certificate of the signature will fail in case the revocation data is older than 2 days.

7.1.4. CertificateVerifier configuration

The `CertificateVerifier` (with default implementation `CommonCertificateVerifier`) determines how DSS accesses the external resources and how it should react in some occasions. This object is used to provide the following sources of information and parameters:

- 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 until the trust anchor. This source is only needed when these certificates are not included in the signature itself;
- the source of AIA;
- the source of OCSP;
- the source of CRL;
- set of alerts defining the behavior on various occasions.

In the current implementation this object is only used when profiles `BASELINE-LT` or `BASELINE-LTA` are created. This configuration shall be provided into both augmentation and validation processes.

CertificateVerifier usage

```
CertificateVerifier cv = new CommonCertificateVerifier();

// The trusted certificate source is used to provide trusted certificates
// (the trust anchors where the certificate chain building should stop)
cv.setTrustedCertSources(trustedCertSource);

// The adjunct certificate source is used to provide missing intermediate certificates
// (not trusted certificates)
cv.setAdjunctCertSources(adjunctCertSource);

// The AIA source is used to collect certificates from external resources (AIA)
cv.setAIASource(aiaSource);

// The OCSP Source to be used for external accesses (can be configured with a
// cache,...)
cv.setOcspSource(ocspSource);

// The CRL Source to be used for external accesses (can be configured with a
// cache,...)
cv.setCrlSource(crlSource);

// Sets the default digest algorithm that will be used for digest calculation
// of tokens used during the validation process.
// The values will be used in validation reports.
// Default : DigestAlgorithm.SHA256
cv.setDefaultDigestAlgorithm(DigestAlgorithm.SHA512);
```

```
// Define the behavior to be followed by DSS in case of revocation checking for
// certificates issued from an unsure source (DSS v5.4+)
// Default : revocation check is disabled for unsure sources (security reasons)
cv.setCheckRevocationForUntrustedChains(false);

// DSS v5.4+ : The 3 below configurations concern the extension mode (LT/LTA
// extension)

// Defines a behavior in case of missing revocation data
// Default : ExceptionOnStatusAlert -> interrupt the process
cv.setAlertOnMissingRevocationData(new ExceptionOnStatusAlert());

// Defines a behavior if a TSU certificate chain is not covered with a
// revocation data (timestamp generation time > CRL/OCSP production time).
// Default : LogOnStatusAlert -> a WARN log
cv.setAlertOnUncoveredPOE(new LogOnStatusAlert(Level.WARN));

// Defines a behavior if a revoked certificate is present
// Default : ExceptionOnStatusAlert -> interrupt the process
cv.setAlertOnRevokedCertificate(new ExceptionOnStatusAlert());

// Defines a behavior if an invalid timestamp is found
// Default : ExceptionOnStatusAlert -> interrupt the process
cv.setAlertOnInvalidTimestamp(new ExceptionOnStatusAlert());

// DSS v5.5+ : defines a behavior in case if there is no valid revocation
// data with thisUpdate time after the best signature time
// Example: if a signature was extended to T level then the obtained revocation
// must have thisUpdate time after production time of the signature timestamp.
// Default : LogOnStatusAlert -> a WARN log
cv.setAlertOnNoRevocationAfterBestSignatureTime(new LogOnStatusAlert(Level.ERROR));

// DSS 5.9+ :
// Defines behavior in case if the signing certificate or its related POE(s) have been
expired
// Default : ExceptionOnStatusAlert -> interrupt the process
cv.setAlertOnExpiredSignature(new ExceptionOnStatusAlert());

// DSS 5.9+ :
// RevocationDataLoadingStrategy defines logic for loading OCSP or CRL data
// Default : OCSPFirstRevocationDataLoadingStrategy -> loads OCSP first,
//           if not available or the response is invalid, then tries to load CRL
cv.setRevocationDataLoadingStrategy(new OCSPFirstRevocationDataLoadingStrategy());
```

See section Use of Alerts throughout the framework in the Annex for more information on alerts.

7.2. AdES validation constraints/policy

The validation process is driven by a set of constraints that are contained in the XML policy file.

In order to run a validation process with a custom validation policy, an XML file shall be created in compliance with the policy.xsd schema and passed to the relevant `DocumentValidator` as shown below.

Custom validation policy

```
Reports reports = validator.validateDocument(new
File("/path/to/validation/policy.xml"));
```

7.2.1. XML policy structure

The validation policy allows defining different behavior for various token types or signature formats. The following groups are considered:

- `ContainerConstraints` - defines rules for processing of ASiC containers validation;
- `SignatureConstraints` - defines rules for signature basic building blocks processing and the related certificate chain;
- `CounterSignatureConstraints` - allows defining custom rules for counter signature processing;
- `Timestamp` - defines rules for timestamp validation;
- `Revocation` - defines rules for revocation data validation;
- `Cryptographic` - defines common rules for cryptographic validation of used algorithms. The general constraints are used when no cryptographic constraints are defined for a particular token type;
- `Model` - defines the way of a certificate chain processing;
- `eIDAS` - defines rules for validation of Trusted Lists.

7.2.2. Constraints

Each constraint defined in the policy forces an execution of a relevant check in the validation process.

If a constraint is missing in the policy - the check is not processed.

The following constraint types are supported:

- `LevelConstraint` - a simple constraint type with a defined processing `Level`;
- `ValueConstraint` - defines a single acceptable value for the constraint;
- `IntValueConstraint` - defines an integer value for the constraint (the behavior depends on the check);
- `MultiValuesConstraint` - defines a set of accepted values relatively to the using constraint;
- `TimeConstraint` - defines a time unit and value for the constraint (the behavior depends on the check). See Revocation freshness for an example of the constraint use.

7.2.3. Level

The `Level` attribute of a constraint defines a validation process behavior in case of a check failure. While used, the following behaviors apply in case of a check failure:

- `FAIL` - brakes the validation process and returns the relevant indication;
- `WARN` - continues the validation process and returns a warning message to the validation process output;
- `INFORM` - continues the validation process and returns an information message to the validation process output;
- `IGNORE` - processes the check in a silent mode. The check is shown in the output report, but does not have impact on the process (equivalent to a not defined constraint).

7.2.4. Multi Values Constraint

When using the `MultiValuesConstraint`, a list of acceptable values shall be defined in the list of `<Id>⋯</Id>` elements, one for each accepted value. While doing, the following rules apply:

- Empty list of values → accept only empty values for the item in question, fails otherwise;
- `"*"` constraint value → accepts all values, reject empty list of values;
- Custom values → accepts only item values matching the constraint.

7.2.5. Cryptographic constraints

Cryptographic constraints define a list of acceptable cryptographic algorithms and their expiration dates when needed. The following settings are possible:

- `AcceptableEncryptionAlgo` - defines a list of acceptable encryption algorithms. All tokens and signatures using other algorithms will be rejected.
- `MiniPublicKeySize` - defines the minimal allowed public key size to be used with the defined encryption algorithms. An algorithm with a key size less than the defined one will be rejected. The minimal key size if required to be defined for an encryption algorithm, otherwise all used key sizes will be rejected.
- `AcceptableDigestAlgo` - defines a list of acceptable digest algorithms. All tokens and signatures using other algorithms will be rejected.
- `AlgoExpirationDate` - defines expiration dates for the algorithms. The algorithm is rejected when it is used after the defined date. If the algorithm expiration date is not defined, or set to null, the algorithm is treated as reliable for an unlimited time.

7.2.6. The default XML policy

The default XML validation policy is present below.

constraint.xml

```
<ConstraintsParameters Name="QES AdESQC TL based"
```

```
xmlns="http://dss.esig.europa.eu/validation/policy">
    <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>
    <ContainerConstraints>
        <AcceptableContainerTypes Level="FAIL">
            <Id>ASiC-S</Id>
            <Id>ASiC-E</Id>
        </AcceptableContainerTypes>
        <!--         <ZipCommentPresent Level="WARN" /> -->
        <!--         <AcceptableZipComment Level="WARN"> -->
        <!--             <Id>mimetype=application/vnd.etsi.asic-s+zip</Id> -->
        <!--             <Id>mimetype=application/vnd.etsi.asic-e+zip</Id> -->
        <!--         </AcceptableZipComment> -->
        <MimeTypeFilePresent Level="FAIL" />
        <AcceptableMimeTypeFileContent Level="WARN">
            <Id>application/vnd.etsi.asic-s+zip</Id>
            <Id>application/vnd.etsi.asic-e+zip</Id>
        </AcceptableMimeTypeFileContent>
        <ManifestFilePresent Level="FAIL" />
        <SignedFilesPresent Level="FAIL" />
        <AllFilesSigned Level="WARN" />
    </ContainerConstraints>
    <SignatureConstraints>
        <StructuralValidation Level="WARN" />
        <AcceptablePolicies Level="FAIL">
            <Id>ANY_POLICY</Id>
            <Id>NO_POLICY</Id>
        </AcceptablePolicies>
        <PolicyAvailable Level="FAIL" />
        <PolicyHashMatch Level="FAIL" />
        <AcceptableFormats Level="FAIL">
            <Id>*</Id>
        </AcceptableFormats>
        <BasicSignatureConstraints>
            <ReferenceDataExistence Level="FAIL" />
            <ReferenceDataIntact Level="FAIL" />
            <ManifestEntryObjectExistence Level="WARN" />
            <SignatureIntact Level="FAIL" />
            <SignatureDuplicated Level="FAIL" />
            <ProspectiveCertificateChain Level="FAIL" />
            <SignerInformationStore Level="FAIL" />
            <PdfPageDifference Level="FAIL" />
            <PdfAnnotationOverlap Level="WARN" />
            <PdfVisualDifference Level="WARN" />
```

```
            <!--             <TrustedServiceTypeIdentifier Level="WARN"> -->
            <!--                 <Id>http://uri.etsi.org/TrstSvc/Svctype/CA/QC</Id>
-->
            <!--             </TrustedServiceTypeIdentifier> -->
            <!--             <TrustedServiceStatus Level="FAIL"> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/undersupervision</Id> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/accredited</Id> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/supervisionincessation</Id> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/granted</Id> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/withdrawn</Id> -->
            <!--             </TrustedServiceStatus> -->
            <SigningCertificate>
                <Recognition Level="FAIL" />
                <Signature Level="FAIL" />
                <NotExpired Level="FAIL" />
                <AuthorityInfoAccessPresent Level="WARN" />
                <RevocationInfoAccessPresent Level="WARN" />
                <RevocationDataAvailable Level="FAIL" />
                <CRLNextUpdatePresent Level="WARN" />
                <RevocationFreshness Level="IGNORE" Unit="DAYS" Value="0" />
                <KeyUsage Level="WARN">
                    <Id>nonRepudiation</Id>
                </KeyUsage>
                <SerialNumberPresent Level="WARN" />
                <NotRevoked Level="FAIL" />
                <NotOnHold Level="FAIL" />
                <RevocationIssuerNotExpired Level="FAIL" />
                <NotSelfSigned Level="WARN" />
                <!--                 <Qualification Level="WARN" /> -->
                <!--                 <SupportedByQSCD Level="WARN" /> -->
                <!--                 <QcLegislationCountryCodes Level="WARN" /> -->
                <!--                 <IssuedToNaturalPerson Level="INFORM" /> -->
                <!--                 <IssuedToLegalPerson Level="INFORM" /> -->
                <UsePseudonym Level="INFORM" />
                <Cryptographic />
            </SigningCertificate>
            <CACertificate>
                <Signature Level="FAIL" />
                <NotExpired Level="FAIL" />
                <RevocationDataAvailable Level="FAIL" />
                <CRLNextUpdatePresent Level="WARN" />
                <RevocationFreshness Level="IGNORE" Unit="DAYS" Value="0" />
                <NotRevoked Level="FAIL" />
                <NotOnHold Level="FAIL" />
                <Cryptographic />
            </CACertificate>
```

```
            <Cryptographic />
        </BasicSignatureConstraints>
        <SignedAttributes>
            <SigningCertificatePresent Level="WARN" />
            <UnicitySigningCertificate Level="WARN" />
            <SigningCertificateRefersCertificateChain Level="WARN" />
            <CertDigestPresent Level="FAIL" />
            <CertDigestMatch Level="FAIL" />
            <IssuerSerialMatch Level="WARN" />
            <SigningTime Level="FAIL" />
            <MessageDigestOrSignedPropertiesPresent Level="FAIL" />
            <!--     <ContentType Level="FAIL" value="1.2.840.113549.1.7.1" />
                       <ContentHints Level="FAIL" value="*" />
                       <CommitmentTypeIndication Level="FAIL">
                           <Id>1.2.840.113549.1.9.16.6.1</Id>
                           <Id>1.2.840.113549.1.9.16.6.4</Id>
                           <Id>1.2.840.113549.1.9.16.6.5</Id>
                           <Id>1.2.840.113549.1.9.16.6.6</Id>
                       </CommitmentTypeIndication>
                       <SignerLocation Level="FAIL" />
                       <ContentTimeStamp Level="FAIL" /> -->
        </SignedAttributes>
        <UnsignedAttributes>
            <!--     <CounterSignature Level="IGNORE" /> check presence -->
        </UnsignedAttributes>
    </SignatureConstraints>
    <CounterSignatureConstraints>
        <BasicSignatureConstraints>
            <ReferenceDataExistence Level="FAIL" />
            <ReferenceDataIntact Level="FAIL" />
            <SignatureIntact Level="FAIL" />
            <SignatureDuplicated Level="FAIL" />
            <ProspectiveCertificateChain Level="FAIL" />
            <!--             <TrustedServiceTypeIdentifier Level="WARN"> -->
            <!--                 <Id>http://uri.etsi.org/TrstSvc/Svctype/CA/QC</Id>
-->
            <!--             </TrustedServiceTypeIdentifier> -->
            <!--             <TrustedServiceStatus Level="FAIL"> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/undersupervision</Id> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/accredited</Id> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/supervisionincessation</Id> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/granted</Id> -->
            <!--
<Id>http://uri.etsi.org/TrstSvc/TrustedList/Svcstatus/withdrawn</Id> -->
            <!--             </TrustedServiceStatus> -->
            <SigningCertificate>
                <Recognition Level="FAIL" />
```

```
            <Signature Level="FAIL" />
            <NotExpired Level="FAIL" />
            <AuthorityInfoAccessPresent Level="WARN" />
            <RevocationInfoAccessPresent Level="WARN" />
            <RevocationDataAvailable Level="FAIL" />
            <CRLNextUpdatePresent Level="WARN" />
            <RevocationFreshness Level="IGNORE" Unit="DAYS" Value="0" />
            <KeyUsage Level="WARN">
                <Id>nonRepudiation</Id>
            </KeyUsage>
            <SerialNumberPresent Level="WARN" />
            <NotRevoked Level="FAIL" />
            <NotOnHold Level="FAIL" />
            <NotSelfSigned Level="WARN" />
            <!--                 <Qualification Level="WARN" /> -->
            <!--                 <SupportedByQSCD Level="WARN" /> -->
            <!--                 <IssuedToNaturalPerson Level="INFORM" /> -->
            <!--                 <IssuedToLegalPerson Level="INFORM" /> -->
            <UsePseudonym Level="INFORM" />
            <Cryptographic />
        </SigningCertificate>
        <CACertificate>
            <Signature Level="FAIL" />
            <NotExpired Level="FAIL" />
            <RevocationDataAvailable Level="FAIL" />
            <CRLNextUpdatePresent Level="WARN" />
            <RevocationFreshness Level="IGNORE" Unit="DAYS" Value="0" />
            <NotRevoked Level="FAIL" />
            <NotOnHold Level="FAIL" />
            <Cryptographic />
        </CACertificate>
        <Cryptographic />
    </BasicSignatureConstraints>
    <SignedAttributes>
        <SigningCertificatePresent Level="WARN" />
        <CertDigestPresent Level="FAIL" />
        <CertDigestMatch Level="FAIL" />
        <IssuerSerialMatch Level="WARN" />
        <SigningTime Level="FAIL" />
        <MessageDigestOrSignedPropertiesPresent Level="FAIL" />
        <!--     <ContentType Level="FAIL" value="1.2.840.113549.1.7.1" />
                    <ContentHints Level="FAIL" value="*" />
                    <CommitmentTypeIndication Level="FAIL">
                        <Id>1.2.840.113549.1.9.16.6.1</Id>
                        <Id>1.2.840.113549.1.9.16.6.4</Id>
                        <Id>1.2.840.113549.1.9.16.6.5</Id>
                        <Id>1.2.840.113549.1.9.16.6.6</Id>
                    </CommitmentTypeIndication>
                    <SignerLocation Level="FAIL" />
                    <ContentTimeStamp Level="FAIL" /> -->
    </SignedAttributes>
```

```
    </CounterSignatureConstraints>
    <Timestamp>
        <TimestampDelay Level="IGNORE" Unit="DAYS" Value="0" />
        <RevocationTimeAgainstBestSignatureTime  Level="FAIL" />
        <BestSignatureTimeBeforeExpirationDateOfSigningCertificate Level="FAIL" />
        <Coherence Level="WARN" />
        <BasicSignatureConstraints>
            <ReferenceDataExistence Level="FAIL" />
            <ReferenceDataIntact Level="FAIL" />
            <SignatureIntact Level="FAIL" />
            <ProspectiveCertificateChain Level="FAIL" />
            <SigningCertificate>
                <Recognition Level="FAIL" />
                <Signature Level="FAIL" />
                <NotExpired Level="FAIL" />
                <RevocationDataAvailable Level="FAIL" />
                <CRLNextUpdatePresent Level="WARN" />
                <RevocationFreshness Level="IGNORE" Unit="DAYS" Value="0" />
                <ExtendedKeyUsage Level="WARN">
                    <Id>timeStamping</Id>
                </ExtendedKeyUsage>
                <NotRevoked Level="FAIL" />
                <NotOnHold Level="FAIL" />
                <NotSelfSigned Level="WARN" />
                <Cryptographic />
            </SigningCertificate>
            <CACertificate>
                <Signature Level="FAIL" />
                <NotExpired Level="FAIL" />
                <RevocationDataAvailable Level="WARN" />
                <CRLNextUpdatePresent Level="WARN" />
                <RevocationFreshness Level="IGNORE" Unit="DAYS" Value="0" />
                <NotRevoked Level="FAIL" />
                <NotOnHold Level="FAIL" />
                <Cryptographic />
            </CACertificate>
            <Cryptographic />
        </BasicSignatureConstraints>
        <SignedAttributes>
            <SigningCertificatePresent Level="WARN" />
            <!-- <UnicitySigningCertificate Level="WARN" /> RFC 5816 -->
            <SigningCertificateRefersCertificateChain Level="WARN" />
            <CertDigestPresent Level="WARN" />
            <IssuerSerialMatch Level="WARN" />
        </SignedAttributes>
        <TSAGeneralNameContentMatch Level="WARN" />
    </Timestamp>
    <Revocation>
        <UnknownStatus Level="FAIL" />
        <SelfIssuedOCSP Level="WARN" />
        <BasicSignatureConstraints>
```

```
            <ReferenceDataExistence Level="FAIL" />
            <ReferenceDataIntact Level="FAIL" />
            <SignatureIntact Level="FAIL" />
            <ProspectiveCertificateChain Level="FAIL" />
            <SigningCertificate>
                <Recognition Level="FAIL" />
                <Signature Level="FAIL" />
                <NotExpired Level="FAIL" />
                <RevocationDataAvailable Level="FAIL" />
                <CRLNextUpdatePresent Level="WARN" />
                <RevocationFreshness Level="IGNORE" Unit="DAYS" Value="0" />
                <NotRevoked Level="FAIL" />
                <NotOnHold Level="FAIL" />
                <Cryptographic />
            </SigningCertificate>
            <CACertificate>
                <Signature Level="FAIL" />
                <NotExpired Level="FAIL" />
                <RevocationDataAvailable Level="WARN" />
                <CRLNextUpdatePresent Level="WARN" />
                <RevocationFreshness Level="IGNORE" Unit="DAYS" Value="0" />
                <NotRevoked Level="FAIL" />
                <NotOnHold Level="FAIL" />
                <Cryptographic />
            </CACertificate>
            <Cryptographic />
        </BasicSignatureConstraints>
    </Revocation>
    <Cryptographic Level="FAIL">
        <AcceptableEncryptionAlgo>
            <Algo>RSA</Algo>
            <Algo>DSA</Algo>
            <Algo>ECDSA</Algo>
            <Algo>PLAIN-ECDSA</Algo>
            <!--         <Algo>Ed25519</Algo>                Not referenced in
ETSI/SOGIS -->
        </AcceptableEncryptionAlgo>
        <MiniPublicKeySize>
            <Algo Size="1024">DSA</Algo>
            <Algo Size="1024">RSA</Algo>
            <Algo Size="160">ECDSA</Algo>
            <Algo Size="160">PLAIN-ECDSA</Algo>
            <!--         <Algo Size="24">Ed25519</Algo>      Not referenced in
ETSI/SOGIS -->
        </MiniPublicKeySize>
        <AcceptableDigestAlgo>
            <Algo>MD2</Algo>
            <Algo>MD5</Algo>
            <Algo>SHA1</Algo>
            <Algo>SHA224</Algo>
            <Algo>SHA256</Algo>
```

```
            <Algo>SHA384</Algo>
            <Algo>SHA512</Algo>
            <Algo>SHA3-224</Algo>
            <Algo>SHA3-256</Algo>
            <Algo>SHA3-384</Algo>
            <Algo>SHA3-512</Algo>
            <Algo>RIPEMD160</Algo>
            <Algo>WHIRLPOOL</Algo>
        </AcceptableDigestAlgo>
        <AlgoExpirationDate Format="yyyy">
            <!-- Digest algorithms -->
            <Algo Date="2005">MD2</Algo> <!-- The same as for MD5 -->
            <Algo Date="2005">MD5</Algo> <!-- ETSI TS 102 176-1 (Historical) V2.1.1
-->
            <Algo Date="2009">SHA1</Algo> <!-- ETSI TS 102 176-1 (Historical) V2.0.0
-->
            <Algo Date="2023">SHA224</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026">SHA256</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026">SHA384</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026">SHA512</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026">SHA3-224</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026">SHA3-256</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026">SHA3-384</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026">SHA3-512</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2011">RIPEMD160</Algo> <!-- ETSI TS 102 176-1 (Historical)
V2.0.0 -->
            <Algo Date="2015">WHIRLPOOL</Algo> <!-- ETSI 119 312 V1.1.1 -->
            <!-- end Digest algorithms -->
            <!-- Encryption algorithms -->
            <Algo Date="2013" Size="1024">DSA</Algo> <!-- ETSI TS 102 176-1
(Historical) V2.1.1 -->
            <Algo Date="2023" Size="2048">DSA</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026" Size="3072">DSA</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2009" Size="1024">RSA</Algo> <!-- ETSI TS 102 176-1
(Historical) V2.0.0 -->
            <Algo Date="2016" Size="1536">RSA</Algo> <!-- ETSI 119 312 V1.1.1 -->
            <Algo Date="2023" Size="1900">RSA</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026" Size="3000">RSA</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2013" Size="160">ECDSA</Algo> <!-- ETSI TS 102 176-1
(Historical) V2.1.1 -->
            <Algo Date="2013" Size="192">ECDSA</Algo> <!-- ETSI TS 102 176-1
(Historical) V2.1.1 -->
            <Algo Date="2016" Size="224">ECDSA</Algo> <!-- ETSI 119 312 V1.1.1 -->
            <Algo Date="2026" Size="256">ECDSA</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026" Size="384">ECDSA</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2026" Size="512">ECDSA</Algo> <!-- ETSI 119 312 V1.3.1 -->
            <Algo Date="2013" Size="160">PLAIN-ECDSA</Algo> <!-- ETSI TS 102 176-1
(Historical) V2.1.1 -->
            <Algo Date="2013" Size="192">PLAIN-ECDSA</Algo> <!-- ETSI TS 102 176-1
(Historical) V2.1.1 -->
            <Algo Date="2016" Size="224">PLAIN-ECDSA</Algo> <!-- ETSI 119 312 V1.1.1
```

```
-->
            <Algo Date="2026" Size="256">PLAIN-ECDSA</Algo> <!-- ETSI 119 312 V1.3.1
-->
            <Algo Date="2026" Size="384">PLAIN-ECDSA</Algo> <!-- ETSI 119 312 V1.3.1
-->
            <Algo Date="2026" Size="512">PLAIN-ECDSA</Algo> <!-- ETSI 119 312 V1.3.1
-->

            <!--         <Algo Date="2026" Size="32">Ed25519</Algo>      Not
referenced in ETSI/SOGIS -->
            <!-- end Encryption algorithms -->
        </AlgoExpirationDate>
    </Cryptographic>

    <Model Value="SHELL" />

    <!-- eIDAS REGL 910/EU/2014 -->
    <eIDAS>
        <TLFreshness Level="WARN" Unit="HOURS" Value="6" />
        <TLNotExpired Level="WARN" />
        <TLWellSigned Level="WARN" />
        <TLVersion Level="FAIL" value="5" />
    </eIDAS>
</ConstraintsParameters>
```

7.3. Signature validation and reports

Generally, a signature validation process outputs an indication status and a validation report as described in section Signature validation (introduction).

In DSS, the result of the validation process consists of four elements:

- the Simple Report,
- the Detailed Report ,
- the Diagnostic Data and
- the ETSI Validation Report.

All these reports are represented in XML format, which allows the implementer to easily manipulate and extract information for further analysis. For each report, XML Schema and JaxB model are available as maven dependencies.

DSS also provides XSLT for generation of PDF or HTML reports (simple and detailed reports).

You will find below a detailed description of each of these elements.

7.3.1. Validating an AdES signature

The DSS validation process is based on the ETSI standard EN 319 102-1 [R09]. 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 dependencies between those 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.

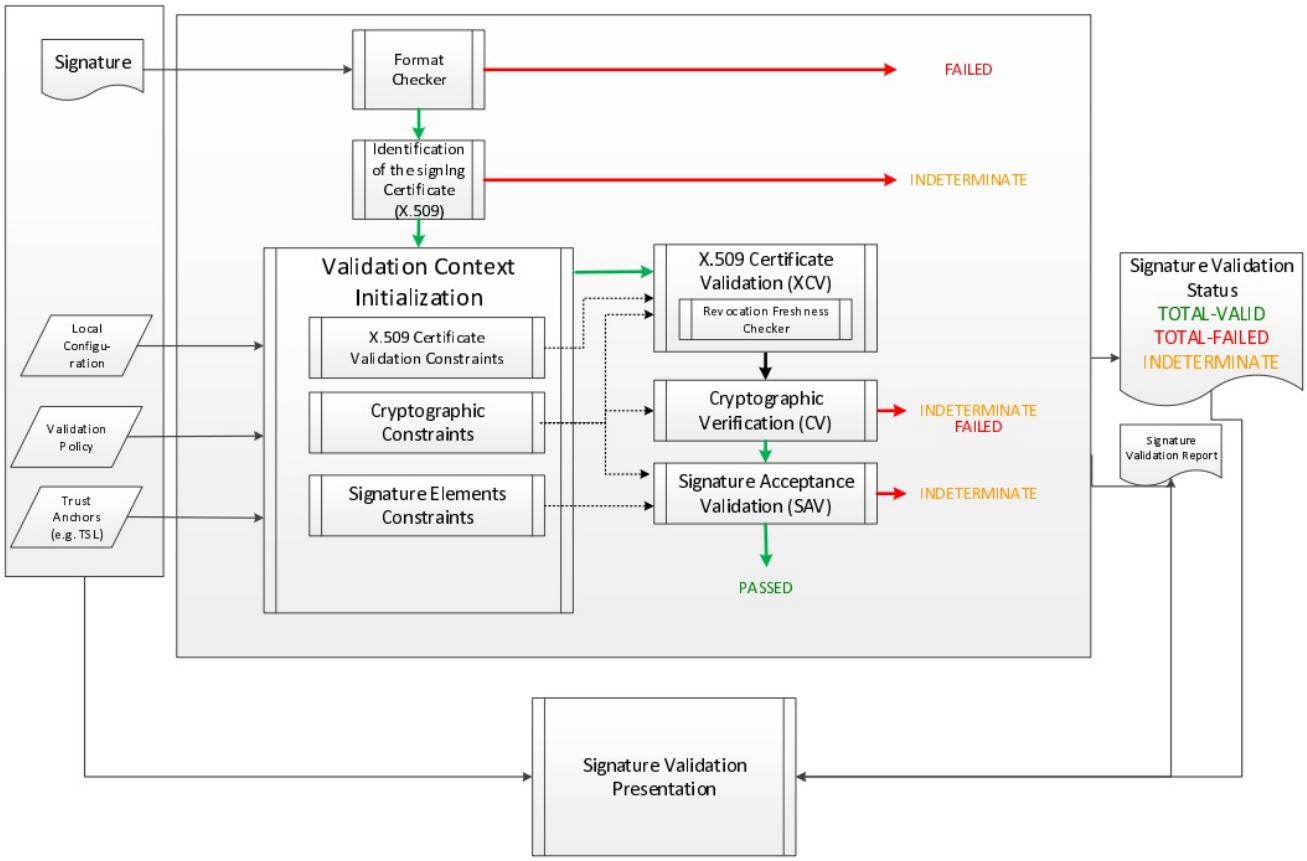


Figure 1. Signature Validation Process

Note that depending on a used signature format and packaging, the whole or only a part of the original data is signed. Thus, in XAdES the signed content depends on the used transforms within a reference element, and in case of CAdES or PAdES signature the whole document must be signed.

At the end of the validation process four reports are created. They contain different detail levels concerning the validation result. They provide different kinds of visions for the validation process: macroscopic, microscopic, validation data and ETSI Validation report conformant with the standard [R09]. For more information about these reports, please refer to Simple Report, Detailed Report, Diagnostic Data and ETSI Validation Report chapters respectively.

Below is the simplest example of signature validation from an input document. The first step consists in 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 next step is to create an object that will check the status of a certificate using the Trusted List model (see Trusted Lists for more information). In order to achieve this, an instance of a `CertificateVerifier` must be created with a defined source of trusted certificates. In our example, the trusted source is instantiated with `CommonTrustedCertificateSource` class. As well as a trusted

source, the `CertificateVerifier` 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 Revocation Data Management for more information about revocation sources.

Validation of a signature

```
// First, we need a Certificate verifier
CertificateVerifier cv = new CommonCertificateVerifier();

// We can inject several sources. eg: OCSP, CRL, AIA, trusted lists

// Capability to download resources from AIA
cv.setAIASource(new DefaultAIASource());

// Capability to request OCSP Responders
cv.setOcspSource(new OnlineOCSPSource());

// Capability to download CRL
cv.setCrlSource(new OnlineCRLSource());

// Create an instance of a trusted certificate source
CommonTrustedCertificateSource trustedCertSource = new
CommonTrustedCertificateSource();
// import the keystore as trusted
trustedCertSource.importAsTrusted(keystoreCertSource);

// Add trust anchors (trusted list, keystore,...) to a list of trusted certificate
sources
// Hint : use method {@code CertificateVerifier.setTrustedCertSources(certSources)} in
order to overwrite the existing list
cv.addTrustedCertSources(trustedCertSource);

// Additionally add missing certificates to a list of adjunct certificate sources (not
trusted certificates)
cv.addAdjunctCertSources(adjunctCertSource);

// Here is the document to be validated (any kind of signature file)
DSSDocument document = new FileDocument(new File("src/test/resources/signature-
pool/signedXmlXadesLT.xml"));

// We create an instance of DocumentValidator
// It will automatically select the supported validator from the classpath
SignedDocumentValidator documentValidator =
SignedDocumentValidator.fromDocument(document);

// We add the certificate verifier (which allows to verify and trust certificates)
documentValidator.setCertificateVerifier(cv);

// Here, everything is ready. We can execute the validation (for the example, we use
the default and embedded
// validation policy)
```

```
Reports reports = documentValidator.validateDocument();

// We have 4 reports
// The diagnostic data which contains all used and static data
DiagnosticData diagnosticData = reports.getDiagnosticData();

// The detailed report which is the result of the process of the diagnostic data and
the validation policy
DetailedReport detailedReport = reports.getDetailedReport();

// The simple report is a summary of the detailed report (more user-friendly)
SimpleReport simpleReport = reports.getSimpleReport();

// The JAXB representation of the ETSI Validation report (ETSI TS 119 102-2)
ValidationReportType estiValidationReport = reports.getEtsiValidationReportJaxb();
```

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, e.g. 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.

7.3.1.1. SignedDocumentValidator

For execution of the validation process, DSS uses the `SignedDocumentValidator` class. The DSS framework provides the following implementations of the validator:

- `XMLDocumentValidator` - validates documents in XML format (XAdES format);
- `CMSDocumentValidator` - validates documents in CMS format (CAdES format);
- `PDFDocumentValidator` - validates documents in PDF format (PADES format);
- `JWSCompactDocumentValidator` - validates documents with base64url encoded content (JAdES compact format);
- `JWSSerializationDocumentValidator` - validates documents in JSON format (JAdES serialization formats);
- `ASiCContainerWithXAdESValidator` - validates ASiC with XAdES containers;
- `ASiCContainerWithCAdESValidator` - validates ASiC with CAdES containers;
- `DetachedTimestampValidator` - validates CMS timestamps provided alone.

DSS initializes a relevant validator based on specific characteristics of an input file (e.g. a PDF file version declaration for a PDF file). It checks the file format and loads the required validator from a classpath. Below you can find a list of settings that can be used for the configuration of the class.

SignedDocumentValidator usage

```
// The method allows instantiation of a related validator for a provided document
// independently on its format (the target dss module must be added as dependency)
SignedDocumentValidator documentValidator =
SignedDocumentValidator.fromDocument(document);

// Allows specifying a custom certificate verifier (online or offline)
documentValidator.setCertificateVerifier(new CommonCertificateVerifier());

// Allows specifying which tokens need to be extracted in the diagnostic data
(Base64).
// Default : NONE)
documentValidator.setTokenExtractionStrategy(TokenExtractionStrategy.EXTRACT_CERTIFICA
TES_AND_TIMESTAMPS);

// Allows providing signing certificate(s) in the explicit way, in case if the
// certificate is not provided in the signature itself (can be used for non-ASiC
signatures)
CertificateSource signingCertificateSource = new CommonCertificateSource();
signingCertificateSource.addCertificate(DSSUtils.loadCertificateFromBase64EncodedStrin
g(

"MIIC9TCCAd2gAwIBAgIBAjANBgkqhkiG9w0BAQUFADArMQswCQYDVQQGEwJBQTEMMAoGA1UEChMDRFNTMQ4wD
AYDVQQDEwVJQ0EgQTAeFw0xMzEyMDIxNzMzMTBaFw0xNTEyMDIxNzMzMTBaMDAxCzAJBgNVBAYTAkFBMQwwCgY
DVQQKEwNEU1MxEzARBgNVBAMTCnVzZXIgQSBSU0EwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAJUHHAphm
SDdQ1t62tppK+dLTANsE2nAj+HCpasS3ohlBsrhteRsvTAbrDyIzCmTYWu/nVI4TGvbzBESwV/QitlkoMLpYFw
32MIBf2DLmECzGJ3vm5haw6u8S9quR1h8Vu7QWd+5KMabZuR+j91RiSuoY0xS2ZQxJw1vhvW9hRYjAgMBAAGjg
aIwgZ8wCQYDVR0TBAIwADAdBgNVHQ4EFgQU9ESnTWfwg13c3LQZzqqwibY5WVYwUwYDVR0jBEwwSoAUIO1CDsB
SUcEoFZxKaWf1PAL1U+uhL6QtMCsxDDAKBgNVBAoTA0RTUzELMAkGA1UEBhMCQUExDjAMBgNVBAMTBVJDQSBBg
gEBMAsGA1UdDwQEAwIHgDARBgNVHSAECjAIMAYGBFUdIAAwDQYJKoZIhvcNAQEFBQADggEBAGnhhnoyVUhDnr/
BSbZ/uWfSuwzFPG+2V9K6WxdIaaXOORFGIdFwGlAwA/Qzpq9snfBxuTkAykxq0uEDhHTj0qXxWRjQ+Dop/Drmc
coF/zDvgGusyY1YXaABd/kc3IYt7ns7z3tpiqIz4A7a/UHplBRXfqjyaZurZuJQRaSdxh6CNhdEUiUBxkbb1Sd
MjuOgjzSDjcDjcegjvDquMKdDetvtu2Qh4ConBBo3fUImwiFRWnbudS5H2HE18ikC7gY/QIuNr7USf1PNyUgcG
2g31cMtemj7UTBHZ2V/jPf7ZXqwfnVSaYkNvM3weAI6R3PI0STjdxN6a9qjt9xld40YEdw="));
documentValidator.setSigningCertificateSource(signingCertificateSource);

// Sets the detached contents that were used for the detached signature creation
documentValidator.setDetachedContents(Arrays.asList(new InMemoryDocument("Hello
world!".getBytes())));

// Allows defining a custom Process Executor
// By default used {@code new DefaultSignatureProcessExecutor()}
documentValidator.setProcessExecutor(new DefaultSignatureProcessExecutor());

// Sets custom Signature Policy Provider
documentValidator.setSignaturePolicyProvider(new SignaturePolicyProvider());

// Sets an expected signature validation level
// The recommended level is ARCHIVAL_DATA (maximal level of the validation)
// Default : ValidationLevel.ARCHIVAL_DATA
```

```
documentValidator.setValidationLevel(ValidationLevel.ARCHIVAL_DATA);

// Sets if the ETSI validation report must be created
// If true, it will become accessible through the method below
// Default : true
documentValidator.setEnableEtsiValidationReport(true);

// Sets if the semantics for Indication / SubIndication must be included in the
// Simple Report (see table 5 / 6 of the ETSI TS 119 102-1)
// Default : false
documentValidator.setIncludeSemantics(true);

// Executes the validation process and produces validation reports:
// Simple report, Detailed report, Diagnostic data and ETSI Validation Report (if
enabled)
Reports reports = documentValidator.validateDocument();

// Returns ETSI Validation Report (if enabled, NULL otherwise)
ValidationReportType etsiValidationReport = reports.getEtsiValidationReportJaxb();
```

7.3.2. Simple report

The result of the validation process is based on complex algorithm and 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.

A sample of the simple validation report can be found here.

7.3.3. Detailed report

The structure of a Detailed Report is based on the ETSI EN 319 102-1 standard ([R09]).

It is a representation of steps performed during the validation process, as defined in the ETSI EN 319 102-1 standard, and structured using the processes and blocks defined in that standard:

- Basic Building Blocks;
- Validation Process for Basic Signatures;
- Time-stamp validation building block;
- Validation process for Signatures with Time and Signatures with Long-Term Validation Material;
- Validation process for Signatures providing Long Term Availability and Integrity of Validation.

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

- **FC** - Format Checking;
- **ISC** - Identification of the Signing Certificate;

- **VCI** - Validation Context Initialization;
- **RFC** - Revocation Freshness Checker;
- **XCV** - X.509 certificate validation;
- **CV** - Cryptographic Verification;
- **SAV** - Signature Acceptance Validation.

The following additional elements also can be executed in case of validation in the past:

- **PCV** - Past Certificate Validation;
- **VTS** - Validation Time Sliding process;
- **POE extraction** - Proof Of Existence extraction;
- **PSV** - Past Signature Validation.

To process the revocation data, DSS performs the following additional checks:

- **CRS** (CertificateRevocationSelector) - validates a set of revocation data for a given certificate and returns the latest valid entry known to contain information about the concerned certificate;
- **RAC** (RevocationAcceptanceCheck) - verifies whether one single revocation data is known to contain information about the concerned certificate.

Past certificate/signature validation is used when basic validation of a certificate/signature fails at the current time with an `INDETERMINATE` status such that the provided proofs of existence may help to go to a determined status. The process shall initialize the *best-signature-time* either to a time indication for a related POE provided, or the current time when this parameter has not been used by the algorithm.

- **Best-signature-time** is an internal variable for the algorithm denoting the earliest time when it can be trusted by the SVA (either because proven by some POE present in the signature or passed by the DA and for this reason assumed to be trusted) that a signature has existed. [R09]

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 `PASSED`. Otherwise, `FAILED` or `INDETERMINATE` indication is returned depending on the ETSI standard definition.

Furthermore, a module has been introduced in DSS to allow changing the language of reports generated by DSS. Currently, this is only possible for messages for the checks executed during the validation process. For more information on that topic, see section Language of reports.

A sample of a DetailedReport is provided here, and an illustration on how to interpret "what went wrong" based on a detailed report is provided in Interpreting a detailed report

7.3.4. Diagnostic data

Diagnostic data is a data set constructed from the information contained in the signature itself, but also from information retrieved dynamically like revocation data and information extrapolated like

the mathematical validity of a signature. The diagnostic data is constructed before the validation is completed, and it is used by DSS to validate the signature and create a validation report.

The diagnostic data is independent of the applied validation policy. Two different validation policies applied to the same diagnostic data can lead to different results.

It is also possible to provide a Diagnostic Data directly to the validation process without the actual signature ("replay the diagnostic data"). Since the diagnostic data is constructed before the validation, it can be used to see what the validation report would have been if certain fields of the diagnostic data would have been different. For example, changing the digest method from SHA-256 to SHA-1 would result in different validation reports. The impact of the different fields on the validation can be observed by replaying the diagnostic data.

The validation report resulting from the replay of the diagnostic data is useful for observation but cannot be used as a proof of signature validity like the validation report directly resulting from a validation process.

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

7.3.5. ETSI validation report

The ETSI Validation Report represents an implementation of TS 119 102-2 (cf. [R13]). The report contains a standardized result of an ASiC digital signature validation. It includes the original validation input data, the applied validation policy, as well as the validation result of one or more signature(s) and its(their) constraints.

An example of the ETSI validation report can be found here.

7.3.6. Stylesheets for validation reports and diagnostic data

The reports are generated in the XML format, which is not the most straightforward way of reading a report. To represent the information in a user-friendly manner stylesheets are used. A stylesheet is a set of rules that transforms XML content into an HTML or PDF representation to have a human-readable text. Refer to section Report stylesheets for more information on the stylesheets used for final report generation.

It is also possible to use a stylesheet to generate an SVG image from an XML document such as the diagnostic data. (cf. Diagnostic data stylesheets)

7.4. Various DSS validation options

7.4.1. Validation level

There exist four signature levels: `BASELINE-B`, `BASELINE-T`, `BASELINE-LT` and `BASELINE-LTA` (cf. Signature classes/levels). For signature validation, DSS allows indicating which level of the signature needs to be validated. The validation process can be for basic signatures, signatures with long-term validation data or signatures with archival data.

7.4.2. Signing Certificate

DSS allows adding the certificate that was used to sign the document to the inputs for the validation process. This might be useful if the signing certificate was not included as a signed attribute, for example when validating non-AdES signatures.

7.4.3. Adjunct Certificates

DSS allows adding a set of certificates that could be used for a certificate path building, e.g. a timestamp certificate, CA certificate, and so on.

7.4.4. Certificates

In DSS, it is possible to return the certificates, included in the signature, as output of the validation process.

7.4.5. Timestamps

DSS allows returning the timestamps, included in the signature, as output of the validation process.

7.4.6. Revocation data

In DSS, it is possible to return the revocation data (CRLs and OCSPs), included in the signature, as output of the validation process.

7.4.7. User-friendly identifiers

DSS supports the use of user-friendly identifiers instead of hash-based values to represent signatures and tokens. A hash-base representation could be "S-651B6527872B53437C7B9A8696BD9F7A6C311CE6EE418EFE34A4A994C05D08C8". The same information but presented in a user-friendly way is a string composed of "SIGNATURE" to indicate that it is a signature, the name in the certificate chain, the signature claimed time and so on.

7.4.8. Semantics

With DSS, it is possible to add a "Semantics" section at the end of the reports explaining the meaning of the result indications, i.e. `TOTAL_PASSED`, `PASSED`, `INDETERMINATE`, `NO_CERTIFICATE_CHAIN_FOUND`.

7.5. DocumentValidator implementation management

For signature document or a signature policy validation, DSS is able to load a corresponding implementation of validator for the given document format at runtime using a `ServiceLoader` (e.g. `XMLDocumentValidator` for XAdES signature).

DSS is able to choose the required implementation for the following interfaces:

- `DocumentValidationFactory` - checks a provided signed file's format and loads a relevant validator;

- `SignaturePolicyValidator` - checks a signature policy file and loads a relevant validator to be able to process the detected format.

If no appropriate available implementation is found, an exception will be thrown.

For more information about `ServiceLoader` usage please refer to the chapter ServiceLoader.

7.5.1. Document Validation Factory

This factory is used to create a required instance of a `DocumentValidator` based on the provided file's format (signature or timestamp). An implementation shall process a file format check and load the related SignedDocumentValidator implementation to be used for the file's validation.

The following implementations are present in DSS:

- `CMSDocumentValidatorFactory`: loads `CMSDocumentValidator`, used for a CAdES validation (delivered in `dss-cades` module);
- `XMLDocumentValidatorFactory`: loads `XMLDocumentValidator`, used for a XAdES validation (delivered in `dss-xades` module);
- `PDFDocumentValidatorFactory`: loads `PDFDocumentValidator`, used for a PAdES validation (delivered in `dss-pades` module);
- `JAdESDocumentValidatorFactory`: loads `JWSCompactDocumentValidator` or `JWSSerializationDocumentValidator`, depending on provided JSON signature type (delivered in `dss-jades` module);
- `ASiCContainerWithCAdESValidatorFactory`: loads `ASiCContainerWithCAdESValidator` (delivered in `dss-asic-cades` module);
- `ASiCContainerWithXAdESValidatorFactory`: loads `ASiCContainerWithXAdESValidator` (delivered in `dss-asic-xades` module);
- `DetachedTimestampValidatorFactory`: loads `DetachedTimestampValidator`, for an independent timestamp validation (delivered in `dss-document` module).

7.5.2. Signature Policy Validator

During the signature validation process, the signature policy shall be validated to verify that the retrieved policy is the one that was used for the signature creation. This can be achieved by verifying whether the digest within the `SignaturePolicyIdentifier` signed attribute of the signature matches the computed digest of the retrieved signature policy document.

The signature policy document can be retrieved from the signature itself when a `SignaturePolicyStore` attribute is present. It can also be retrieved from online or local sources using the `SignaturePolicyProvider` class (e.g. by URL from the Internet).

The interface `SignaturePolicyValidator` is used to validate a signature policy reference extracted from a signature. The choice of the implementation is format-specific. The following implementations are provided:

- `BasicASNSignaturePolicyValidator`: validates ASN.1 signature policies;

- `XMLSignaturePolicyValidator`: validates XML signature policies supporting transformations;
- `NonASN1SignaturePolicyValidator`: validates a policy by digest computed on an original file's content;
- `ZeroHashSignaturePolicyValidator`: validates a policy if "zero hash" value is defined in a signature (see [R02]);
- `EmptySignaturePolicyValidator`: is proceeded if a policy file is not found or not accessible.

7.6. Format specificities

7.6.1. PAdES

7.6.1.1. Shadow attack detection

"Shadow attack" is a class of attacks on a signed PDF document that constitutes a change of a visual content of a document after the signature has been made. Due to a structure of PDF document, the signature stays cryptographically valid even after the content's modification has been taken place. There is no known algorithm to detect the malicious change with 100% guarantee. For more information, please refer to the website.

DSS provides a set of own utils to detect the "shadow attack" on a signed PDF document. The following algorithms have been introduced:

- `Page amount difference` - the validation tool compares the number of pages between the obtained PDF and signed revision. If the numbers do not match, the validation fail. The validation level can be configured within the AdES validation constraints/policy with the constraint `<PdfPageDifference>`.
- `Annotations overlap` - DSS checks if any annotation overlaps occurred. The overlapping is potentially dangerous, because some annotations can cover a visual content, e.g. forms and signature fields. How this check is applied can be configured with the constraint `<PdfAnnotationOverlap>`.
- `Visual difference` - DSS verifies the visual difference between the provided document and signed revision, excluding the newly created annotations (between the validating revisions). How this check is applied can be configured with the constraint `<PdfVisualDifference>`.

7.6.1.2. Object modification detection

As an additional tool to detect malicious changes within a PDF document, an object modification detection has been introduced in DSS 5.10. The util detects all changes occurred within a PDF document after a concerned signature's or a timestamp's revision.

The detected modifications are categorized to four categories, depending on the "insecurity" level:

- **Extension change** is a secure change defining modification occurred within a PDF document for signature augmentation reasons (e.g. a document timestamp or a /DSS dictionary revision was added);
- **Signature or Form fill** is a change occurred for a signature, visual timestamp creation or

available form filling (can be restricted by /DocMDP dictionary);

- **Annotation creation change** defines a created or modified annotation (can be restricted by /DocMDP dictionary);
- **Undefined change** defines a modification that could not be categorized to the upper three categories. It is recommended to investigate the modification in details.

The following constraints are available to verify the validity of a signature based on encountered object modifications:

- `<DocMDP>` - when a `/DocMDP` dictionary is present within a signature, verifies whether the performed modifications in a PDF document are permitted according to the defined level.
- `FieldMDP` - when a `/FieldMDP` dictionary is present within a signature, verifies whether the performed modifications in a PDF document are permitted according to the defined level.
- `SigFieldLock` - when a `/FieldMDP` dictionary is present within a signature, verifies whether the performed modifications in a PDF document are permitted.
- `UndefinedChanges` - verifies whether the document contains modifications that cannot be unambiguously identified.

7.6.1.3. Password-protected documents

PDF files can be protected using a password. However, DSS does not support the validation of PAdES signatures on PDF documents protected by a password. DSS has no functionality allowing the user to enter the password when submitting a PDF for validation. Thus, the validation of such documents fails.

8. Requesting a timestamp token in DSS

Timestamping is essential when creating digital signatures that need to be preserved. Refer to section Timestamping for information about the general principles of the timestamping process. The following sections present how a timestamp token can be requested in DSS.

8.1. Configuring timestamp sources

The DSS framework proposes a `TSPSource` interface to implement the communication with a Time Stamp Authority (see section TSA for more information on Time Stamp Authorities). The class `OnlineTSPSource` is the default implementation of `TSPSource` using a HTTP(S) communication layer.

The following snippet of Java code illustrates how you might use this class:

OnlineTSPSource use

```
final String tspServer = "http://dss.nowina.lu/pki-factory/tsa/good-tsa";
OnlineTSPSource tspSource = new OnlineTSPSource(tspServer);
tspSource.setDataLoader(new TimestampDataLoader()); // uses the specific content-type

final DigestAlgorithm digestAlgorithm = DigestAlgorithm.SHA256;
final byte[] toDigest = "Hello world".getBytes("UTF-8");
final byte[] digestValue = DSSUtils.digest(digestAlgorithm, toDigest);
final TimestampBinary tsBinary = tspSource.getTimeStampResponse(digestAlgorithm,
digestValue);

LOG.info(DSSUtils.toHex(tsBinary.getBytes()));
```

8.1.1. Timestamp policy

A time-stamp policy is a "named set of rules that indicates the applicability of a time-stamp token to a particular community and/or class of application with common security requirements". A TSA may define its own policy which enhances the policy defined in RFC 3628. Such a policy shall incorporate or further constrain the requirements identified in RFC 3628. The user may request the TSA to issue a timestamp under a specific time-stamp policy that is supported by the TSA.

Timestamp policy

```
OnlineTSPSource tspSource = new OnlineTSPSource(tspServer);
tspSource.setPolicyOid("0.4.0.2023.1.1"); // provide a policy OID
```

8.1.2. Composite TSP sources

Sometimes timestamping servers may encounter interruptions (e.g. restart, configuration issues, etc.). To avoid failing signature augmentation, DSS allows a user to configure several TSP Sources. DSS will try one source after the other until getting a usable timestamp token.

Configuration of a CompositeTSPSource

```
// Create a map with several TSPSources
TimestampDataLoader timestampDataLoader = new TimestampDataLoader();// uses the
specific content-type

OnlineTSPSource tsa1 = new OnlineTSPSource("http://dss.nowina.lu/pki-factory/tsa/ee-
good-tsa");
tsa1.setDataLoader(timestampDataLoader);
OnlineTSPSource tsa2 = new OnlineTSPSource("http://dss.nowina.lu/pki-factory/tsa/good-
tsa");
tsa2.setDataLoader(timestampDataLoader);

Map<String, TSPSource> tspSources = new HashMap<>();
tspSources.put("TSA1", tsa1);
tspSources.put("TSA2", tsa2);

// Instantiate a new CompositeTSPSource and set the different sources
CompositeTSPSource tspSource = new CompositeTSPSource();
tspSource.setTspSources(tspSources);

final DigestAlgorithm digestAlgorithm = DigestAlgorithm.SHA256;
final byte[] toDigest = "Hello world".getBytes("UTF-8");
final byte[] digestValue = DSSUtils.digest(digestAlgorithm, toDigest);

// DSS will request the tsp sources (one by one) until getting a valid token.
// If none of them succeeds, a DSSException is thrown.
final TimestampBinary tsBinary = tspSource.getTimeStampResponse(digestAlgorithm,
digestValue);

LOG.info(DSSUtils.toHex(tsBinary.getBytes()));
```

9. Standalone timestamping

The DSS framework allows an independent document timestamping (without a signature). These are standalone time assertions, i.e. that no augmentation to the level `BASELINE-T` nor a creation of a signature to this level occurs.

The following Document Signature Services support the standalone timestamping :

- `PAdESService` - adds a timestamp to a PDF document;
- `ASiCWithCAdESService` - creates a timestamped ASiC container with provided documents.

DSS also provides a validation service for timestamped documents.

9.1. Timestamping a PDF

When timestamping a PDF document, a standalone timestamp can be used, creating a new revision.

This algorithm ensures that no existing signature nor timestamp will not be broken, for example because of adding the timestamp to the existing CMS signature (as it can be done in CAdES or XAdES, for instance). The same timestamping procedure is used for timestamping a PDF document without embedded signatures.

The code below illustrates a time-stamping process for a PDF document.

PDF timestamping

```
// Loads a document to be timestamped
DSSDocument documentToTimestamp = new FileDocument(new File("src/main/resources/hello-
world.pdf"));

// Configure a PAdES service for PDF timestamping
PAdESService service = new PAdESService(getCompleteCertificateVerifier());
service.setTspSource(getGoodTsa());

// Execute the timestamp method
DSSDocument timestampedDoc = service.timestamp(documentToTimestamp, new
PAdESTimestampParameters());
```

9.2. Timestamping with a container (ASiC)

Standalone time assertions can be used in both `ASiC-S` and `ASiC-E` containers: * In `ASiC-S` a timestamp is created on the original document or a ZIP-archive containing the original documents; * In `ASiC-E` a timestamp is created on a Manifest file listing the multiple data objects included in the container.

A typical example illustrating a time-stamping process that encapsulates the provided documents and the generated time-stamp to an ASiC-E container can be found below

ASIC-E time assertion

```
// Loads document(s) to be timestamped
DSSDocument documentToTimestampOne = new FileDocument(new
File("src/main/resources/hello-world.pdf"));
DSSDocument documentToTimestampTwo = new FileDocument(new
File("src/main/resources/xml_example.xml"));

// Configure the ASiCWithCAdESService service for documents timestamping within a
container
ASiCWithCAdESService service = new
ASiCWithCAdESService(getCompleteCertificateVerifier());
service.setTspSource(getGoodTsa());

// Initialize parameters and define target container type
ASiCWithCAdESTimestampParameters timestampingParameters = new
ASiCWithCAdESTimestampParameters();

// Specify the target container level
timestampingParameters.aSiC().setContainerType(ASiCContainerType.ASiC_E);

// Execute the timestamp method
DSSDocument timestampedDoc = service.timestamp(
        Arrays.asList(documentToTimestampOne, documentToTimestampTwo),
timestampingParameters);
```

In order to create an ASiC-S, just change the expected container property in the example above:

ASIC-S time assertion

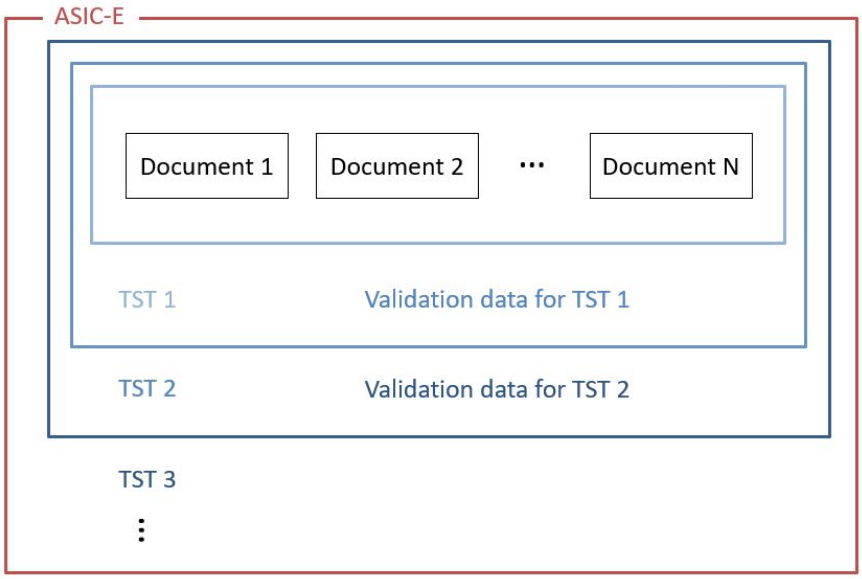
```
timestampingParameters.aSiC().setContainerType(ASiCContainerType.ASiC_S);
```

9.3. Standalone timestamps repetition

It is also worth noting that time assertions can cover each other, i.e. that a timestamp can be added over previously created timestamps. In this case, the validation data for a timestamp is incorporated within the previous (last) time assertion and the digest of that timestamp is added to the new created ArchiveManifest XML file, which is covered by a new timestamp.

This is a procedure similar to the augmentation of ASiC with CAdES with multiple LTA levels. LTA timestamps are created in different time-assertion files instead of an archive-time-stamp attribute like it is the case in a CAdES signature.

This concept is illustrated in the following schema using the ASiC format as an example



T1. Creation of the ASIC-E with time-assertion: A first timestamp token (TST 1) covering the documents present in the container is added to the latter.

T2. Addition of a timestamp token: Before the expiration of TST 1, the validation data for that timestamp is added to the container. A second timestamp (TST 2) is added to the container and covers all the documents, TST 1 and the validation data for TST 1.

T3. Addition of a timestamp token: Before the expiration of TST 2, the validation data for that timestamp is added to the container. A third timestamp (TST 3) is added to the container and covers all the documents, TST 1, the validation data for TST 1, TST 2 and the validation data for TST 2.

9.4. Standalone timestamp validation

As well as a single timestamp creation, DSS provides a validation service for timestamped documents. The timestamp validation process represents the one described in section "5.4 Time-stamp validation building block" of [R09]. The validation process is similar to the signature validation process. An appropriate validator will be selected automatically. In total, DSS supports timestamp-alone validation for the following file formats:

- Detached CMS timestamp (`DetachedTimestampValidator`) - a detached signed content must be provided (or its digest);
- PDF document (`PDFDocumentValidator`);
- ASiC CAdES container with a timestamp (`ASiCWithCAdESTimestampValidator`).

The validation process can be run with the following inputs:

Timestamped document validation

```
// Load a document validator. The appropriate validator class will be determined
automatically.
SignedDocumentValidator validator =
SignedDocumentValidator.fromDocument(timestampedDoc);
// Configure the validator. Provide a certificate verifier.
validator.setCertificateVerifier(getCompleteCertificateVerifier());
// Validate the document
Reports reports = validator.validateDocument();
```

The produced reports use the same structure as for the signature validation reports.

You can find an example of a produced timestamp Detailed Report here.

10. Signature augmentation

Signature augmentation is a process of adding material to go from one signature class to another. Signature augmentation is used for extending the validity of a signature in order to allow its long-term validation.

10.1. Configuration of the augmentation process

10.1.1. What are the needed external resources

10.1.1.1. TSA

To augment a signature to levels `BASELINE-T` and `BASELINE-LTA` you shall indicate to the service the TSA source, which delivers for each Timestamp Request a Timestamp Response (RFC 3161 (cf. [R08])) containing tokens. See chapter Requesting a timestamp token in DSS for more details.

10.1.1.2. Revocation sources

To augment a signature to level `BASELINE-LT` you need to configure revocation sources. See chapter Revocation data management for more information.

10.1.2. CertificateVerifier properties

The `CertificateVerifier` and its implementation `CommonCertificateVerifier` determines how DSS accesses the external resources and how it should react in some occasions. This configuration is used in both augmentation and validation mode. See section CertificateVerifier configuration for more information.

10.2. Augmenting an AdES baseline B signature

The `BASELINE-B` level contains immutable signed attributes. Once this level is created, these attributes cannot be changed. The levels `BASELINE-T/-LT/-LTA` add unsigned attributes to the signature. This means that the attributes of these levels could be added afterwards to any AdES signature. These unsigned attributes help protect the signature so that it can be validated over a longer period of time. Refer to section Signature classes/levels for more information on the four signature levels.

The augmentation of the signature is incremental, i.e. when augmenting the signature to the `BASELINE-LT` level, the lower level `BASELINE-T` will also be added.

The whole augmentation process is implemented by reusing components from signature creation. To augment a signature we proceed in the same way as in the case of a signature creation, except that you have to call the function "extendDocument" instead of the "sign" function.

When a document is signed with several signatures, all the signatures are augmented. Augmenting a set of selected signatures is not supported in DSS.

10.2.1. To baseline T

AdES-BASELINE-T is a signature for which there exists a trusted time associated to the signature (cf. Signature classes/levels). 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 during the signature creation process as well as during the signature augmentation process.

The AdES-BASELINE-T form must be built on an AdES-BASELINE-B form. The DSS framework allows augmenting the old -BES and -EPES profiles "as if" they were baseline profiles. The added components/attributes are the same, but the created signature is different (e.g. no claimed signing time). Moreover, there is some more support for XAdES extended profiles where the user can select the target augmentation profile.

The framework adds the timestamp only if there is no timestamp yet or there is one but the creation of a new augmentation of the level BASELINE-T is deliberate (using another TSA). It is not possible to augment a signature to the BASELINE-T level if it already incorporates a higher level, i.e. BASELINE-LT or BASELINE-LTA. In theory, it would be possible to add another BASELINE-T level when the signature has already reached level BASELINE-LT but the framework prevents this operation.

Below is the source code that creates a XAdES-BASELINE-T signature. For our example, we need to initialize an instance of OnlineTSPSource (that has already been configured).

Augment a XAdES signature

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

CommonCertificateVerifier certificateVerifier = new CommonCertificateVerifier();

// Init service for signature augmentation
XAdESService xadesService = new XAdESService(certificateVerifier);

// init TSP source for timestamp requesting
xadesService.setTspSource(getOnlineTSPSource());

DSSDocument tLevelSignature = xadesService.extendDocument(signedDocument, parameters);
```

Here is the result of adding a new extension of type BASELINE-T to an already existing BASELINE-T level signature (for XAdES):

XAdES Unsigned Signature Properties

```
<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>
```

10.2.2. To baseline LT

The `AdES-BASELINE-LT` profile implements the signature class *Signature with Long-Term Validation Material* (cf. Signature classes/levels). Augmenting to the `AdES-BASELINE-LT` will add the `CertificateValues` and `RevocationValues` unsigned qualifying properties to the signature.

- 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 (in case of XAdES, or within an alternative element for another format) of the signature.
- The `RevocationValues` element includes the sources of CRL and/or OCSP.

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 (e.g. a source of trusted certificates). Please refer to the CertificateVerifier configuration section for further information.

The code below shows how to use the default parameters with the `CertificateVerifier` object and how to implement the `BASELINE-LT` level of signature:

SignXmlXadesLTTest.java

```
parameters = new XAdESSignatureParameters();
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_LT);

certificateVerifier = new CommonCertificateVerifier();

CommonsDataLoader commonsDataLoader = new CommonsDataLoader();

// init revocation sources for CRL/OCSP requesting
certificateVerifier.setCrlSource(new OnlineCRLSource(commonsDataLoader));
certificateVerifier.setOcspSource(new OnlineOCSPSource());

// Trust anchors should be defined for revocation data requesting
certificateVerifier.setTrustedCertSources(getTrustedCertificateSource());

xadesService = new XAdESService(certificateVerifier);
xadesService.setTspSource(getOnlineTSPSource());
DSSDocument ltLevelDocument = xadesService.extendDocument(tLevelSignature,
parameters);
```

The following XML segment will be added to the signature qualified and unsigned properties (for XAdES):

Validation data values

```
<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 the previous code example, the `CommonsDataLoader` is used to provide the communication layer for

various protocols (e.g. HTTP, HTTPS, LDAP, LDAPS). Each source that requires to go through the network to retrieve data needs to have this component set.

10.2.3. To baseline LTA.

The `AdES-BASELINE-LTA` profile implements the signature class *Signature providing Long Term Availability and Integrity of Validation Material* (cf. Signature classes/levels). In practice, the augmentation to `BASELINE-LTA` level is made by adding timestamps and optionally additional validation data to protect all the validation data incorporated at `BASELINE-LT` level. For example, this augmentation should happen before one of the certificates arrives to its expiration date or when there is a risk of cryptographic obsolescence of the algorithms and parameters used.

E.g. `XAdES-BASELINE-LTA` level adds the `ArchiveTimeStamp` element within the `UnsignedSignatureProperties` and may contain several `EncapsulatedTimeStamp` elements.

Below is an example of the implementation of this level of signature:

Signature level setting

```
parameters = new XAdESSignatureParameters();
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_LTA);

// Initialize CertificateVerifier with data revocation data requesting
certificateVerifier = new CommonCertificateVerifier();

certificateVerifier.setCrlSource(new OnlineCRLSource());
certificateVerifier.setOcspSource(new OnlineOCSPSource());

certificateVerifier.setTrustedCertSources(getTrustedCertificateSource());

// Initialize signature service with TSP Source for time-stamp requesting
xadesService = new XAdESService(certificateVerifier);
xadesService.setTspSource(getOnlineTSPSource());

DSSDocument ltaLevelDocument = xadesService.extendDocument(ltLevelDocument,
parameters);
```

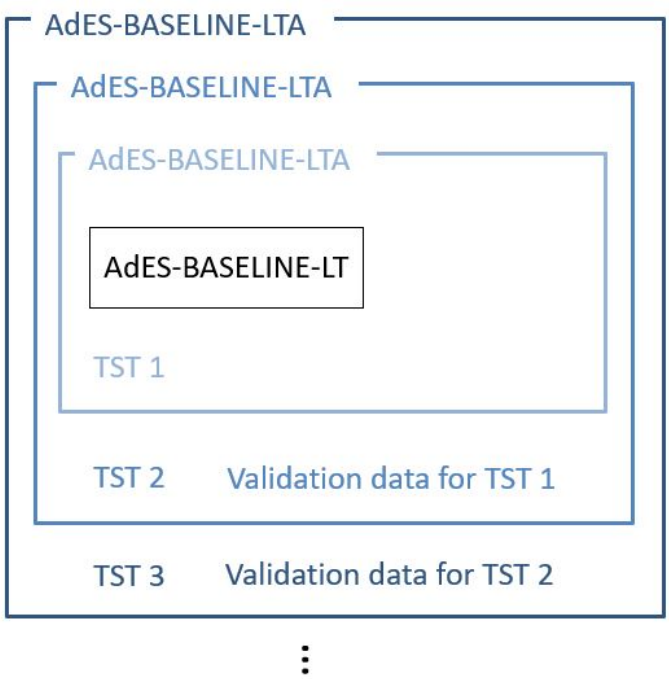
The following XML segment will be added to the signature qualified and unsigned properties (for XAdES):

XAdES Archive Timestamp

```
<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>
```

The time-stamping process may be repeated every time the protection used becomes weak. A new time-stamp needs to be affixed before either the signing certificate of the TSA is expired or the algorithms used by the TSA of the previous time-stamp have become obsolete. The new timestamp and validation material are added into the existing `BASELINE-LTA` signature.

This concept of `BASELINE-LTA` repetition is illustrated in the following schema.



T1. Creation of the AdES-BASELINE-LTA: A first timestamp token (TST 1) covering the AdES-B-LT signature is added to the latter.

T2. Addition of a timestamp token: Before the expiration of TST 1, the validation data for that timestamp is added to the signature as well as a second timestamp (TST 2) that covers the signature, TST 1 and the validation data for TST 1.

T3. Addition of a timestamp token: Before the expiration of TST 2, the validation data for that timestamp is added to the signature as well as a third timestamp (TST 3) that covers the signature, TST 1, the validation data for TST 1, TST 2 and the validation data for TST 2.

10.3. Creating a baseline T signature

The common process is a creation of a `-B` level signature and then augmenting it, when necessary, with superior levels. However, the framework allows signing directly with any level. Thus, a direct signature creation to `BASELINE-T` level can benefit the signer with providing a *best-signature-time* (and a proof of existence of the signature, respectively) as close as possible to the claimed signing time.

Let's see an example of signing with the level `BASELINE-T`.

Create a XAdES-BASELINE-T with an OnlineTSPSource

```
// 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://dss.nowina.lu/pki-factory/tsa/good-tsa";
OnlineTSPSource onlineTSPSource = new OnlineTSPSource(tspServer);
onlineTSPSource.setDataLoader(new TimestampDataLoader()); // uses the specific
content-type
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);
```

The timestamp source shall be defined for `BASELINE-T` and `BASELINE-LTA` levels creation.

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

XAdES Signature Timestamp

```
<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>
```

10.4. Best practices regarding baseline levels

For signature creation, it is recommended to use a BASELINE-T level in order to provide the proof of existence (POE) to the signature with a signature time-stamp. This will ensure that the signature has been made during the validity period of the signing-certificate. The BASELINE-B level should not be used for signatures that need to be valid over a longer period of time because it does not contain a timestamp to prove the time of signing.

It is not recommended using BASELINE-LT or higher level on signature creation. The BASELINE-LT level should be added to a signature only after a revocation data update, so that revocation's thisUpdate parameter value will be after the *best-signing-time* provided by the signature time-stamp on BASELINE-T level. This is a requirement of a revocation freshness constraint defined in ETSI EN 319 102-1 (cf. [R09]) and enforced by TS 119 172-4 (cf. [R10]) with a value '0'. Since the lower levels are also added when the signature is augmented to a certain level, the BASELINE-LTA level should not be used for signature creation either as it would add BASELINE-LT too.

The AdES-BASELINE-T trusted time indications should be created before the signing certificate has been revoked or expired and close to the time that the AdES signature was produced.

- If the signing certificate has expired before the trusted time indications have been added, it will not be possible to validate the signature anymore. This is why the timestamp should be created before the expiration of the signing certificate.
- While the expiration date is known since the moment of the creation of the certificate, a certificate can be revoked at any time. To avoid that the certificate gets revoked before the creation of the timestamp, it is important to create the timestamp close to the time that the AdES signature was created.

After the revocation data update, satisfying the revocation freshness constraint, the BASELINE-LT level can be incorporated to the signature, providing the information about validity of the used certificate chains. In order to prove the existence of the revocation data, the BASELINE-LTA level should be added after. The BASELINE-LTA level will prove the existence of the incorporated revocation data, so it can be trusted by the validators even after its expiration, as well as will provide a POE for cryptographic constraints and all the previous certificate chains, including the ones used for timestamp creation.

11. Trusted Lists

11.1. Configuration of TL validation job

The `TLValidationJob` allows downloading, parsing and validating the Trusted List(s) (TL) (cf. EU MS Trusted List) and List(s) Of Trusted Lists (LOTL) (cf. List of Trusted Lists (LOTL)). Once the task is done, its result is stored in the `TrustedListsCertificateSource`. The job uses 3 different caches (download, parsing and validation) and a state-machine to be efficient.

Trusted lists are stored in the file system. That offers the possibility to run the validation job in offline mode with the stored trusted lists. Trusted Lists can be loaded from the file system and/or from Internet.

In the next sections the different configurations will be covered.

11.1.1. TLSource and LOTLSource

Several TLSources and several LOTLSources can be injected in a `TLValidationJob`. The only constraint is the uniqueness of the Trusted List URLs.

Multiple TLSources and multiple LOTLSources configuration

```
TLValidationJob validationJob = new TLValidationJob();
// Specify where the TL/LOTL is hosted and which are the signing certificate(s) for
these TL/LOTL.
validationJob.setTrustedListSources(boliviaTLSource(), costaRicaTLSource());
validationJob.setListOfTrustedListSources(europeanLOTLSource(),
unitedStatesLOTLSource());
```

11.1.1.1. Trusted List Source (TLSource)

A `TLSource` allows quickly setting up a trusted list configuration. The URL and the signing certificates for this TL are mandatory. Optionally, predicates or/and filters can be configured to retrieve only a part of the trust service providers or trust services.

TLSource configuration

```
TLSource tlSource = new TLSource();

// Mandatory : The url where the TL needs to be downloaded
tlSource.setUrl("http://www.ssi.gouv.fr/eidas/TL-FR.xml");

// A certificate source which contains the signing certificate(s) for the
// current trusted list
tlSource.setCertificateSource(getSigningCertificatesForFrenchTL());

// Optional : predicate to filter trust services which are/were granted or
// equivalent (pre/post eIDAS).
// Input : implementation of TrustServicePredicate interface.
// Default : none (select all)
tlSource.setTrustServicePredicate(new GrantedTrustService());

// Optional : predicate to filter the trust service providers
// Input : implementation of TrustServiceProviderPredicate interface.
// Default : none (select all)
tlSource.setTrustServiceProviderPredicate(new CryptologOnlyTrustServiceProvider());

//instance of CertificateSource where all trusted certificates and their properties
(service type,...) are stored.
tlValidationJob.setTrustedListSources(tlSource);
```

11.1.1.2. List Of Trusted Lists Source (LOTLSource)

A similar configuration is possible for Lists Of Trusted Lists (LOTL) with a `LOTLSource`. That requires an URL and a list of potential LOTL signers. Some other parameters are also possible. By default, all listed trusted lists are loaded.

LOTLSource configuration

```
LOTLSource lotlSource = new LOTLSource();

// Mandatory : The url where the LOTL needs to be downloaded
lotlSource.setUrl("https://ec.europa.eu/tools/lotl/eu-lotl.xml");

// A certificate source which contains the signing certificate(s) for the
// current list of trusted lists
lotlSource.setCertificateSource(getSigningCertificatesForEuropeanLOTL());

// true or false for the pivot support. Default = false
// More information :
// https://ec.europa.eu/tools/lotl/pivot-lotl-explanation.html
lotlSource.setPivotSupport(true);

// Optional : the predicate which allows to find the LOTL definition in the LOTL
// Input : implementation of Predicate<OtherTSLPointerType> interface (e.g.
```

```
OtherTSLPointerPredicate)
// Default : European configuration
lotlSource.setLotlPredicate(new EULOTLOtherTSLPointer().and(new
XMLOtherTSLPointer()));

// Optional : the predicate which allows to find and/or filter the TL
// definitions in the LOTL
// Input : implementation of Predicate<OtherTSLPointerType> interface (e.g.
OtherTSLPointerPredicate)
// Default : all found trusted lists in the European LOTL
lotlSource.setTlPredicate(new EUTLOtherTSLPointer().and(new XMLOtherTSLPointer()));

// Optional : a predicate which allows to find back the signing certificates for
// the current LOTL
// Input : implementation of LOTLSigningCertificatesAnnouncementSchemeInformationURI
interface.
// Default : not defined
//
// OfficialJournalSchemeInformationURI allows to specify the Official Journal
// URL where the signing certificates are published
lotlSource.setSigningCertificatesAnnouncementPredicate(
        new OfficialJournalSchemeInformationURI("https://eur-lex.europa.eu/legal-
content/EN/TXT/?uri=uriserv:OJ.C_.2019.276.01.0001.01.ENG"));

// Optional : predicate to filter trust services which are/were granted or
// equivalent (pre/post eIDAS). This parameter is applied on the related trusted
// lists
// Input : implementation of TrustServicePredicate interface.
// Default : none (select all)
lotlSource.setTrustServicePredicate(new GrantedTrustService());

// Optional : predicate to filter the trust service providers. This parameter is
// applied on the related trusted lists
// Input : implementation of TrustServiceProviderPredicate interface.
// Default : none (select all)
lotlSource.setTrustServiceProviderPredicate(new CryptologOnlyTrustServiceProvider());

tlValidationJob.setListOfTrustedListSources(lotlSource);
```

11.1.2. DSSFileLoader

The `FileCacheDataLoader` is used to download the trusted list contents in the file system. Two different configurations are needed. Both of them share the same folder:

- **offline refresh**: disabled download from Internet and unlimited cache expiration
- **online refresh**: enabled download from Internet and limited cache expiration

Offline and Online refresh configuration

```
public DSSFileLoader offlineLoader() {
    FileCacheDataLoader offlineFileLoader = new FileCacheDataLoader();
    offlineFileLoader.setCacheExpirationTime(-1); // negative value means cache never
expires
    offlineFileLoader.setDataLoader(new IgnoreDataLoader()); // do not download from
Internet
    offlineFileLoader.setFileCacheDirectory(tlCacheDirectory());
    return offlineFileLoader;
}

public DSSFileLoader onlineLoader() {
    FileCacheDataLoader onlineFileLoader = new FileCacheDataLoader();
    onlineFileLoader.setCacheExpirationTime(0);
    onlineFileLoader.setDataLoader(dataLoader()); // instance of DataLoader which can
access to Internet (proxy,...)
    onlineFileLoader.setFileCacheDirectory(tlCacheDirectory());
    return onlineFileLoader;
}
```

11.1.3. The SynchronizationStrategy

The `SynchronizationStrategy` defines the trusted lists or list(s) of trusted lists to be synchronized. By default, DSS synchronizes all of them and DSS does not reject any trusted lists (e.g. expired, invalid, etc.). The default behavior can be customized with implementation of the `SynchronizationStrategy`.

DSS provides two implementations within the framework:

- `ExpirationAndSignatureCheckStrategy` - rejects Trusted Lists with invalid signature or expired. The certificates from a such Trusted List are not loaded to the `TrustedListsCertificateSource`.
- `AcceptAllStrategy` (default) - accepts all Trusted List, whatever the validation status is.

An example of a custom implementation of `SynchronizationStrategy` can be found below:

Example of a custom SynchronizationStrategy

```
public SynchronizationStrategy allValidTrustedListsStrategy() {

    return new SynchronizationStrategy() {

        @Override
        public boolean canBeSynchronized(TLInfo trustedList) {
            return trustedList.getValidationCacheInfo().isValid();
        }

        @Override
        public boolean canBeSynchronized(LOTLInfo listOfTrustedList) {
            return listOfTrustedList.getValidationCacheInfo().isValid();
        }

    };

}
```

11.1.4. The CacheCleaner

The `CacheCleaner` specifies how DSS clears the cache (e.g. in case of expired URL, etc.). Two cleaning options are available : memory and file system.

CacheCleaner Configuration

```
public CacheCleaner cacheCleaner() {
    CacheCleaner cacheCleaner = new CacheCleaner();

    cacheCleaner.setCleanMemory(true); // free the space in memory

    cacheCleaner.setCleanFileSystem(true); // remove the stored file(s) on the file-
system

    // if the file-system cleaner is enabled, inject the configured loader from the
    // online or offline refresh data loader.
    cacheCleaner.setDSSFileLoader(offlineLoader());

    return cacheCleaner;
}
```

11.1.5. Alerting from TL Loading

DSS allows running of custom alerts in some situations (e.g. invalid TL signature, LOTL location change, etc.). Alert works with two concepts: detection and alert handler. After the download/parsing/validation and before the synchronization, the results are tested to detect events and launch alert(s).

Examples of Alerting

```
TLValidationJob job = new TLValidationJob();
// ...

// Add a log message in case of invalid signatures
TLAlert tlBrokenSignatureAlert = new TLAlert(new TLSignatureErrorDetection(), new
LogTLSignatureErrorAlertHandler());

// Send an email in case of new Official Journal detected
AlertHandler<LOTLInfo> mailSender = new AlertHandler<LOTLInfo>() {

    @Override
    public void process(LOTLInfo currentInfo) {
        String newOJUrl =
currentInfo.getParsingCacheInfo().getSigningCertificateAnnouncementUrl();
        // code to send an email
        SampleUtils.sendEmail(newOJUrl);
    }

};

// The europeanLOTLSource is configured with an
// OfficialJournalSchemeInformationURI
LOTLAlert officialJournalDesynchronizationAlert = new LOTLAlert(new
OJUrlChangeDetection(europeanLOTLSource()), mailSender);

// Update a database in case of LOTL location change
AlertHandler<LOTLInfo> databaseUpgrader = new AlertHandler<LOTLInfo>() {

    @Override
    public void process(LOTLInfo currentInfo) {
        String newLOTLUrl = null;

        String currentLOTLUrl = currentInfo.getUrl();
        List<PivotInfo> pivots = currentInfo.getPivotInfos();
        for (PivotInfo pivot : pivots) {
            if (!Utils.areStringsEqual(currentLOTLUrl, pivot.getLOTLLocation())) {
                newLOTLUrl = pivot.getLOTLLocation();
                break;
            }
        }

        // code to update a database
        SampleUtils.updateDatabase(newLOTLUrl);
    }

};
LOTLAlert lotlLocationChangeAlert = new LOTLAlert(new
LOTLLocationChangeDetection(europeanLOTLSource()), databaseUpgrader);
```

```
// add all alerts on the job
job.setLOTLAlerts(Arrays.asList(officialJournalDesynchronizationAlert,
lotlLocationChangeAlert));
job.setTLAlerts(Arrays.asList(tlBrokenSignatureAlert));
```

See section Use of Alerts throughout the framework in the Annex for more information on related alerts.

11.1.6. LOTL/TL filter predicates

TSL predicates provide an option to filter the extracted TSL Pointers from LOTL or TL sources, allowing a customization of a trusted certificates and trusted services loading.

The following predicates are provided within the framework:

- `EULOTLOtherTSLPointer` - filters the EU LOTL pointer;
- `EUTLOtherTSLPointer` - filters the EU TL pointers;
- `MimetypeOtherTSLPointer` - filters TL pointers by a MimeType (e.g. to filter XML files only);
- `XMLOtherTSLPointer` - filters XML TL pointers;
- `PDFOtherTSLPointer` - filters PDF TL pointers;
- `SchemeTerritoryOtherTSLPointer` - filters TL pointers with a specific scheme territory (i.e. filter by country).

Examples of TSL Loading Predicates configuration

```
// the predicates filter TSL pointers to XML documents with
// "http://uri.etsi.org/TrstSvc/TrustedList/TSLType/EUlistofthelists" type
lotlSource.setLotlPredicate(new EULOTLOtherTSLPointer().and(new
XMLOtherTSLPointer()));

// the predicates filter only TSL pointers with scheme territories "DE" (Germany) and
"RO" (Romania)
// to XML documents with "http://uri.etsi.org/TrstSvc/TrustedList/TSLType/EUgeneric"
type
lotlSource.setTlPredicate(new SchemeTerritoryOtherTSLPointer(Arrays.asList("DE","RO"))
        .and(new EULOTLOtherTSLPointer()).and(new XMLOtherTSLPointer()));
```

11.1.7. Executor Service

An Executor Service allows you to customize a way of the program execution on your Java machine, by configuring a number of possible threads to be running, await time and so on.

Executor Service

```
// Allows configuration of the execution process
// Default : Executors.newCachedThreadPool() is used
tlValidationJob.setExecutorService(Executors.newSingleThreadExecutor());
```

11.1.8. Complete configuration for the European LOTL

Below, you can find a complete configuration for the European List Of Trusted Lists. The URLs need to be externalized.

European LOTL Configuration

```
    // Should be externalized
    private static final String LOTL_URL = "https://ec.europa.eu/tools/lotl/eu-
lotl.xml";
    private static final String OJ_URL = "https://eur-lex.europa.eu/legal-
content/EN/TXT/?uri=uriserv:OJ.C_.2019.276.01.0001.01.ENG";

    @Test
    public void test() {
        CommonCertificateVerifier commonCertificateVerifier = new
CommonCertificateVerifier();
        TLValidationJob job = job();
        TrustedListsCertificateSource trustedListsCertificateSource = new
TrustedListsCertificateSource();
        job.setTrustedListCertificateSource(trustedListsCertificateSource);
        job.onlineRefresh();

commonCertificateVerifier.setTrustedCertSources(trustedListsCertificateSource);
        commonCertificateVerifier.setCrlSource(new OnlineCRLSource());
        commonCertificateVerifier.setOcspSource(new OnlineOCSPSource());
        commonCertificateVerifier.setAIASource(new DefaultAIASource());

        SignedDocumentValidator validator = SignedDocumentValidator.fromDocument(
                new FileDocument("src/test/resources/signature-
pool/signedXmlXadesB.xml"));
        validator.setCertificateVerifier(commonCertificateVerifier);

        validator.validateDocument();
    }

    public TLValidationJob job() {
        TLValidationJob job = new TLValidationJob();
        job.setOfflineDataLoader(offlineLoader());
        job.setOnlineDataLoader(onlineLoader());
        job.setTrustedListCertificateSource(trustedCertificateSource());
        job.setSynchronizationStrategy(new AcceptAllStrategy());
        job.setCacheCleaner(cacheCleaner());

        LOTLSource europeanLOTL = europeanLOTL();
        job.setListOfTrustedListSources(europeanLOTL);

        job.setLOTLAlerts(Arrays.asList(ojUrlAlert(europeanLOTL),
lotlLocationAlert(europeanLOTL)));
        job.setTLAlerts(Arrays.asList(tlSigningAlert(), tlExpirationDetection()));
```

```
        return job;
    }

    public TrustedListsCertificateSource trustedCertificateSource() {
        return new TrustedListsCertificateSource();
    }

    public LOTLSource europeanLOTL() {
        LOTLSource lotlSource = new LOTLSource();
        lotlSource.setUrl(LOTL_URL);
//      lotlSource.setCertificateSource(officialJournalContentKeyStore());
        lotlSource.setCertificateSource(new CommonCertificateSource());
        lotlSource.setSigningCertificatesAnnouncementPredicate(new
OfficialJournalSchemeInformationURI(OJ_URL));
        lotlSource.setPivotSupport(true);
        return lotlSource;
    }

    public CertificateSource officialJournalContentKeyStore() {
        try {
            return new KeyStoreCertificateSource(new
File("src/main/resources/keystore.p12"), "PKCS12", "dss-password");
        } catch (IOException e) {
            throw new DSSException("Unable to load the keystore", e);
        }
    }

    public DSSFileLoader offlineLoader() {
        FileCacheDataLoader offlineFileLoader = new FileCacheDataLoader();
        offlineFileLoader.setCacheExpirationTime(Long.MAX_VALUE);
        offlineFileLoader.setDataLoader(new IgnoreDataLoader());
        offlineFileLoader.setFileCacheDirectory(tlCacheDirectory());
        return offlineFileLoader;
    }

    public DSSFileLoader onlineLoader() {
        FileCacheDataLoader onlineFileLoader = new FileCacheDataLoader();
        onlineFileLoader.setCacheExpirationTime(0);
        onlineFileLoader.setDataLoader(dataLoader());
        onlineFileLoader.setFileCacheDirectory(tlCacheDirectory());
        return onlineFileLoader;
    }

    public File tlCacheDirectory() {
        File rootFolder = new File(System.getProperty("java.io.tmpdir"));
        File tslCache = new File(rootFolder, "dss-tsl-loader");
        if (tslCache.mkdirs()) {
            LOG.info("TL Cache folder : {}", tslCache.getAbsolutePath());
        }
        return tslCache;
    }
```

```
    public CommonsDataLoader dataLoader() {
        return new CommonsDataLoader();
    }

    public CacheCleaner cacheCleaner() {
        CacheCleaner cacheCleaner = new CacheCleaner();
        cacheCleaner.setCleanMemory(true);
        cacheCleaner.setCleanFileSystem(true);
        cacheCleaner.setDSSFileLoader(offlineLoader());
        return cacheCleaner;
    }

    // Optionally : alerting.
    // Recommended detections : OJUrlChangeDetection + LOTLLocationChangeDetection

    public TLAlert tlSigningAlert() {
        TLSignatureErrorDetection signingDetection = new TLSignatureErrorDetection();
        LogTLSignatureErrorAlertHandler handler = new
LogTLSignatureErrorAlertHandler();
        return new TLAlert(signingDetection, handler);
    }

    public TLAlert tlExpirationDetection() {
        TLExpirationDetection expirationDetection = new TLExpirationDetection();
        LogTLExpirationAlertHandler handler = new LogTLExpirationAlertHandler();
        return new TLAlert(expirationDetection, handler);
    }

    public LOTLAlert ojUrlAlert(LOTLSource source) {
        OJUrlChangeDetection ojUrlDetection = new OJUrlChangeDetection(source);
        LogOJUrlChangeAlertHandler handler = new LogOJUrlChangeAlertHandler();
        return new LOTLAlert(ojUrlDetection, handler);
    }

    public LOTLAlert lotlLocationAlert(LOTLSource source) {
        LOTLLocationChangeDetection lotlLocationDetection = new
LOTLLocationChangeDetection(source);
        LogLOTLLocationChangeAlertHandler handler = new
LogLOTLLocationChangeAlertHandler();
        return new LOTLAlert(lotlLocationDetection, handler);
    }
```

11.1.9. The TL / LOTL refresh

The TL / LOTL loading in DSS works as below :

- Download / parse / validate all LOTLSources from the configuration with/without pivot support (multi-threaded);
- Analyze introduced changes and expired cache entries (new TL URLs, new signing certificates

for a TL, etc.);

- Create TLSources from the retrieved LOTLs;
- Combine these TLSources with independent TLSources (from the configuration);
- Download / parse / validate all TLs (multi-threaded);
- If alerts are configured, test if an alert needs to be launched;
- If the debug is enabled, print in the log the cache status;
- Synchronize the `TrustedListCertificateSource`;
- If the cache cleaner is configured, execute it;
- If the debug is enabled, print in the log the cache status.

The refresh can be called with the offline or the online loader and run exactly the same code:

How to refresh the Trusted List(s) and Lists of Trusted Lists

```
TLValidationJob validationJob = new TLValidationJob();

// call with the Offline Loader (application initialization)
validationJob.offlineRefresh();

// call with the Online Loader (callable every day/hour in a cron)
validationJob.onlineRefresh();
```

11.1.9.1. Java Keystore Management

Generally (like in case of European LOTL) DSS downloads Trusted Lists by using the SSL protocol (for resources using HTTPS extension), that requires to have a certificate of a remote source in the Java trust store. The certificates have their own validity period and can expire. If a certificate is expired, it will be replaced on a server by a new one in order to support a secure SSL connection. The easiest way to know if your Java trust store is outdated and new certificates need to be added is to check your logs during a `TLValidationJob` update :

```
ERROR 14052 --- [pool-2-thread-30] e.e.e.dss.tsl.runnable.AbstractAnalysis  : Unable
to process GET call for url [https://sr.riik.ee/tsl/estonian-tsl.xml]. Reason : [PKIX
path building failed: sun.security.provider.certpath.SunCertPathBuilderException:
unable to find valid certification path to requested target]
```

The `SunCertPathBuilderException` means that the certificate establishing the secure connection is not trusted by your Java Virtual Machine. In order to add the certificate to the trust store, you need to do the following steps (the example is based on Windows OS and Google Chrome browser):

1. Open the failed URL in your browser. In our case it will be 'https://sr.riik.ee/tsl/estonian-tsl.xml' obtained from the logs.
2. Click on a lock icon next to the URL in the tab you just opened. It will open a window about the current connection status.

3. Click on 'Certificate' button to open the Certificate window.
4. Go to 'Details' tab and choose 'Copy to File...'.
5. Process the 'Certificate Export Wizard', by saving the certificate in one of '.CER' formats. Store the file in your file system. For us it will create a file 'ee.cer'.
6. Run 'Command Prompt' with administrator permissions (right click → 'Run As Administrator').
7. Execute the following line (ensure that 'keytool' is installed):

Certificate import

```
keytool -import -alias newCert -file pathToCert\ee.cer -keystore
pathToJavaDirectory\lib\security\cacerts -storepass changeit
```

The default password for a Java keystore is `changeit`. Ensure that you have a default configuration, or use another password you have configured.

In order to apply changes, the application using Java must be rebooted.

After these steps the `TLValidationJob` will successfully download the target Trusted List (i.e. Estonian in our example).

This described algorithm is not only one available solution, if you have difficulties with this, you can search in the Internet for another working for you solution.

11.1.10. TLValidationJobSummary

The class `TLValidationJobSummary` contains all processed data about the download (time, error, etc.), the parsing (extracted information, parsing error, etc.) and the signature validation (signing certificate, signing time, etc.).

How to retrieve the information about the TLValidationJob process

```
TrustedListsCertificateSource trustedListCertificateSource = new
TrustedListsCertificateSource();

TLValidationJob job = new TLValidationJob();
job.setTrustedListCertificateSource(trustedListCertificateSource);

// ... config & refresh ...

// A cache content summary can be computed on request
TLValidationJobSummary summary = job.getSummary();

// All information about processed LOTLSources
List<LOTLInfo> lotlInfos = summary.getLOTLInfos();
LOTLInfo lotlInfo = lotlInfos.get(0);
// All data about the download (last occurrence, cache status, error,...)
DownloadInfoRecord downloadCacheInfo = lotlInfo.getDownloadCacheInfo();

// All data about the parsing (date, extracted data, cache status,...)
ParsingInfoRecord parsingCacheInfo = lotlInfo.getParsingCacheInfo();

// All data about the signature validation (signing certificate, validation
// result, cache status,...)
ValidationInfoRecord validationCacheInfo = lotlInfo.getValidationCacheInfo();

// All information about processed TLSources (which are not linked to a
// LOTLSource)
List<TLInfo> otherTLInfos = summary.getOtherTLInfos();

// or the last update can be collected from the TrustedListsCertificateSource
TLValidationJobSummary lastSynchronizedSummary =
trustedListCertificateSource.getSummary();
```

11.2. Validation policy for trusted lists

An eIDAS XML node linked to the status of the Trusted List is included in the DSS validation policy (see AdES validation constraints/policy for more information). The following elements are contained in that node:

- **TLFreshness**: Given that TLs are too heavy to be downloaded at each validation, TLs are downloaded every few hours. However, it can happen that a download is not possible because the TL is not available or because there is a problem with the downloader on the signature validation algorithm side. For such situations, it is useful to display a message to indicate that the TL is not “fresh”. To achieve this, the TL freshness indicates that TLs were downloaded no later than the validation time minus the TL freshness time interval.
- **TLNotExpired**: The TL might be expired in case of an attack consisting in replacing the current TL by an older one. Using an old TL might bring problems. The expired TL could for example contain Certificate Authorities that are not present in the current TL. Thus, an expired TL

should lead to a warning during the validation.

- **TLWellSigned**: If the signature of the TL is not valid, the TL should be discarded. A non-valid TL signature might indicate that the content of that TL is "fake" and should not be trusted.
- **TLVersion**: If the version of the trusted list does not correspond to the version indicated in the validation policy, the validation process fails.

11.3. Using non-EU trusted lists

Non-EU trusted lists are supported by DSS. However, there are a few limitations:

- Non-EU trusted lists shall have the same XML structure as EU TLs, i.e. they shall be compliant with the XSD schema.
- There is no guarantee for a proper qualification determination as the non-EU TL shall also be compliant with EU regulations.

Even though the ETSI TS 119 615 standard ([R14]) is EU-specific, the diagnostic data contains the necessary information for implementers to implement non-EU ETSI TS 119 615 status determination.

11.4. Signing a trusted list

The standard ETSI TS 119 612 (cf. [R11]) specifies in its annex B the XML structure and the format of the signature (XAdES, enveloped signature, transformation, canonicalization, etc.). With the class `TrustedListSignatureParametersBuilder`, DSS is able to pre-configure the signature parameters to comply with the specifications and simplify the signature creation.

Sign a Trusted List with the TrustedListSignatureParametersBuilder

```
DSSDocument trustedList = new FileDocument("src/main/resources/trusted-list.xml");

DSSPrivateKeyEntry privateKeyEntry = signingToken.getKeys().get(0);
CertificateToken signingCertificate = privateKeyEntry.getCertificate();

// This class creates the appropriated XAdESSignatureParameters object to sign a
trusted list.
// It handles the configuration complexity and creates a ready-to-be-used
XAdESSignatureParameters with a correct configuration.
TrustedListSignatureParametersBuilder builder = new
TrustedListSignatureParametersBuilder(signingCertificate, trustedList);
XAdESSignatureParameters parameters = builder.build();

XAdESService service = new XAdESService(new CommonCertificateVerifier());

ToBeSigned dataToSign = service.getDataToSign(trustedList, parameters);
SignatureValue signatureValue = signingToken.sign(dataToSign,
parameters.getDigestAlgorithm(), privateKeyEntry);
DSSDocument signedTrustedList = service.signDocument(trustedList, parameters,
signatureValue);
```

12. eIDAS

12.1. Overview of certificates

12.1.1. Type of certificate

A certificate can be for electronic signature, for electronic seal or for website authentication.

12.1.2. Qualified status of certificate

Explanations as well as an example on the determination of the qualified status of a certificate are presented in section Validation of a certificate.

12.2. How certificate type and qualification are represented in DSS

12.2.1. Certificate Qualification determination

In order to determine a type and qualification of certificate, the `CertificateVerifier` can be used, provided the relevant information extracted from a Trusted List(s).

An example of a qualification data extraction for a certificate, can be found below:

Certificate qualification validation

```
// Configure the internet access
CommonsDataLoader dataLoader = new CommonsDataLoader();

// We set an instance of TrustAllStrategy to rely on the Trusted Lists content
// instead of the JVM trust store.
dataLoader.setTrustStrategy(TrustAllStrategy.INSTANCE);

// Configure the TLValidationJob to load a qualification information from the
corresponding LOTL/TL
TLValidationJob tlValidationJob = new TLValidationJob();
tlValidationJob.setOnlineDataLoader(new FileCacheDataLoader(dataLoader));

// Configure the relevant TrustedList
TLSource tlSource = new TLSource();
tlSource.setUrl("http://dss-test.lu");
tlValidationJob.setTrustedListSources(tlSource);

// Initialize the trusted list certificate source to fill with the information
extracted from TLValidationJob
TrustedListsCertificateSource trustedListsCertificateSource = new
TrustedListsCertificateSource();
tlValidationJob.setTrustedListCertificateSource(trustedListsCertificateSource);

// Update TLValidationJob
tlValidationJob.onlineRefresh();

// Thirdly, we need to configure the CertificateVerifier
CertificateVerifier cv = new CommonCertificateVerifier();
cv.setTrustedCertSources(trustedListsCertificateSource); // configured trusted list
certificate source
cv.setAIASource(aiaSource); // configured AIA Access
cv.setOcspSource(ocspSource); // configured OCSP Access
cv.setCrlSource(crlSource); // configured CRL Access

// Create an instance of CertificateValidator for the SSL Certificate with the
// CertificateVerifier
CertificateValidator validator = CertificateValidator.fromCertificate(certificate);
validator.setCertificateVerifier(cv);

// Validate the certificate
CertificateReports reports = validator.validate();
SimpleCertificateReport simpleReport = reports.getSimpleReport();

// Extract the qualification information
CertificateQualification qualificationAtCertificateIssuance =
simpleReport.getQualificationAtCertificateIssuance();
CertificateQualification qualificationAtValidationTime =
simpleReport.getQualificationAtValidationTime();

// Extract the requested information about a certificate type and its qualification
```

```
CertificateType type = qualificationAtValidationTime.getType();
boolean isQualifiedCertificate = qualificationAtValidationTime.isQc();
boolean isQSCD = qualificationAtValidationTime.isQscd();
```

12.2.2. Qualified certificate for WebSite Authentication (QWAC)

With DSS, it is possible to validate SSL certificate against the EUMS TL and the ETSI TS 119 615 (cf. [R14]) to determine if it is a Qualified certificate for WebSite Authentication (QWAC).

DSS provides a special class `SSLCertificateLoader` allowing to extract the SSL certificate chain from the given URL. The qualification verification is similar to the example defined in chapter Certificate Qualification determination.

Validate an SSL certificate and retrieve its qualification level

```
// Secondly, we create an instance of SSLCertificateLoader which is responsible
// for the SSL certificate(s) downloading.
SSLCertificateLoader sslCertificateLoader = new SSLCertificateLoader();
// We set the configured dataLoader
sslCertificateLoader.setCommonsDataLoader(dataLoader);

// We retrieve the SSL certificates for the given URL
List<CertificateToken> certificates =
sslCertificateLoader.getCertificates("https://www.microsec.hu");

CertificateToken sslCertificate = certificates.get(0);

// Add intermediate certificates as non-trusted certificates (adjunct)
CertificateSource adjunctCertSource = new CommonCertificateSource();
for (CertificateToken certificateToken : certificates) {
    adjunctCertSource.addCertificate(certificateToken);
}
cv.setAdjunctCertSources(adjunctCertSource);

// Create an instance of CertificateValidator for the SSL Certificate with the
// CertificateVerifier
CertificateValidator validator = CertificateValidator.fromCertificate(sslCertificate);
validator.setCertificateVerifier(cv);

CertificateReports reports = validator.validate();
SimpleCertificateReport simpleReport = reports.getSimpleReport();
```

12.3. Overview of AdES signatures

12.3.1. Type of AdES

Under eIDAS, there exist advanced electronic signatures and seals.

12.3.2. Qualified status of AdES signature

Below is an example of the validation and verification of the qualified status of an AdES signature:

12.4. How signature type and qualification are represented in DSS

12.4.1. Signature Qualification determination

In order to determine a type and qualification of a signature, an instance of `SignedDocumentValidator` can be used, provided the relevant information is extracted from a Trusted List(s).

An example of a qualification data extraction for a signature, can be found below:

Signature qualification validation

```
// Configure the internet access
CommonsDataLoader dataLoader = new CommonsDataLoader();

// We set an instance of TrustAllStrategy to rely on the Trusted Lists content
// instead of the JVM trust store.
dataLoader.setTrustStrategy(TrustAllStrategy.INSTANCE);

// Configure the TLValidationJob to load a qualification information from the
corresponding LOTL/TL
TLValidationJob tlValidationJob = new TLValidationJob();
tlValidationJob.setOnlineDataLoader(new FileCacheDataLoader(dataLoader));

// Configure the relevant TrustedList
TLSource tlSource = new TLSource();
tlSource.setUrl("http://dss-test.lu");
tlValidationJob.setTrustedListSources(tlSource);

// Initialize the trusted list certificate source to fill with the information
extracted from TLValidationJob
TrustedListsCertificateSource trustedListsCertificateSource = new
TrustedListsCertificateSource();
tlValidationJob.setTrustedListCertificateSource(trustedListsCertificateSource);

// Update TLValidationJob
tlValidationJob.onlineRefresh();

// No we need to configure the DocumentValidator
CertificateVerifier cv = new CommonCertificateVerifier();
cv.setTrustedCertSources(trustedListsCertificateSource); // configured trusted list
certificate source
cv.setAIASource(aiaSource); // configured AIA Access
cv.setOcspSource(ocspSource); // configured OCSP Access
cv.setCrlSource(crlSource); // configured CRL Access

// Create an instance of SignedDocumentValidator
DocumentValidator validator = SignedDocumentValidator.fromDocument(signedDocument);
validator.setCertificateVerifier(cv);

// Validate the signature
Reports reports = validator.validateDocument();
SimpleReport simpleReport = reports.getSimpleReport();

// Extract the qualification information
SignatureQualification signatureQualification =
simpleReport.getSignatureQualification(simpleReport.getFirstSignatureId());
```

12.5. Verifying the qualified status of timestamp

ETSI TS 119 615 ([R14]) specifies standardized procedures for the determination of the qualification of a timestamp. DSS is able to determine a qualification level of a timestamp if a relative information about TrustServiceProviders is provided to a certificate verifier (loaded automatically to a trusted certificate source with Configuration of TL validation job).

Three qualification levels are supported by DSS and can be obtained :

- `QTSA` (issued from a granted trust service with TSA/QTST type at the timestamp production time);
- `TSA` any other from a known trust anchor;
- `N/A` for others.

In order to determine a type and qualification of signature, an instance of `DetachedTimestampValidator` can be used for a detached CMS time-stamp verification, provided the relevant information extracted from a Trusted List(s).

For standalone time-stamps within different containers (e.g. PDF or ASiC) a corresponding instance of a `TimestampValidator` shall be used.

The following example verifies the qualification level of a timestamp:

Validate a timestamp and retrieve its qualification level

```
// Configure the internet access
CommonsDataLoader dataLoader = new CommonsDataLoader();

// We set an instance of TrustAllStrategy to rely on the Trusted Lists content
// instead of the JVM trust store.
dataLoader.setTrustStrategy(TrustAllStrategy.INSTANCE);

// Configure the TLValidationJob to load a qualification information from the
corresponding LOTL/TL
TLValidationJob tlValidationJob = new TLValidationJob();
tlValidationJob.setOnlineDataLoader(new FileCacheDataLoader(dataLoader));

// Configure the relevant TrustedList
TLSource tlSource = new TLSource();
tlSource.setUrl("http://dss-test.lu");
tlValidationJob.setTrustedListSources(tlSource);

// Initialize the trusted list certificate source to fill with the information
extracted from TLValidationJob
TrustedListsCertificateSource trustedListsCertificateSource = new
TrustedListsCertificateSource();
tlValidationJob.setTrustedListCertificateSource(trustedListsCertificateSource);

// Update TLValidationJob
tlValidationJob.onlineRefresh();

// No we need to configure the DocumentValidator
CertificateVerifier cv = new CommonCertificateVerifier();
cv.setTrustedCertSources(trustedListsCertificateSource); // configured trusted list
certificate source
cv.setAIASource(aiaSource); // configured AIA Access
cv.setOcspSource(ocspSource); // configured OCSP Access
cv.setCrlSource(crlSource); // configured CRL Access

// Create an instance of DetachedTimestampValidator
DocumentValidator validator = new DetachedTimestampValidator(timeStampDocument);
validator.setCertificateVerifier(cv);

// Validate the time-stamp
Reports reports = validator.validateDocument();
SimpleReport simpleReport = reports.getSimpleReport();

// Extract the qualification information
TimestampQualification timestampQualification =
simpleReport.getTimestampQualification(simpleReport.getFirstTimestampId());
```

13. Webservices

DSS offers REST and SOAP web services.

The different webservices are :

- **Signature webservices** (`dss-signature-soap` / `dss-signature-rest`) and their clients: expose methods to allow signing and augmenting or counter-signing a signature from a client.
- **Server-signing webservice** (`dss-server-signing-soap` / `dss-server-signing-rest`) and their clients: expose methods to retrieve keys from a server (PKCS#11, PKCS#12, HSM, etc.) and to sign the digest on the server side.
- **Signature validation webservices** (`dss-validation-soap` / `dss-validation-rest`) and their clients: expose methods to allow signature validation, with an optional detached file and an optional validation policy.
- **Certificate validation webservices** (`dss-certificate-validation-soap` / `dss-certificate-validation-rest`) and their clients: expose methods to allow certificate validation, with an optional provided certificate chain and custom validation time.
- **Timestamp webservices** (`dss-timestamp-remote-soap` / `dss-timestamp-remote-rest`) and their clients: expose methods to allow remote timestamp creation, by providing digest value to be timestamped and a digest algorithm, used for the digest calculation.

The data structure in webservices is similar in both REST and SOAP modules.

The documentation will cover the REST calls. All the REST services present in DSS are compliant with OpenAPI Specification.

Additionally, we also provide a SOAP-UI and Postman samples in the `dss-cookbook` module for simplicity.

13.1. REST

13.1.1. REST signature service

This service is composed by two modules:

- `dss-signature-service-client` - provides client interfaces for the REST webservices;
- `dss-signature-service` - contains REST webservices for signature creation, augmentation and document timestamping.

This service exposes several methods taking as input one or more document and having as output a signed data object (possibly a timestamped document):

Rest signature service

```
        // Initialize the rest client
        RestDocumentSignatureService restClient = new
RestDocumentSignatureServiceImpl();
```

```
        // Define RemoteSignatureParameters
        RemoteSignatureParameters parameters = new RemoteSignatureParameters();
        parameters.setSignatureLevel(SignatureLevel.PAdES_BASELINE_B);
        parameters.setSigningCertificate(new
RemoteCertificate(privateKey.getCertificate().getEncoded()));
        parameters.setSignaturePackaging(SignaturePackaging.ENVELOPING);
        parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

        // Initialize a RemoteDocument object to be signed
        FileDocument fileToSign = new FileDocument(new
File("src/test/resources/sample.pdf"));
        RemoteDocument toSignDocument = new
RemoteDocument(Utils.toByteArray(fileToSign.openStream()), fileToSign.getName());

        // Compute the digest to be signed
        ToBeSignedDTO dataToSign = restClient.getDataToSign(new
DataToSignOneDocumentDTO(toSignDocument, parameters));

        // Create a SignOneDocumentDTO
        SignatureValue signatureValue =
signingToken.sign(DTOConverter.toToBeSigned(dataToSign), DigestAlgorithm.SHA256,
privateKey);
        SignOneDocumentDTO signDocument = new SignOneDocumentDTO(toSignDocument,
parameters,
                new SignatureValueDTO(signatureValue.getAlgorithm(),
signatureValue.getValue()));

        // Add the signature value to the document
        RemoteDocument signedDocument = restClient.signDocument(signDocument);

        // Define the extension parameters
        RemoteSignatureParameters extendParameters = new RemoteSignatureParameters();
        extendParameters.setSignatureLevel(SignatureLevel.PAdES_BASELINE_T);

        // Extend the existing signature
        RemoteDocument extendedDocument = restClient.extendDocument(new
ExtendDocumentDTO(signedDocument, extendParameters));

        // Define timestamp parameters
        RemoteTimestampParameters remoteTimestampParameters = new
RemoteTimestampParameters();
        remoteTimestampParameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

        // Define a Timestamp document DTO
        TimestampOneDocumentDTO timestampOneDocumentDTO = new
TimestampOneDocumentDTO(extendedDocument, remoteTimestampParameters);

        // Timestamp a provided document (available for PDF, ASiC-E and ASiC-S
container formats)
        RemoteDocument timestampedDocument =
```

```
restClient.timestampDocument(timestampOneDocumentDTO);
```

13.1.1.1. Single document signing

A document signing assumes a signature creation in three (or four) consecutive steps (see Signature creation in DSS for more information). Two of the steps, namely "get data to sign" and "sign document" are available within the current module.

Below you can find examples for processing these steps when signing a single document.

13.1.1.1.1. Get data to sign

The method allows retrieving the data to be signed. The user sends the document to be signed, the parameters (signature level, etc.) and the certificate chain.

The parameters in getDataToSign and signDocument MUST be the same (especially the signing date).

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.1.1.2. Sign document

The method allows generation of the signed document with the received signature value.

The parameters in getDataToSign and signDocument MUST be the same (especially the signing date).

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.1.2. Multiple document signing

Similarly to Single document signing, the service exposes methods which allow signing of multiple documents with one signature (format dependent).

13.1.1.2.1. Get data to sign

The method allows retrieving the data to be signed. The user sends the documents to be signed, the parameters (signature level, etc.) and the certificate chain.

The parameters in getDataToSign and signDocument MUST be the same (especially the signing date).

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.1.2.2. Sign document

The method allows generation of the signed document with the received signature value.

The parameters in getDataToSign and signDocument MUST be the same (especially the signing date).

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.1.3. Extend document

The method allows augmentation of an existing signature to a higher level.

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.1.4. Timestamp document

The method allows timestamping of a provided document. Available for PDF, ASiC-E and ASiC-S container formats.

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request

- **Curl**: Request

13.1.1.5. Counter-signature

Similarly to Single document signing, the counter-signature creation requires execution of three (or four) consecutive steps, with the difference requiring a signed document to be provided as an input and Id of the signature to be counter-signed.

13.1.1.5.1. Get data to be counter-signed

This method returns the data to be signed in order to create a counter signature. The user should provide a document containing a signature to be counter-signed, id of the signature, and other parameters similarly to the method 'getDataToSign()'.

The parameters in getDataToBeCounterSigned and counterSignSignature MUST be the same (especially the signing date).

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.1.5.2. Counter-Sign Signature

This method incorporates a created counter signature to unsigned properties of the master signature with this specified id.

The parameters in getDataToBeCounterSigned and counterSignSignature MUST be the same (especially the signing date).

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.1.6. Trusted List signing

Special methods have been exposed (since DSS 5.10) allowing to sign a Trusted List (TL) or a List of Trusted Lists (LOTL) using a simplified interface with a pre-configured set of parameters.

The key difference with Single document signing methods is the use of the `RemoteTrustedListSignatureParameters` object, containing a limited set of important parameters for TL-signature creation, instead of `RemoteSignatureParameters` object, containing a wide set of various parameters.

13.1.1.6.1. Get data to sign

The method allows retrieving the data to be signed. The user sends the Trusted List to be signed and the parameters (signing certificate, signing date, etc.).

The parameters in getDataToSign and signDocument MUST be the same (especially the signing date).

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.1.6.2. Sign document

The method allows generation of the signed Trusted List with the received signature value.

The parameters in getDataToSign and signDocument MUST be the same (especially the signing date).

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.2. REST server signing service

This service also exposes some methods for server signing operations:

Rest server signing service

```
// Instantiate a RestSignatureTokenConnection
RestSignatureTokenConnection remoteToken = new RestSignatureTokenConnectionImpl();

// Retrieves available keys on server side
List<RemoteKeyEntry> keys = remoteToken.getKeys();

String alias = keys.get(0).getAlias();

// Retrieves a key on the server side by its alias
RemoteKeyEntry key = remoteToken.getKey(alias);

DSSDocument documentToSign = new InMemoryDocument("Hello world!".getBytes());

// Create a toBeSigned DTO
ToBeSignedDTO toBeSigned = new ToBeSignedDTO(DSSUtils.toByteArray(documentToSign));

// Signs the document with a given Digest Algorithm and alias for a key to use
// Signs the digest value with the given key
SignatureValueDTO signatureValue = remoteToken.sign(toBeSigned,
DigestAlgorithm.SHA256, alias);

// Or alternatively we can sign the document by providing digest only

// Prepare digestDTO.
// NOTE: the used Digest algorithm must be the same!
DigestDTO digestDTO = new DigestDTO(DigestAlgorithm.SHA256,
DSSUtils.digest(DigestAlgorithm.SHA256, documentToSign));

// Signs the digest
SignatureValueDTO signatureValueFromDigest = remoteToken.signDigest(digestDTO, alias);
```

13.1.2.1. Get keys

This method allows retrieving of all available keys on the server side (PKCS#11, PKCS#12, HSM, etc.). All keys will have an alias, a signing certificate and its chain. The alias will be used in following steps.

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.2.2. Get key

This method allows retrieving a key information for a given alias.

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.2.3. Sign

This method allows signing of given digests with a server side certificate.

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.3. REST validation service

DSS provides also a module for documents validation.

Rest validation service

```
// Initialize the rest client
RestDocumentValidationService validationService = new
RestDocumentValidationServiceImpl();

// Initialize document to be validated
FileDocument signatureToValidate = new FileDocument(new
File("src/test/resources/XAdESLTA.xml"));
RemoteDocument signedDocument =
RemoteDocumentConverter.toRemoteDocument(signatureToValidate);

// Initialize original document file to be provided as detached content (optional)
FileDocument detachedFile = new FileDocument("src/test/resources/sample.xml");
RemoteDocument originalDocument =
RemoteDocumentConverter.toRemoteDocument(detachedFile);

// Initialize XML validation policy to be used (optional, if not provided the default
policy will be used)
FileDocument policyFile = new FileDocument("src/test/resources/policy.xml");
RemoteDocument policy = RemoteDocumentConverter.toRemoteDocument(policyFile);

// Create the object containing data to be validated
DataToValidateDTO toValidate = new DataToValidateDTO(signedDocument, originalDocument,
policy);

// Validate the signature
WSReportsDTO result = validationService.validateSignature(toValidate);
```

13.1.3.1. Validate a document

This service allows a signature validation (all formats/types) against a validation policy.

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.3.2. Retrieve original document(s)

This service returns the signed data for a given signature.

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response

- **HTTPie**: Request
- **Curl**: Request

13.1.4. REST certificate validation service

The certificate validation service is used for validation of a certificate with the respective certificate chain.

Rest certificate validation service

```
// Instantiate a rest certificate validation service
RestCertificateValidationService validationService = new
RestCertificateValidationServiceImpl();

// Instantiate the certificate to be validated
CertificateToken certificateToken = DSSUtils.loadCertificate(new
File("src/test/resources/CZ.cer"));
RemoteCertificate remoteCertificate =
RemoteCertificateConverter.toRemoteCertificate(certificateToken);

// Instantiate certificate chain (optional, to be used when certificate chain cannot
be obtained by AIA)
CertificateToken caCertificate = DSSUtils.loadCertificate(new
File("src/test/resources/CA_CZ.cer"));
RemoteCertificate issuerRemoteCertificate =
RemoteCertificateConverter.toRemoteCertificate(caCertificate);

CertificateToken rootCertificate = DSSUtils.loadCertificate(new
File("src/test/resources/ROOT_CZ.cer"));
RemoteCertificate rootRemoteCertificate =
RemoteCertificateConverter.toRemoteCertificate(rootCertificate);

// Define validation time (optional, if not defined the current time will be used)
Calendar calendar = Calendar.getInstance();
calendar.set(2018, 12, 31);
Date validationDate = calendar.getTime();

// Create objects containing parameters to be provided to the validation process
CertificateToValidateDTO certificateToValidateDTO = new CertificateToValidateDTO(
        remoteCertificate, Arrays.asList(issuerRemoteCertificate,
rootRemoteCertificate), validationDate);

// Validate the certificate
CertificateReportsDTO reportsDTO =
validationService.validateCertificate(certificateToValidateDTO);
```

13.1.4.1. Validate a certificate

This service allows a certificate validation (provided in a binary format).

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.1.5. REST timestamp service

This service implements a timestamp creation.

Rest timestamp service

```
// Initialize the rest client
RestTimestampService timestampService = new RestTimestampServiceImpl();

// Initialize data to be timestamped (e.g. a document)
byte[] contentToBeTimestamped = "Hello World!".getBytes();

// Apply hash-function on the data
byte[] digestValue = DSSUtils.digest(DigestAlgorithm.SHA256, contentToBeTimestamped);

// Create an object to be provided to the timestamping service
// NOTE: ensure that the same DigestAlgorithm is used in both method calls
DigestDTO digest = new DigestDTO(DigestAlgorithm.SHA256, digestValue);

// Timestamp the digest
TimestampResponseDTO timestampResponse =
timestampService.getTimestampResponse(digest);
```

13.1.5.1. Get Timestamp Response

This service allows a remote timestamp creation. The method takes as an input the digest to be timestamped and digest algorithm that has been used for the digest value computation. The output of the method is the generated timestamp's binaries.

Samples:

- **JSON**: Request | Response
- **HTTP**: Request | Response
- **HTTPie**: Request
- **Curl**: Request

13.2. SOAP

The use of SOAP webServices is very similar to the REST implementation explained above. The main difference is the used implementation of the service. All methods, used parameters and

output objects are aligned between both REST and SOAP implementations.

13.2.1. SOAP signature service

SOAP signature service's client is initialized using `SoapDocumentSignatureService` class.

Soap signature service

```
// Initializes the SOAP client
SoapDocumentSignatureService soapClient = new SoapDocumentSignatureServiceImpl();
```

The use of the client is similar to REST signature service.

13.2.2. SOAP server signing service

SOAP server signing service's client is initialized using `SoapSignatureTokenConnection` class.

Soap server signing service

```
// Instantiate a SoapSignatureTokenConnection
SoapSignatureTokenConnection remoteToken = new SoapSignatureTokenConnectionImpl();
```

The use of the client is similar to REST server signing service.

13.2.3. SOAP validation service

SOAP validation service's client is initialized using `SoapDocumentValidationService` class.

Soap validation service

```
// Initialize the soap client
SoapDocumentValidationService validationService = new
SoapDocumentValidationServiceImpl();
```

The use of the client is similar to REST validation service.

13.2.4. SOAP certificate validation service

SOAP validation service's client is initialized using `SoapCertificateValidationService` class.

Soap certificate validation service

```
// Instantiate a soap certificate validation service
SoapCertificateValidationService validationService = new
SoapCertificateValidationServiceImpl();
```

The use of the client is similar to REST certificate validation service.

13.2.5. SOAP timestamp service

SOAP validation service's client is initialized using `SoapTimestampService` class.

Soap timestamp service

```
// Initialize the soap client
SoapTimestampService timestampService = new SoapTimestampServiceImpl();
```

The use of the client is similar to REST timestamp service.

14. Internationalization (i18n)

14.1. Language of reports

DSS provides a module allowing changing a language for generated reports.

Internationalization module supports translation of only validation process messages.

A target language of the report can be set with the following code:

Language customization

```
SignedDocumentValidator validator =
SignedDocumentValidator.fromDocument(signedDocument);
// A target Locale must be defined for the validator
validator.setLocale(Locale.FRENCH); // for French language
```

In case if no language is specified, the framework will use a default `Locale` obtained from OS on a running machine. If a requested language is not found, a default translation will be used.

As a default configuration DSS provides English translation.

In order to provide a custom translation, a new file must be created inside `src\main\resources` directory of your project with a name followed by one of the patterns:

`dss-messages_XX.properties` or `dss-messages_XX_YY.properties`, where:

- XX - an abbreviation of a target language;
- YY - a country code.

For example, for a French language a file with a name `dss-messages_fr.properties` should be created, or `dss-messages_fr_FR.properties` to use it only in France local.

15. Exceptions

This section provides an overview of runtime Exceptions which are being thrown by various modules of DSS framework.

The following Exceptions can be obtained by the upper level:

- `NullPointerException` is thrown when a mandatory parameter has not been provided by the end-user to the method/process, requiring the property;
- `IllegalArgumentException` is thrown when a configuration of input parameters is not valid for the called method or some parameters cannot be used together (e.g. on a signature creation);
- `IllegalInputException` is thrown when a provided input document is not valid for the requested process and/or the configuration of parameters is not applicable for the given document;
- `UnsupportedOperationException` is thrown when a method is not implemented or its usage with the requested parameters is not (yet) supported;
- `IllegalStateException` is thrown when the requested method cannot be performed at the current method (e.g. another method shall be executed before);
- `DSSException` is thrown in case of an error obtained during the internal DSS process (e.g. data conversion, CRL/OCSP parsing, etc.);
- `DSSExternalResourceException` is thrown if an error occurs during a remote source request (AIA, CRL, OCSP requests, etc.);
- `DSSRemoteServiceException` is thrown in case of a request/response error within REST and SOAP;
- `SecurityConfigurationException` is thrown in case of invalid configuration of security features (see XML securities).

16. Privacy

16.1. Use of digested documents

Digested documents can be used during signature creation instead of the original documents to keep the latter private. In that case, the signature is created in a detached way, using the digest of the original document only.

Refer to section Representation of documents in DSS for more information on digested documents, section Signature packaging for information on the detached packaging of signatures and section Detached signature based on digested document for code illustrations.

16.2. Original document in the Data To Be Signed

The data to be signed (DTBS) for CAdES, XAdES and PAdES does not contain the original document on which the signature value is computed. JAdES DTBS however, when not the DTBS of a detached JAdES signature making use of the ObjectIdByURIHash referencing mechanism, does contain the whole original content.

16.3. Private information in logs

Five log levels are used in DSS:

- **ERROR** - is used for the most critical issues, interrupting a normal process workflow (encoding issues, not processable signed attributes, etc.).
- **WARN** - is used to indicate problems occurred during an executed process. Such issue does not stop or invalidate the process explicitly, but can have an impact on the final output.
- **INFO** - returns important or useful information to the logs.
- **DEBUG** - is used to print extended information, such as token binaries, attribute values, etc.
- **TRACE** - is used to indicate the currently performing methods and state of objects.

When setting a log execution level to `ERROR`, `WARN` or `INFO` levels, no private information will be print to the log file. However, `DEBUG` and `TRACE` might potentially contain private information, as they display more information, including the source binaries of tokens.

Since the logs are hardcoded it is not possible to modify the behavior. Therefore, to avoid disclosing private information, users should not use the `DEBUG` and `TRACE` levels. It is also not recommended using these two levels in the production.

For example, when an error occurs on a certificate reading, DSS only `WARN` users that an error occurred. However, if the log level is set to `DEBUG` on the user's side, the binaries of the failed certificate are printed.

Sometimes, DSS uses mixing rules for logging, such as it displays more information within a `WARN` or `INFO` level when `DEBUG` level is enabled.

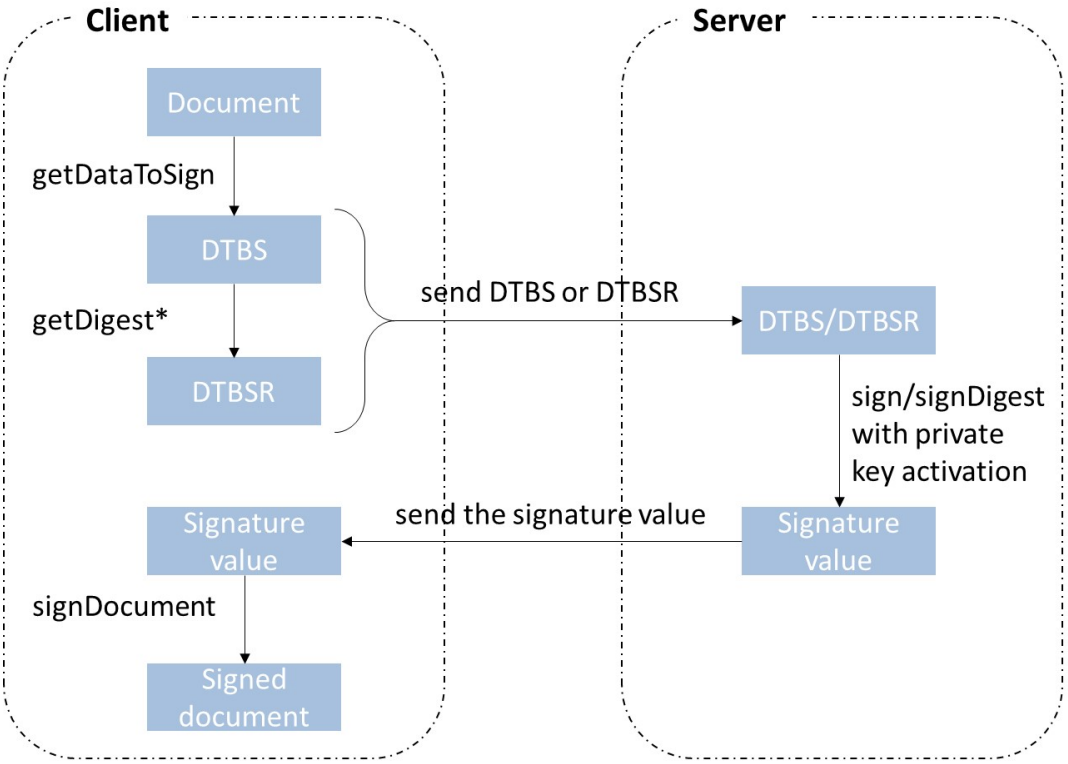
16.4. Client-side signature creation with server-side remote key activation

With DSS, it is possible to sign a document without needing to send it to the signing server. This is useful for users who do not want signing servers to have access to the information contained in their documents. Such a process is possible because DSS decomposes the signature of a document in three or four atomic steps. See section Signature creation in 3 stateless methods and Signature creation in 4 stateless methods for an extensive description of these steps.

1. The first step is performed by the client and consists in the computation of the data to be signed (DTBS).
2. For the XAdES, CAdES and PAdES formats, the client can optionally compute the digest of the DTBS (DTBSR). For JAdES the client should compute the digest of the DTBS to "hide" the content of the original document, given that the DTBS of this format (usually) contains the whole original content. The client sends the DTBS or the DTBSR to the server.
3. Then, the server computes the signature value by encrypting the DTBSR (or hashing the DTBS and encrypting the resulting DTBSR in one go) using the private key. The server sends the signature value back to the client.

4. The last step takes place at the client-side. The client adds the signature value to the appropriate field.

The following schema illustrates the different steps



For code illustrations of the different steps, refer to the Client-side signature creation with server-side remote key activation section in the Annex.

17. Advanced DSS java concepts

17.1. ServiceLoader

DSS incorporates modules that are loaded in the run time based on the chosen configuration and the input data via a ServiceLoader. This provides a flexibility for an end-user to work only with selected modules and a possibility to expand DSS with custom implementations.

In order to provide a chosen implementation(s) to `ServiceLoader`, a file listing all the desired implementations should be created in the resource directory `META-INF/services` with a name matching the implemented interface. When merging sources (e.g. creating a Fat JAR module), the files can be lost/overwritten, and should be configured manually (all the required implementations shall be listed).

It is also possible to customize the order of the used implementation, but creating a corresponding file in `META-INF/services` directory within your project.

If a DSS module(s) implementing a required interface(s) is added to your project's dependency list, the implementation shall be loaded automatically.

The following modules are provided with independent implementations:

- DSS Utils;
- DSS CRL Parser;
- DSS PAdES.

At least one implementation shall be chosen.

17.1.1. DSS Utils

The module `dss-utils` offers an interface with utility methods to operate on String, Collection, I/O, etc. DSS framework provides two different implementations with the same behaviour :

- `dss-utils-apache-commons`: this module uses Apache Commons libraries (commons-lang3, commons-collection4, commons-io and commons-codec);
- `dss-utils-google-guava`: this module uses Google Guava (recommended on Android).

If your integration includes `dss-utils`, you will need to select an implementation. For example, to choose the `dss-utils-apache-commons` implementation within a Maven project you need to define the following:

pom.xml

```
<dependency>
    <groupId>eu.europa.ec.joinup.sd-dss</groupId>
    <artifactId>dss-utils-apache-commons</artifactId>
</dependency>
```

17.1.2. DSS CRL Parser

DSS contains two ways to parse/validate a CRL and to retrieve revocation data. An alternative to the `X509CRL` java object was developed to face memory issues in case of large CRLs. The `X509CRL` object fully loads the CRL in memory and can cause `OutOfMemoryError`.

- `dss-crl-parser-x509crl`: this module uses the `X509CRL` java object.
- `dss-crl-parser-streams`: this module offers an alternative with a CRL streaming.

If your integration requires `dss-crl-parser`, you will need to choose your implementation. For example, to choose the `dss-crl-parser-streams` implementation within a Maven project you need to define the following:

pom.xml

```
<dependency>
    <groupId>eu.europa.ec.joinup.sd-dss</groupId>
    <artifactId>dss-crl-parser-stream</artifactId>
</dependency>
```

17.1.3. DSS PAdES

DSS allows generation, augmentation and validation of PAdES signatures with two different frameworks: PDFBox and OpenPDF (fork of iText). The `dss-pades` module only contains the common code and requires an underlying implementation :

- `dss-pades-pdfbox`: PAdES implementation based on Apache PDFBox. Supports drawing of custom text, images, as well as text+image, in a signature field.
- `dss-pades-openpdf`: PAdES implementation based on OpenPDF (fork of iText). Supports drawing of custom text OR images in a signature field.

DSS permits to override the visible signature generation with these interfaces:

- `eu.europa.esig.dss.pdf.IPdfObjFactory`;
- `eu.europa.esig.dss.pdf.visible.SignatureDrawerFactory` (selects the `SignatureDrawer` depending on the `SignatureImageParameters` content);
- `eu.europa.esig.dss.pdf.visible.SignatureDrawer`.

A new instance of the `IPdfObjFactory` can be created with its own `SignatureDrawerFactory` and injected in the `padesService.setPdfObjFactory(IPdfObjFactory)`. By default, DSS uses an instance of `ServiceLoaderPdfObjFactory`. This instance checks for any registered implementation in the classpath with the ServiceLoader (potentially a service from `dss-pades-pdfbox`, `dss-pades-openpdf` or your own(s)).

17.1.3.1. DSS PDFBox

DSS allows switching between two implementations of the PDFBox framework: default (original) and native.

- **Native Drawer**: A native implementation of PDFBox Drawer, allowing a user to add a vector text, image or combination of text and image to a visible signature field. The native implementation embeds the provided custom text to the inner PDF structure, that makes the text selectable and searchable, but also clearer and smoother in comparison with the raster implementation.
- **Default Drawer**: The original drawer implemented on the PDFBox framework, supports displaying of custom text, images, but also text and image combination in a signature field. The implementation does not include the provided custom text to the inner PDF structure, instead of it, the drawer creates an image representation of the provided text, which is added to the signature field (i.e. the text is not selectable and not searchable).

By default, DSS uses the "Native Drawer" as the PDFBox implementation. In order to switch the

implementation, that allowed at runtime, you have to set a new instance for PdfObjFactory as following:

Runtime PDF Object Factory changing

```
service.setPdfObjFactory(new PdfBoxNativeObjectFactory());
```

Or create a new file `META-INF/services/eu.europa.esig.dss.pdf.IPdfObjFactory` within project's resources, defining the desired implementation (see ServiceLoader for more information).

17.1.3.2. DSS OpenPDF

This implementation is based on the OpenPDF framework. DSS provides two drawers using the implementation:

- `TextOnlySignatureDrawer` - to draw a vector text information within a signature field. The text information is selectable and searchable.
- `ImageOnlySignatureDrawer` - to draw an image within a signature field.

DSS provides a limited support of OpenPDF framework, therefore not all features are supported (e.g. text+image drawing).

17.2. Multithreading

DSS can be used in multi-threaded environments but some points need to be considered like resources sharing and caching. All operations are stateless and this fact requires to be maintained. Some resources can be shared, others are proper to an operation.

For each provided operation, DSS requires a `CertificateVerifier` object. This object is responsible to provide certificates and accesses to external resources (AIA, CRL, OCSP, etc.). At the beginning of all operation, CertificateSources and RevocationSources are created for each signature / timestamp / revocation data. Extracted information are combined with the configured sources in the CertificateVerifier. For these reasons, integrators need to be careful about the `CertificateVerifier` configuration.

17.2.1. Resource sharing

The trusted certificates can be shared between multiple threads because these certificates are static. This means they don't require more analysis. Their status won't evolve. For these certificates, DSS doesn't need to collect issuer certificate and/or their revocation data.

In opposition, the adjunct certificates cannot be shared. These certificates concern a specific signature/validation operation. This parameter is used to provide missing certificate(s). When DSS is unable to build the complete certificate path with the provided certificates (as signature parameters or embedded within a signature), it is possible to inject certificates that are not present. These certificates are not necessarily trusted and may require future "modifications" like revocation data collection, etc.

17.2.2. Caching

In case of multi-threading usage, we strongly recommend caching of external resources. All external resources can be cached (AIA, CRL, OCSP) to improve performances and to avoid requesting too much time the same resources. FileCacheDataLoader and JdbcCacheCRLSource can help you in this way.

See section Caching use cases of the Annex for complete examples of caching revocation data, certificates and trusted lists.

17.3. JAXB modules

17.3.1. General

DSS provides the following JAXB modules with a harmonized structure :

- `dss-policy-jaxb` - defines validation policy JAXB model;
- `dss-diagnostic-jaxb` - defines Diagnostic Data JAXB model;
- `dss-detailed-report-jaxb` - defines Detailed Report JAXB model;
- `dss-simple-report-jaxb` - defines Simple Report JAXB model;
- `dss-simple-certificate-report-jaxb` - defines Certificate Simple Report JAXB model.

All modules share the same logic and have the following structure (where *** is a model name):

`dss-***-jaxb`

`/src/main/java`

`eu.europa.esig.dss.***`

- `***.java` - wrapper(s) which eases the JAXB manipulation
- `…`
- `***Facade.java` - class which allows marshalling/unmarshalling of jaxb objects, generation of HTML/PDF content, etc.
- `***XmlDefiner.java` - class which contains the model definition (XSD, XSLT references, ObjectFactory)
- `/jaxb` - generated on compile time
  - `Xml***.java` - JAXB model
  - `…`

`/src/main/resources`

`/xsd`

- `***.xsd` - XML Schema (XSD) for the Detailed Report model
- `binding.xml` - XJC instructions to generate the JAXB model from the XSD

/xslt

- /html
  - ***.xslt** - XML Stylesheet for the HTML generation
- /pdf
  - ***.xslt** - XML Stylesheet for the PDF generation

In the main classes, a Facade is present to quickly operate with the JAXB objects (e.g. marshall, unmarshall, generate the HTML/PDF, validate the XML structure, etc.).

DetailedReportFacade usage

```
Reports completeReports = documentValidator.validateDocument();

DetailedReportFacade detailedReportFacade = DetailedReportFacade.newFacade();

// Transforms the JAXB object to String (xml content)
String marshalledDetailedReport =
detailedReportFacade.marshall(completeReports.getDetailedReportJaxb());

// Transforms the String (xml content) to a JAXB Object
XmlDetailedReport xmlDetailedReport =
detailedReportFacade.unmarshall(marshalledDetailedReport);

// Generates the HTML content for the given Detailed Report (compatible with
// BootStrap)
// Similar method is available for PDF generation (requires Apache FOP)
String htmlDetailedReport =
detailedReportFacade.generateHtmlReport(completeReports.getDetailedReportJaxb());
```

An XmlDefiner is also available with the access to the embedded XML Schemas (XSD), the XML Stylesheets (XSLT) to be able to generate the HTML or the PDF content (for DSS specific JAXB) and the JAXB Object Factory.

DetailedReportXmlDefiner usage

```
// The JAXB Object Factory
ObjectFactory objectFactory = DetailedReportXmlDefiner.OBJECT_FACTORY;

// The JAXBContext (cached)
JAXBContext jaxbContext = DetailedReportXmlDefiner.getJAXBContext();

// The XML Schema to validate a XML content (cached)
Schema schema = DetailedReportXmlDefiner.getSchema();

// The Templates object with the loaded XML Stylesheet to generate the HTML
// content from the JAXB Object (cached)
Templates bootstrap4Templates = DetailedReportXmlDefiner.getHtmlBootstrap4Templates();

// The Templates object with the loaded XML Stylesheet to generate the PDF
// content from the JAXB Object (cached)
Templates pdfTemplates = DetailedReportXmlDefiner.getPdfTemplates();
```

17.3.2. Creating a trusted list

It is possible to programmatically create or/and edit an XML Trusted List using the JAXB module.

Below is an example of how to use JAXB modules to create a trusted list (not complete solution):

Creation of a trusted list

```
// Create an empty 'TrustServiceStatusList' element
TrustStatusListType trustStatusListType = objectFactory.createTrustStatusListType();

// Fill the requred information, Id, ...
trustStatusListType.setId("Demo-TL");

// Store to file
DSSDocument modifiedUnsignedTL = null;
try (final ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
    TrustedListFacade.newFacade().marshall(trustStatusListType, baos, false);
    modifiedUnsignedTL = new InMemoryDocument(baos.toByteArray());
}
modifiedUnsignedTL.save("target/unsigned_TL.xml");
```

And an example how to modify an existing Trusted List (e.g. change its version):

Edit a trusted list

```
// Load original Trusted List
final File file = new File("src/main/resources/trusted-list.xml");

// Parse it to JAXB object
final TrustStatusListType jaxbObject = TrustedListFacade.newFacade().unmarshall(file,
false);
assertNotNull(jaxbObject);

// Modify JAXB Object where required
jaxbObject.getSchemeInformation().setTSLSequenceNumber(new BigInteger("39"));

// Store to file
DSSDocument modifiedUnsignedTL;
try (final ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
    TrustedListFacade.newFacade().marshall(jaxbObject, baos, false);
    modifiedUnsignedTL = new InMemoryDocument(baos.toByteArray());
}
modifiedUnsignedTL.save("target/unsigned_TL.xml");
```

17.3.3. Validating XSD conformity

You can also use JAXB modules not only for the content creation or changing, but also in order to verify the conformity of an XML document against its XSD schema.

For example, in order to validate a XAdES signature conformance against 1.3.2 XSD schema, you can use the corresponding class `XAdES319132Utils`:

Validating XSD conformity

```
Document signatureDocDom = DomUtils.buildDOM(xadesSignatureDocument);
List<String> errors = XAdES319132Utils.getInstance().validateAgainstXSD(new
DOMSource(signatureDocDom));
```

17.3.4. Report stylesheets

The report modules (namely: `dss-simple-report-jaxb`, `dss-simple-certificate-report-jaxb` and `dss-detailed-report-jaxb`) contain XSLT style sheets each for final reports generation:

- Bootstrap 4 XSLT for HTML report;
- PDF XSLT for PDF report.

Since DSS 5.9 only Bootstrap 4 XSLT is provided within the framework for HTML report generation.

In order to generate a report with a selected style sheet you need to call a relevant method in a Facade class (see classes definition above):

HTML report generation

```
String bootstrap4Report =
SimpleReportFacade.newFacade().generateHtmlReport(xmlSimpleReport);
```

Otherwise, in case you need to customize the transformer, you can create a report by using an `XmlDefiner`:

Custom report generation

```
try (Writer writer = new StringWriter()) {
    Transformer transformer =
SimpleCertificateReportXmlDefiner.getHtmlBootstrap4Templates().newTransformer();
    // specify custom parameters if needed
    transformer.transform(new StreamSource(new StringReader(simpleReport)), new
StreamResult(writer));
    String bootstrap4Report = writer.toString();
}
```

17.3.5. Diagnostic data stylesheets

The `dss-diagnostic-jaxb` module contains an XSLT stylesheet that creates an SVG image from an XML document in order to visually represent the signature at validation time.

SVG generation

```
// Initialize DiagnosticData to create an SVG image from
File diagnosticDataXmlFile = new File("src/test/resources/diag-data.xml");

// Initialize the DiagnosticData facade in order to unmarshall the XML Diagnostic Data
DiagnosticDataFacade newFacade = DiagnosticDataFacade.newFacade();

// Unmarshall the DiagnosticData
XmlDiagnosticData diagnosticData = newFacade.unmarshall(diagnosticDataXmlFile);

// Generate and store the SVG image
try (FileOutputStream fos = new FileOutputStream("target/diag-data.svg")) {
    Result result = new StreamResult(fos);
    newFacade.generateSVG(diagnosticData, result);
}
```

17.4. XML securities

The framework allows custom configuration of XML-related modules for enabling/disabling of XML securities (e.g. in order to use Xalan or Xerces).

We strongly do not recommend disabling of security features and usage of deprecated dependencies. Be aware: the feature is designed only for experienced users, and all changes made in the module are at your own risk.

The configuration is available for the following classes:

- `javax.xml.parsers.DocumentBuilderFactory` with a `DocumentBuilderFactoryBuilder` - builds a DOM document object from the obtained XML file and creates a new `org.w3c.dom.Document`;
- `javax.xml.transform.TransformerFactory` with a `TransformerFactoryBuilder` - loads XML templates and builds DOM objects;
- `javax.xml.validation.SchemaFactory` with a `SchemaFactoryBuilder` - loads XML Schema;
- `javax.xml.validation.Validator` with a `ValidatorConfigurator` - configures a validator to validate an XML document against an XML Schema.

All the classes can be configured with the following methods (example for `TransformerFactory`):

XMLSecurities configuration

```
// Obtain a singleton instance of {@link XmlDefinerUtils}
XmlDefinerUtils xmlDefinerUtils = XmlDefinerUtils.getInstance();

// returns a predefined {@link TransformerFactoryBuilder} with all securities in place
TransformerFactoryBuilder transformerBuilder =
TransformerFactoryBuilder.getSecureTransformerBuilder();

// sets an alert in case of exception on feature/attribute setting
transformerBuilder.setSecurityExceptionAlert(new LogOnStatusAlert(Level.WARN));

// allows to enable a feature
transformerBuilder.enableFeature(XMLConstants.FEATURE_SECURE_PROCESSING);

// allows to disable a feature
transformerBuilder.disableFeature("FEATURE_TO_DISABLE");

// allows to set an attribute with a value
transformerBuilder.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");

// sets the transformer (will be applied for all calls)
xmlDefinerUtils.setTransformerFactoryBuilder(transformerBuilder);
```

The `javax.xml.parsers.DocumentBuilderFactory`, that allows XML files parsing and creation of DOM `org.w3c.dom.Document` object, can be configured with the following methods:

Since DSS 5.9 the configuration of `javax.xml.parsers.DocumentBuilderFactory` has been moved from `DomUtils` to a new singleton class `DocumentBuilderFactoryBuilder`.

DocumentBuilderFactory configuration

```
// returns a configured secure instance of {@link DocumentBuilderFactoryBuilder}
DocumentBuilderFactoryBuilder documentBuilderFactoryBuilder =
DocumentBuilderFactoryBuilder.getSecureDocumentBuilderFactoryBuilder();

// allows enabling of a feature
documentBuilderFactoryBuilder.enableFeature("http://xml.org/sax/features/external-
general-entities");

// allows disabling of a feature
documentBuilderFactoryBuilder.disableFeature("http://apache.org/xml/features/nonvalida
ting/load-external-dtd");

// allows to set an attribute
documentBuilderFactoryBuilder.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");

// sets the DocumentBuilderFactoryBuilder (will be applied for all calls)
xmlDefinerUtils.setDocumentBuilderFactoryBuilder(documentBuilderFactoryBuilder);
```

The class `XmlDefinerUtils` is a singleton, therefore all changes performed on the instance will have an impact to all calls of the related methods.

See section Use of Alerts throughout the framework in chapter Annex for more information on alerts.

18. DSS upgrades and change history

18.1. Release notes

Table 10. Release notes

Version	Release date	Features
v5.10.RC1	February 2022	Main new features / improvements : • Cookbook update; • PAdES : object modification detection; • PAdES : visual signature preview; • PAdES : avoid repeated creation of OCSP/CRL tokens; • PAdES : enforce signature creation/validation against ISO 32 000 restrictions (DocMDP, Lock, etc.); • PAdES : add validation data on timestamp method (including data for standalone timestamps); • XAdES and CAdES : added support of extended profiles on validation; • ASiC services refactoring (various improvements); • WebService to sign a Trusted List; • Apple KeyStore as a signature token connection; • ED448 signature algorithm support; • Revocation check on B/T-level signature creation; • Added supportive information to Status object in alerts; • Same instance of signature parameters can be used for multiple signing operation; • Demo : new viewer for XML reports (i.e. for DiagnosticData and ETSI VR); • Dependencies upgrade (HttpClient5, BouncyCastle, Santuario, logback, etc.); • Java 17 support.

Version	Release date	Features
v5.10.RC1	February 2022	Bug fixes : • PAdES : erroneously triggered visual signature difference warning; • PAdES : wrong LT-/LTA-level determination for documents with multiple signatures; • PAdES : original documents extraction does not work against carriage return; • XAdES : NPE on validation of XAdES v.1.1.1, 1.2.2; • CAdES : NPE on signature validation without signing-certificate; • CAdES : counter-signature produces duplicates of existing counter-signatures; • JAdES : wrong payload computation for 'sigD' with ObjectIdByURI mechanism; • ASiC : MimeType is lost on re-signature; • Signature policy caching issue; • Revocation freshness checks use different values across the code; • Demo : jumping rows on collapse of TL-validation table; • Demo : inability to sign when encryption algorithm of the token is different from the one used in signature; • Demo : wrong encoding on uploaded filenames containing non-ASCII characters.

Version	Release date	Features
v5.9	September 2021	Main new features / improvements : • Many improvements in the validation reports; • AIASource introduction : more customizations; • Customization of revocation collection strategy (OCSP/CRL first); • DocumentBuilderFactory securities; • ECDSA / ECDSA-PLAIN support; • JAdES (JSON AdES) consolidations; • PAdES visual signature refactorings / improvements : ◦ Image scaling : STRETCH / ZOOM_AND_CENTER / CENTER; ◦ Text wrapping : BOX_FILL / FILL_BOX_AND_LINEBREAK / FONT_BASIC. • Dependency upgrades (Santuario, BouncyCastle, PDFBox,…); • Java 16 support. Bug fixes : • Short term OCSP response; • On hold certificate; • Qualification conflict (issuance time / best signing time); • ASiC-S can’t be timestamped twice; • PAdES revision extraction; • PAdES wrong level detection (files with multiple signatures/timestamps); • ETSI Validation report : multiple files / references.
v5.8	February 2021	• JAdES implementation (ETSI TS 119 182 v0.0.6) : signature creation, extension and validation (advanced electronic signatures based on JWS); • PDF Shadow attacks : prevention and detection; • Counter Signature creation (CAdES, XAdES, JAdES and ASiC containers); • Support of the unsigned attribute SignaturePolicyStore (CAdES, XAdES, JAdES and ASiC containers); • Support of the QCLimitValue attribute; • Support of Java 8 up to 15.

Version	Release date	Features
v5.7	August 2020	• CertificatePool removal and performance amelioration; • QWAC validator; • New design of PDF reports; • Support of PSD2 attributes; • Support of EdDSA; • Signature representation with a timeline; • Visual signature creation with REST/SOAP webservices.
v5.6	March 2020	• Complete rewriting of the TL/LOTL loading with: ◦ online / offline refresh; ◦ 3 caches (download / parse / validate); ◦ multiple LOTL support; ◦ multiple TL support (not linked to a LOTL); ◦ Pivot LOTL support; ◦ Synchronization strategy (eg : expired TL/LOTL are rejected/accepted); ◦ multi-lingual support (trust service matching); ◦ alerting (eg : LOTL/OJ location desynchronization,…); ◦ complete reporting (summary of download / parsing / validation). • Independent timestamp creation and validation (not linked to a signature, with ASiC and PDF); • Timestamp qualification; • Internationalization of the validation reports; • Multiple Trusted Sources support; • XAdES support of different prefixes / versions.
v5.5	October 2019	• The implementation of the ETSI Validation Report; • The support of Java 12 (multi-release jars); • Webservice which allows to validate certificates.
v5.4.3	August 2019	• Hotfix release.

Version	Release date	Features
v5.4	January 2019	• Augmentation of signatures with invalid time-stamps, archive-time-stamps and revoked certificates; • Upgrade to Java 8 or 9; • Certify documents; • Add support of KeyHash in OCSP Responses.
v5.3.2	October 2018	• Security patch, following a security assessment from the Ruhr-Universität Bochum.
v5.3.1	July 2018	• Certificate validation; • content-timestamps generation; • SHA-3 support; • non-EU trusted list(s) support; • integration of the last version of MOCCA.
v5.3	May 2018	• Certificate validation; • content-timestamps generation; • SHA-3 support; • non-EU trusted list(s) support; • integration of the last version of MOCCA.
v5.2.1	October 2018	• Security patch, following a security assessment from the Ruhr-Universität Bochum.
v5.2	December 2017	• Qualification matrix guidelines and documentation; • Improvements regarding visual representation of a signature; • Alternative packaging: Image docker / spring-boot; • CRL streaming, the demo won't use the X509CRL java object by default (it can be changed). With some signatures, we had large CRLs (+60Mo in Estonia) and that could cause memory issues. • RSASSA-PSS support, I received some requests to support these algorithms : ◦ SHA1withRSAandMGF1; ◦ SHA224withRSAandMGF1; ◦ SHA256withRSAandMGF1; ◦ SHA384withRSAandMGF1; ◦ SHA512withRSAandMGF1.

Version	Release date	Features
v5.1	September 2017	• Webservices for Server signing REST and SOAP; • PAdES : Support of signature fields; • PAdES : distinction of PAdES and PKCS7 signatures; • Proxy configuration fix.
v5.0	April 2017	• Refactoring of ASiC format handling, following the ETSI ASiC Plugtest; • Signature of multiple files (ASiC and XAdES); • Integration of the Qualification matrix as described in draft ETSI 119 172-4, for supporting signatures before and after 01/07/2016 (eIDAS entry into force); • Migration to PDFBox 2 for handling PDFs. • Complete refactoring of the ASiC part (creation, extension and validation); • Compliance to eIDAS regulation.
v4.7	October 2016	A XAdES PlugTest is planned in October / November 2015. Remaining changes resulting from this PlugTest and not included in v4.6 may be included in this release. An eSignature Validation PlugTest is planned in April 2016. Depending on the actual timeframe, impacts from this PlugTest may be included in this release, and the release of DSS 4.7 will be postponed accordingly. Other potential improvements and features: • Extension of signature validation policy support; • CAdES attribute certificates; • CRL in multiple parts; • Distributed timestamps method; • Support of cross-certification in path building.

Version	Release date	Features
v4.6*	March 2016	Based on standards: • Signature formats when creating a signature: baseline profiles ETSI TS 103 171, 103 172, 103 173, and 103 174; • Signature formats when validating a signature: baseline profiles, and core specs ETSI TS 101 903, 101 733, 102 778 and 102 918; • Signature validation process ETSI TS 102 853. Improvements in packaging and core functionalities: • CAdES optimisation, CAdES multiple Signer Information. A CAdES PlugTest is occurring in June and July 2015. Changes resulting from this PlugTest will be included in this release. CAdES countersignature will not be supported. • Impacts from XAdES PlugTest of October 2015. • Processing of large files. • Further refactoring of demo applet (size, validation policy editor). • SOAP and REST Web Services. • Standalone demo application.

** October 2015: Implementing Acts Art. 27 & 37 (eSig formats)*

18.2. Version upgrade

To upgrade version of DSS, locate to the `pom.xml` file of your project, search for the properties and then change the dss version in the corresponding field(s).

The following example shows how to switch to DSS version 5.10 using Integration with Bill of Materials (BOM) module.

pom.xml

```
<properties>
    ...
    <dss.version>5.10</dss.version>
    ...
</properties>

...

 <dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>eu.europa.ec.joinup.sd-dss</groupId>
            <artifactId>dss-bom</artifactId>
            <version>${dss.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

18.3. Migration guide

This chapter covers the most significant changes in DSS code occurred between different versions, requiring review and possible changes from code implementors.

For changes within XML Signature Policy please refer Validation policy migration guide.

Table 11. Code changes from version 5.9 to 5.10

Title	v5.9	v5.10
ASiC container extraction	`ASiCExtractResult extractedResult = asicContainerExtractor.extract();`	`ASiCContent extractedResult = asicContainerExtractor.extract();`
HttpClient5 transition	`import org.apache.http.*`	`import org.apache.hc.client5.http.*` `import org.apache.hc.core5.http.*`

FileCacheDataLoader	`fileCacheDataLoader.setCacheExpirationTime(Long.MAX_VALUE);`	`fileCacheDataLoader.setCacheExpirationTime(-1); // negative value means cache never expires`
DiagnosticData: PDF signature field name	`List<String> fieldNames = xmlPDFRevision.getSignatureFieldName();` `String name = fieldNames.get(i);`	`List<PDFSignatureField> signatureFields = xmlPDFRevision.getPDFSignatureField();` `String name = signatureFields.get(i).getName();`

Table 12. Code changes from version 5.8 to 5.9

Title	v5.8	v5.9
AIA data loader	`certificateVerifier.setDataLoader(dataLoader);`	`AIASource aiaSource = new DefaultAIASource(dataLoader);` `certificateVerifier.setAIASource(aiaSource);`

Signature Policy Provider	`certificateVerifier.setDataLoader(dataLoader);`	`SignaturePolicyProvider signaturePolicyProvider = new SignaturePolicyProvider();`
		`signaturePolicyProvider.setDataLoader(dataLoader);`
		`documentValidator.setSignaturePolicyProvider(signaturePolicyProvider);`

JDBC dataSource	`JdbcRevocationSource.setDataSource(dataSource);`	`JdbcCacheConnector jdbcCacheConnector = new JdbcCacheConnector(dataSource);` `jdbcRevocationSource.setJdbcCacheConnector(jdbcCacheConnector);`

DiagnosticData: Signature policy	`String notice = xmlPolicy.getNotice(); Boolean zeroHash = xmlPolicy.isZeroHash(); XmlDigestAlgoAndValue digestAlgoAndValue = xmlPolicy.getDigestAlgoAndValue(); Boolean status = xmlPolicy.isStatus(); Boolean digestAlgorithsEqual = xmlPolicy.isDigestAlgorithsEqual();`	`XmlUserNotice notice = xmlPolicy.getUserNotice(); Boolean zeroHash = xmlPolicy.getDigestAlgoAndValue().isZeroHash(); XmlPolicyDigestAlgoAndValue digestAlgoAndValue = xmlPolicy.getDigestAlgoAndValue(); Boolean status = xmlPolicy.getDigestAlgoAndValue().isMatch(); Boolean digestAlgorithsEqual = xmlPolicy.getDigestAlgoAndValue().isDigestAlgorithsEqual();`
DiagnosticData: QCStatements	`XmlPSD2Info psd2Info = xmlCertificate.getPSD2Info(); List<XmlOID> qcStatementIds = xmlCertificate.getQCStatementIds(); List<XmlOID> qcTypes = xmlCertificate.getQCTypes(); QCLimitValue qcLimitValue = xmlCertificate.getQCLimitValue(); OID semanticsIdentifier = xmlCertificate.getSemanticsIdentifier();`	`XmlPSD2Info psd2Info = xmlCertificate.getQcStatements().getPSD2Info(); QcCompliance qcCompliance = xmlCertificate.getQcStatements().getQcCompliance(); BigInteger qcEuRetentionPeriod = xmlCertificate.getQcStatements().getQcEuRetentionPeriod(); QcEuPDS qcEuPDS = xmlCertificate.getQcStatements().getQcEuPDS(); List<XmlOID> qcTypes = xmlCertificate.getQcStatements().getQCTypes(); QcEuLimitValue qcLimitValue = xmlCertificate.getQcStatements().getQcEuLimitValue(); QCLimitValue qcLimitValue = xmlCertificate.getQcStatements().getQCLimitValue(); OID semanticsIdentifier = xmlCertificate.getQcStatements().getSemanticsIdentifier();`

18.4. Validation policy migration guide

This chapter covers the changes occurred between different versions of DSS within AdES validation constraints/policy.

Table 13. Policy changes from version 5.9 to 5.10

Title	v5.9	v5.10

Revocation freshness (time constraint enforced)	`<CertificateConstraints>`
	`    ...`
	`    <RevocationDataFreshness Level="FAIL" />`
	`    ...`
	`</CertificateConstraints>`
	
	`...`
	
	`<RevocationConstraints>`
	`    ...`
	`    <RevocationFreshness Level="FAIL" Unit="DAYS" Value="0" />`
	`    ...`
	`</RevocationConstraints>`
		`<CertificateConstraints>`
		`    ...`
		`    <RevocationFreshness Level="FAIL" Unit="DAYS" Value="0" />`
		`    ...`
		`</CertificateConstraints>`

Revocation freshness (no time constraint)	`<CertificateConstraints>`
	`    ...`
	`    <RevocationDataFreshness Level="FAIL" />`
	`    ...`
	`</CertificateConstraints>`
	
	`...`
	
	`<RevocationConstraints>`
	`    ...`
	`    <!--<RevocationFreshness />-->`
	`    ...`
	`</RevocationConstraints>`
		`<CertificateConstraints>`
		`    ...`
		
		`<RevocationFreshnessNextUpdate Level="FAIL" />`
		`    ...`
		`</CertificateConstraints>`

Signing-certificate reference certificate chain	<CertificateConstraints> ... <SemanticsIdentifierForNaturalPerson /> <SemanticsIdentifierForLegalPerson /> ... </CertificateConstraints>	<CertificateConstraints> ... <SemanticsIdentifier> <Id>0.4.0.194121.1.1</Id> // for natural person <Id>0.4.0.194121.1.2</Id> // for legal person </SemanticsIdentifier> ... </CertificateConstraints>

Table 14. Policy changes from version 5.8 to 5.9

Title	v5.8	v5.9
Revocation nextUpdate check	<CertificateConstraints> ... <RevocationDataNextUpdatePresent /> ... </CertificateConstraints>	<CertificateConstraints> ... <CRLNextUpdatePresent /> <OCSPNextUpdatePresent /> ... </CertificateConstraints>
Signing-certificate reference certificate chain	<SignedAttributesConstraints> ... <AllCertDigestsMatch /> ... </SignedAttributesConstraints>	<SignedAttributesConstraints> ... <SigningCertificateRefersCertificateChain /> ... </SignedAttributesConstraints>

18.5. Frequently asked questions and implementation issues

This chapter covers the most frequently asked questions and issues occurring in implementations using DSS.

Table 15. Possible problems and solutions

Version	Description	Solution

v5.9	Returned signature level is *_NOT_ETSI	**DSS 5.9** enforces validation of AdES BASELINE signature profiles as per ETSI standards. **The signature is not BASELINE** if you receive **_NOT_ETSI* output. **Since DSS 5.10 a support of AdES extended signature profiles** for (XAdES and CAdES) has been added as per ETSI standards.
v5.8 and higher	PAdES : performance downgrade	`pdfSignatureService.setMaximalPagesAmountForVisualComparison(0); // skip validation` For a complete code please see the example.

v5.7 and higher	How to filter certain Trusted Lists (for example countries)	To filter LOTLs or TLs you can use predicates. For example, to filter TLs from Germany and Romania:
		
		`// the predicates filter only TSL pointers with scheme territories "DE" (Germany) and "RO" (Romania)`
		`// to XML documents with "http://uri.etsi.org/TrstSvc/TrustedList/TSLType/EUgeneric" type`
		`lotlSource.setTlPredicate(new SchemeTerritoryOtherTSLPointer(Arrays.asList("DE","RO"))`
		`.and(new EULOTLOtherTSLPointer()).and(new XMLOtherTSLPointer()));`

v5.2 and higher	XAdES : performance downgrade	Xalan dependency has been removed as deprecated.
		We do not recommend to use Xalan in production. To use Xalan, you will need to remove security attributes:
		
		`XmlDefinerUtils.getInstance().setTransformerFactoryBuilder(`
		`TransformerFactoryBuilder.getSecureTransformerBuilder()`
		`.removeAttribute(XMLConstants.ACCESS_EXTERNAL_DTD)`
		`.removeAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET));`

all versions	Build fails when using `quick` profile	• DSS 5.9 and lower: Build the following modules using `mvn clean install` (without any profile): ◦ `dss-utils`; ◦ `dss-crl-parser`; ◦ `dss-test`; ◦ `dss-pades`; ◦ `dss-asic-common` *(since DSS 5.8)*. • DSS 5.10: Use `quick-init` profile for the first build of DSS.

<table>
<tr>
<td>all versions</td>
<td>Revocation data is missing on LT-level extension</td>
<td>Verify whether the trust anchors and CRL/OCSP sources are configured appropriately. You need to provide them to the used CertificateVerifier:

CertificateVerifier configuration
<pre>
// Create common certificate verifier
CommonCertificateVerifier
commonCertificateVerifier = new
CommonCertificateVerifier();

// Provide trust anchors
commonCertificateVerifier.setTrustedCertSources
(tslCertificateSource);

// Instantiate CRL source
OnlineCRLSource onlineCRLSource = new
OnlineCRLSource();
onlineCRLSource.setDataLoader(commonsHttpDataLo
ader);
commonCertificateVerifier.setCrlSource(onlineCR
LSource);

// Instantiate OCSP source
OnlineOCSPSource onlineOCSPSource = new
OnlineOCSPSource();
onlineOCSPSource.setDataLoader(ocspDataLoader);
commonCertificateVerifier.setOcspSource(onlineO
CSPSource);

// For test purpose (not recommended for use in
production)
// Will request unknown OCSP responder /
download untrusted CRL
commonCertificateVerifier.setCheckRevocationFor
UntrustedChains(true);

// Create XAdES service for signature
XAdESService service = new
XAdESService(commonCertificateVerifier);
</pre></td>
</tr>
<tr>
<td>all versions</td>
<td>Unable to access DSS Maven repository</td>
<td>If the error occurs, more likely DSS-team is already aware about the issue.
You need to try to connect again in a few hours.</td>
</tr>
</table>

all versions	PAdES validation without sending the document	PAdES format **requires a complete document** to perform a signature validation. You cannot validate a PDF signature as a DETACHED XAdES or CAdES. The only possible way to perform the **cryptographic signature validation** is to extract the embedded CMS signature and the covered range as a detached document. But the validation result will not be able to conclude it as a valid PAdES nor CAdES.
all versions	When validating a PDF with embedded CAdES-BASELINE signature the returned format is PDF_NOT_ETSI	PAdES-BASELINE format establishes some limitations on the embedded CMS signature, which are not compliant with CAdES-BASELINE signatures. Therefore, **it is not possible** to have a valid PAdES-BASELINE profile with embedded CAdES-BASELINE signature. For more information about supported CMS please see [R03].

19. Annex

19.1. Use of Alerts throughout the framework

The framework includes an extended possibility to execute custom processes in case of arbitrary defined events.

The `Alert` is a basic interface used to trigger a process on a passed object. DSS provides an `AbstractAlert` implementation of the interface with a clearly defined structure. The class must be instantiated with two attributes:

- `AlertDetector` - used to detect an event/state of the object and trigger a process;
- `AlertHandler` - defines a process to be executed on an object.

In its basic module, framework provides a few alerts based on a `Status`:

- `ExceptionOnStatusAlert` - throws an `AlertException` (RuntimeException) when the status reports an issue;
- `LogOnStatusAlert` - logs a message with the defined log level;
- `SilentOnStatusAlert` - ignores the reported issue and does nothing.

The usage of alerts is available in the following classes:

- XML securities configurators from `dss-jaxb-parsers` module : `TransformerFactoryBuilder`, `SchemaFactoryBuilder`, `ValidatorConfigurator` (see chapter XML securities for more information);
- CertificateVerifier - to handle the unexpected situation(s) in a custom way (introduced `AlertException` to re-throw exceptions, see section CertificateVerifier configuration for more information);
- TLValidationJob - to process custom actions on change/state on loading of LOTL/TLs (see

`LOTLAlert` and `TLAlert` in the Alerting from TL Loading section).

19.2. Configuration of validation policy in different use cases

19.2.1. General

A Signature validation policy is a set of rules/constraints that need to be fulfilled to validate a signature. When checking a constraints fails, this leads to a sub-indication in the validation report.

Table 16. Validation policy constraints

Block	Constraint	Type	Indication	SubIndication
ContainerConstraints	AcceptableContainerTypes	MultiValuesConstraint	FAILED	FORMAT_FAILURE
	ZipCommentPresent	LevelConstraint	FAILED	FORMAT_FAILURE
	AcceptableZipComment	MultiValuesConstraint	FAILED	FORMAT_FAILURE
	MimeTypeFilePresent	LevelConstraint	FAILED	FORMAT_FAILURE
	AcceptableMimeTypeFileContent	MultiValuesConstraint	FAILED	FORMAT_FAILURE
	ManifestFilePresent	LevelConstraint	FAILED	FORMAT_FAILURE
	SignedFilesPresent	LevelConstraint	FAILED	FORMAT_FAILURE
	AllFilesSigned	LevelConstraint	FAILED	FORMAT_FAILURE

Block	Constraint	Type	Indication	SubIndication
SignatureConstraints	StructuralValidation	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	AcceptablePolicies	MultiValuesConstraint	INDETERMINATE	POLICY_PROCESSING_ERROR
	PolicyAvailable	LevelConstraint	INDETERMINATE	SIGNATURE_POLICY_NOT_AVAILABLE
	SignaturePolicyStorePresent	LevelConstraint	INDETERMINATE	SIGNATURE_POLICY_NOT_AVAILABLE
	PolicyHashMatch	LevelConstraint	INDETERMINATE	SIGNATURE_POLICY_NOT_AVAILABLE
	AcceptableFormats	MultiValuesConstraint	FAILED	FORMAT_FAILURE
	FullScope	LevelConstraint	FAILED	FORMAT_FAILURE
	BasicSignatureConstraints	BasicSignatureConstraints	See BasicSignatureConstraints	
	SignedAttributes	SignedAttributesConstraints	See SignedAttributesConstraints	
	UnsignedAttributes	UnsignedAttributesConstraints	See UnsignedAttributesConstraints	

Block	Constraint	Type	Indication	SubIndication
BasicSignatureConstraints	ReferenceDataExistence	LevelConstraint	INDETERMINATE	SIGNED_DATA_NOT_FOUND
	ReferenceDataIntact	LevelConstraint	FAILED	HASH_FAILURE
	ManifestEntryObjectExistence	LevelConstraint	INDETERMINATE	SIGNED_DATA_NOT_FOUND
	SignatureIntact	LevelConstraint	FAILED	SIG_CRYPTO_FAILURE
	SignatureDuplicated	LevelConstraint	FAILED	FORMAT_FAILURE
	ProspectiveCertificateChain	LevelConstraint	INDETERMINATE	NO_CERTIFICATE_CHAIN_FOUND
	SignerInformationStore	LevelConstraint	FAILED	FORMAT_FAILURE
	PdfPageDifference	LevelConstraint	FAILED	FORMAT_FAILURE
	PdfAnnotationOverlap	LevelConstraint	FAILED	FORMAT_FAILURE
	PdfVisualDifference	LevelConstraint	FAILED	FORMAT_FAILURE
	DocMDP	LevelConstraint	FAILED	FORMAT_FAILURE
	FieldMDP	LevelConstraint	FAILED	FORMAT_FAILURE
	SigFieldLock	LevelConstraint	FAILED	FORMAT_FAILURE
	UndefinedChanges	LevelConstraint	FAILED	FORMAT_FAILURE
	TrustedServiceTypeIdentifier	MultiValuesConstraint	INDETERMINATE	NO_CERTIFICATE_CHAIN_FOUND
	TrustedServiceStatus	MultiValuesConstraint	INDETERMINATE	NO_CERTIFICATE_CHAIN_FOUND
	SigningCertificate	CertificateConstraints	See CertificateConstraints	
	CACertificate	CertificateConstraints	See CertificateConstraints	
	Cryptographic	CryptographicConstraint	See CryptographicConstraint	

Block	Constraint	Type	Indication	SubIndication
CertificateConstraints	Recognition	LevelConstraint	INDETERMINATE	NO_SIGNING_CERTIFICATE_FOUND
	Signature	LevelConstraint	INDETERMINATE	CERTIFICATE_CHAIN_GENERAL_FAILURE
	NotExpired	LevelConstraint	INDETERMINATE	OUT_OF_BOUNDS_NO_POE OUT_OF_BOUNDS_NOT_REVOKED
			FAILED	EXPIRED
	AuthorityInfoAccessPresent	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	RevocationInfoAccessPresent	LevelConstraint	INDETERMINATE	CERTIFICATE_CHAIN_GENERAL_FAILURE
	RevocationDataAvailable	LevelConstraint	INDETERMINATE	TRY_LATER
	AcceptableRevocationDataFound	LevelConstraint	INDETERMINATE	TRY_LATER
	CRLNextUpdatePresent	LevelConstraint	INDETERMINATE	TRY_LATER
	OCSPNextUpdatePresent	LevelConstraint	INDETERMINATE	TRY_LATER
	RevocationFreshness	TimeConstraint	INDETERMINATE	TRY_LATER
	RevocationFreshnessNextUpdate	LevelConstraint	INDETERMINATE	TRY_LATER

Block	Constraint	Type	Indication	SubIndication
CertificateConstraints	KeyUsage	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	ExtendedKeyUsage	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	Surname	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	GivenName	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	CommonName	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	Pseudonym	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	OrganizationUnit	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	OrganizationName	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	Country	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	SerialNumberPresent	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	NotRevoked	LevelConstraint	INDETERMINATE	REVOKED_NO_POE REVOKED_CA_NO_POE
			FAILED	REVOKED

Block	Constraint	Type	Indication	SubIndication
CertificateConstraints	NotOnHold	LevelConstraint	INDETERMINATE	TRY_LATER
	RevocationIssuerNotExpired	LevelConstraint	INDETERMINATE	REVOCATION_OUT_OF_BOUNDS_NO_POE
	SelfSigned	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	NotSelfSigned	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	PolicyIds	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	PolicyQualificationIds	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	PolicySupportedByQSCDIds	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	QcCompliance	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	QcEuLimitValueCurrency	ValueConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	MinQcEuLimitValue	IntValueConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	MinQcEuRetentionPeriod	IntValueConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	QcSSCD	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE

Block	Constraint	Type	Indication	SubIndication
CertificateConstraints	QcEuPDSLocation	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	QcType	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	QcLegislationCountryCodes	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	IssuedToNaturalPerson	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	IssuedToLegalPerson	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	SemanticsIdentifier	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	PSD2QcTypeRolesOfPSP	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	PSD2QcCompetentAuthorityName	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	PSD2QcCompetentAuthorityId	MultiValuesConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	UsePseudonym	LevelConstraint	INDETERMINATE	CHAIN_CONSTRAINTS_FAILURE
	Cryptographic	CryptographicConstraint	See CryptographicConstraint	

Block	Constraint	Type	Indication	SubIndication
SignedAttributesConstraints	SigningCertificatePresent	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	UnicitySigningCertificate	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	SigningCertificateRefersCertificateChain	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	ReferencesToAllCertificateChainPresent	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	SigningCertificateDigestAlgorithm	LevelConstraint	INDETERMINATE	CRYPTO_CONSTRAINTS_FAILURE_NO_POE
	CertDigestPresent	LevelConstraint	INDETERMINATE	NO_SIGNING_CERTIFICATE_FOUND
	CertDigestMatch	LevelConstraint	INDETERMINATE	NO_SIGNING_CERTIFICATE_FOUND
	IssuerSerialMatch	LevelConstraint	INDETERMINATE	NO_SIGNING_CERTIFICATE_FOUND
	KeyIdentifierPresent	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	KeyIdentifierMatch	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE

Block	Constraint	Type	Indication	SubIndication
SignedAttributesConstraints	SigningTime	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	ContentType	ValueConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	ContentHints	ValueConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	ContentIdentifier	ValueConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	MessageDigestOrSignedPropertiesPresent	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	CommitmentTypeIndication	MultiValuesConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	SignerLocation	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	ClaimedRoles	MultiValuesConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	CertifiedRoles	MultiValuesConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	ContentTimeStamp	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	ContentTimeStampMessageImprint	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
UnsignedAttributesConstraints	CounterSignature	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE

Block	Constraint	Type	Indication	SubIndication
TimestampConstraints	TimestampDelay	TimeConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	RevocationTimeAgainstBestSignatureTime	LevelConstraint	INDETERMINATE	REVOKED_NO_POE REVOKED_CA_NO_POE
	BestSignatureTimeBeforeExpirationDateOfSigningCertificate	LevelConstraint	FAILED	NOT_YET_VALID
	Coherence	LevelConstraint	INDETERMINATE	TIMESTAMP_ORDER_FAILURE
	BasicSignatureConstraints	BasicSignatureConstraints	See BasicSignatureConstraints	
	SignedAttributes	SignedAttributesConstraints	See SignedAttributesConstraints	
	TSAGeneralNamePresent	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	TSAGeneralNameContentMatch	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
	TSAGeneralNameOrderMatch	LevelConstraint	INDETERMINATE	SIG_CONSTRAINTS_FAILURE
RevocationConstraints	UnknownStatus	LevelConstraint	INDETERMINATE	TRY_LATER
	OCSPCertHashPresent	LevelConstraint	INDETERMINATE	TRY_LATER
	OCSPCertHashMatch	LevelConstraint	INDETERMINATE	TRY_LATER
	SelfIssuedOCSP	LevelConstraint	INDETERMINATE	TRY_LATER
	BasicSignatureConstraints	BasicSignatureConstraints	See BasicSignatureConstraints	

Block	Constraint	Type	Indication	SubIndication
Cryptographic	AcceptableEncryptionAlgo	ListAlgo	INDETERMINATE	CRYPTO_CONSTRAINTS_FAILURE_NO_POE CRYPTO_CONSTRAINTS_FAILURE
	MiniPublicKeySize	ListAlgo	INDETERMINATE	CRYPTO_CONSTRAINTS_FAILURE_NO_POE CRYPTO_CONSTRAINTS_FAILURE
	AcceptableDigestAlgo	ListAlgo	INDETERMINATE	CRYPTO_CONSTRAINTS_FAILURE_NO_POE CRYPTO_CONSTRAINTS_FAILURE
	AlgoExpirationDate	AlgoExpirationDate	INDETERMINATE	CRYPTO_CONSTRAINTS_FAILURE_NO_POE CRYPTO_CONSTRAINTS_FAILURE
eIDAS	TLFreshness	TimeConstraint	FAILED	-
	TLNotExpired	LevelConstraint	FAILED	-
	TLWellSigned	LevelConstraint	FAILED	-
	TLVersion	ValueConstraint	FAILED	-

19.2.2. AdES validation

According to ETSI EN 319 102-1 (cf. [R09]), the signature validation process can be separated to different levels:

- **Validation process for basic signatures** - validates the signature at the validation (current) time;
- **Validation process for Signatures with Time and Signatures with Long-Term Validation Material** - verifies the signature against its *best-signature-time* (i.e. against the signature time-stamp's production time);
- **Validation process for Signatures providing Long Term Availability and Integrity of Validation Material** - verifies the signature with all available Long-Term Availability material (i.e. including the validation of archive time-stamps).

DSS allows the user to choose the validation level when performing a signature validation, i.e. to specify the validation process to be used for validation (cf. [R09]). By default, the highest level (with LTA enabled) is used.

19.2.2.1. Basic AdES validation

Below you can find a signature validation example with a basic signature validation level:

B-level AdES validation

```
// The document to be validated (any kind of signature file)
DSSDocument document = new FileDocument(new File("src/test/resources/signature-
pool/signedXmlXadesLT.xml"));

// First, we need a Certificate verifier
CertificateVerifier cv = new CommonCertificateVerifier();
cv.setAIASource(new DefaultAIASource());
cv.setOcspSource(new OnlineOCSPSource());
cv.setCrlSource(new OnlineCRLSource());

cv.addTrustedCertSources(trustedCertSource);
cv.addAdjunctCertSources(adjunctCertSource);

// We create an instance of DocumentValidator
// It will automatically select the supported validator from the classpath
SignedDocumentValidator documentValidator =
SignedDocumentValidator.fromDocument(document);

// We add the certificate verifier
documentValidator.setCertificateVerifier(cv);

// Validate the signature only against its B-level
documentValidator.setValidationLevel(ValidationLevel.BASIC_SIGNATURES);

// Here, everything is ready. We can execute the validation (for the example, we use
the default and embedded
// validation policy)
Reports reports = documentValidator.validateDocument();
```

19.2.2.2. Long Term AdES validation

LTV-level AdES validation

```
// Validate the signature with long-term validation material
documentValidator.setValidationLevel(ValidationLevel.LONG_TERM_DATA);
```

19.2.2.3. Long Term Availability AdES validation

LTA-level AdES validation

```
// Validate the signature with long-term availability and integrity material
documentValidator.setValidationLevel(ValidationLevel.ARCHIVAL_DATA);
```

19.2.3. Trusted list validation

A validation of a Trusted List is similar to a signature validation, with the only difference that the validation of a Trusted List can be done in offline mode.

Additionally, a validation against the XSD schema should be performed.

Validation of a trusted list

```
// Create an instance of a trusted certificate source
// NOTE: signing-certificate of a TL shall be trusted directly
CommonTrustedCertificateSource trustedCertSource = new
CommonTrustedCertificateSource();
trustedCertSource.addCertificate(getSigningCert());

// First, we need a Certificate verifier (online sources are not required for TL-
validation)
CertificateVerifier cv = new CommonCertificateVerifier();
cv.addTrustedCertSources(trustedCertSource);

// We create an instance of XMLDocumentValidator
DocumentValidator documentValidator = new XMLDocumentValidator(signedTrustedList);

// We add the certificate verifier
documentValidator.setCertificateVerifier(cv);

// TL shall be valid at the validation time
documentValidator.setValidationLevel(ValidationLevel.BASIC_SIGNATURES);

// Here, everything is ready. We can execute the validation.
Reports reports = documentValidator.validateDocument();

// Additionally, the TL can be validated against the XSD schema
Document tlDocDom = DomUtils.buildDOM(signedTrustedList);
List<String> errors = TrustedListUtils.getInstance().validateAgainstXSD(new
DOMSource(tlDocDom));
```

19.3. Caching use cases

19.3.1. Caching revocation data (CRL, OCSP)

19.3.1.1. CRL

An example for JdbcCacheCRLSource:

JdbcCacheCRLSource usage

```
// Creates an instance of JdbcCacheCRLSource
JdbcCacheCRLSource cacheCRLSource = new JdbcCacheCRLSource();

// Initialize the JdbcCacheConnector
JdbcCacheConnector jdbcCacheConnector = new JdbcCacheConnector(dataSource);

// Set the JdbcCacheConnector
cacheCRLSource.setJdbcCacheConnector(jdbcCacheConnector);

// Allows definition of an alternative dataLoader to be used to access a revocation
// from online sources if a requested revocation is not present in the repository or
has been expired (see below).
cacheCRLSource.setProxySource(onlineCRLSource);

// All setters accept values in seconds
Long oneWeek = (long) (60 * 60 * 24 * 7); // seconds * minutes * hours * days

// If "nextUpdate" field is not defined for a revocation token, the value of
"defaultNextUpdateDelay"
// will be used in order to determine when a new revocation data should be requested.
// If the current time is not beyond the "thisUpdate" time + "defaultNextUpdateDelay",

// then a revocation data will be retrieved from the repository source, otherwise a
new revocation data
// will be requested from a proxiedSource.
// Default : null (a new revocation data will be requested of "nestUpdate" field is
not defined).
cacheCRLSource.setDefaultNextUpdateDelay(oneWeek);

// Defines a custom maximum possible nextUpdate delay. Allows limiting of a time
interval
// from "thisUpdate" to "nextUpdate" defined in a revocation data.
// Default : null (not specified, the "nextUpdate" value provided in a revocation is
used).
cacheCRLSource.setMaxNextUpdateDelay(oneWeek); // force refresh every week (eg : ARL)

// Defines if a revocation should be removed on its expiration.
// Default : true (removes revocation from a repository if expired).
cacheCRLSource.setRemoveExpired(true);

// Creates an SQL table
cacheCRLSource.initTable();

// Extract CRL for a certificate
CRLToken crlRevocationToken = cacheCRLSource.getRevocationToken(certificateToken,
issuerCertificateToken);
```

19.3.1.2. OCSP

An example for JdbcCacheOCSPSource:

JdbcCacheOCSPSource usage

```
// Creates an instance of JdbcCacheOCSPSource
JdbcCacheOCSPSource cacheOCSPSource = new JdbcCacheOCSPSource();

// Initialize the JdbcCacheConnector
JdbcCacheConnector jdbcCacheConnector = new JdbcCacheConnector(dataSource);

// Set the JdbcCacheConnector
cacheOCSPSource.setJdbcCacheConnector(jdbcCacheConnector);

// Allows definition of an alternative dataLoader to be used to access a revocation
// from online sources if a requested revocation is not present in the repository or
has been expired (see below).
cacheOCSPSource.setProxySource(onlineOCSPSource);

// All setters accept values in seconds
Long threeMinutes = (long) (60 * 3); // seconds * minutes

// If "nextUpdate" field is not defined for a revocation token, the value of
"defaultNextUpdateDelay"
// will be used in order to determine when a new revocation data should be requested.
// If the current time is not beyond the "thisUpdate" time + "defaultNextUpdateDelay",
// then a revocation data will be retrieved from the repository source, otherwise a
new revocation data
// will be requested from a proxiedSource.
// Default : null (a new revocation data will be requested of "nestUpdate" field is
not defined).
cacheOCSPSource.setDefaultNextUpdateDelay(threeMinutes);

// Creates an SQL table
cacheOCSPSource.initTable();

// Extract OCSP for a certificate
OCSPToken ocspRevocationToken = cacheOCSPSource.getRevocationToken(certificateToken,
issuerCertificateToken);
```

19.3.2. Caching certificates (AIA certificates)

An example for JdbcCacheOCSPSource:

Caching of certificates

```
// Creates an instance of JdbcCacheAIASource
JdbcCacheAIASource cacheAIASource = new JdbcCacheAIASource();

// Initialize the JdbcCacheConnector
JdbcCacheConnector jdbcCacheConnector = new JdbcCacheConnector(dataSource);

// Set the JdbcCacheConnector
cacheAIASource.setJdbcCacheConnector(jdbcCacheConnector);

// Allows definition of an alternative dataLoader to be used to access a revocation
// from online sources if a requested revocation is not present in the repository or
has been expired (see below).


// Creates an SQL table
cacheAIASource.initTable();

// Extract certificates by AIA
Set<CertificateToken> aiaCertificates =
cacheAIASource.getCertificatesByAIA(certificateToken);
```

19.3.3. Caching trusted lists

Trusted Lists and List(s) of Trusted Lists are cached automatically as a part if `TLValidationJob` (see Configuration of TL validation job). To configure it you may use `FileCacheDataLoader` (see DSSFileLoader).

To load Trusted Lists from a cache, the offline loader shall be configured, and the action can be performed with the method:

Trusted Lists update from a cache

```
// call with the Offline Loader (application initialization)
validationJob.offlineRefresh();
```

19.4. Complete examples of Signature creation

19.4.1. XAdES

Below is an example of the `XAdES-Baseline-B` signature signing an XML document:

Create a XAdES-BASELINE-B signature

```
// 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);
```

19.4.2. CAdES

Below is an example of the `CAdES-Baseline-B` signature:

Signing a file with CAdES

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

19.4.3. PAdES

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

Signing a PDF file with PAdES

```
// 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 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);

// Optionally or for debug purpose :
// Validate the signature value against the original dataToSign
assertTrue(service.isValidSignatureValue(dataToSign, signatureValue,
privateKey.getCertificate()));

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

19.4.3.1. PAdES Visible Signature

DSS provides a large set of utilities for PDF visible signature creation (see PAdES Visible Signature for more information).

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

Add a visible signature to a PDF document

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

// Initialize visual signature and configure
SignatureImageParameters imageParameters = new SignatureImageParameters();
// set an image
imageParameters.setImage(new
InMemoryDocument(getClass().getResourceAsStream("/signature-pen.png")));

// initialize signature field parameters
SignatureFieldParameters fieldParameters = new SignatureFieldParameters();
imageParameters.setFieldParameters(fieldParameters);
// the origin is the left and top corner of the page
fieldParameters.setOriginX(200);
fieldParameters.setOriginY(400);
fieldParameters.setWidth(300);
fieldParameters.setHeight(200);
parameters.setImageParameters(imageParameters);

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

Additionally, DSS also allows you to insert a visible signature to an existing field :

Add a visible signature to an existing field

```
SignatureFieldParameters fieldParameters = new SignatureFieldParameters();
fieldParameters.setFieldId("field-id");
```

The following sections present examples of existing parameters for creation of visible signatures with DSS.

19.4.3.1.1. Positioning

DSS provides a set of functions allowing to place the signature field on a specific place in the PDF page :

Visible signature positioning

```
// Object containing a list of visible signature parameters
SignatureImageParameters signatureImageParameters = new SignatureImageParameters();

// Allows alignment of a signature field horizontally to a page. Allows the following
values:
/* _NONE_ (_DEFAULT value._ None alignment is applied, coordinates are counted from
the left page side);
   _LEFT_ (the signature is aligned to the left side, coordinated are counted from the
left page side);
   _CENTER_ (the signature is aligned to the center of the page, coordinates are
counted automatically);
   _RIGHT_ (the signature is aligned to the right side, coordinated are counted from
the right page side). */
signatureImageParameters.setAlignmentHorizontal(VisualSignatureAlignmentHorizontal.CEN
TER);

// Allows alignment of a signature field vertically to a page. Allows the following
values:
/* _NONE_ (_DEFAULT value._ None alignment is applied, coordinated are counted from
the top side of a page);
   _TOP_ (the signature is aligned to a top side, coordinated are counted from the top
page side);
   _MIDDLE_ (the signature aligned to a middle of a page, coordinated are counted
automatically);
   _BOTTOM_ (the signature is aligned to a bottom side, coordinated are counted from
the bottom page side). */
signatureImageParameters.setAlignmentVertical(VisualSignatureAlignmentVertical.TOP);

// Rotates the signature field and changes the coordinates' origin respectively to its
values as following:
/* _NONE_ (_DEFAULT value._ No rotation is applied. The origin of coordinates begins
from the top left corner of a page);
   _AUTOMATIC_ (Rotates a signature field respectively to the page's rotation. Rotates
the signature field on the same value as defined in a PDF page);
   _ROTATE_90_ (Rotates a signature field for a 90&#176; clockwise. Coordinates'
origin begins from top right page corner);
```

```
   _ROTATE_180_ (Rotates a signature field for a 180&#176; clockwise. Coordinates'
origin begins from the bottom right page corner);
   _ROTATE_270_ (Rotates a signature field for a 270&#176; clockwise. Coordinates'
origin begins from the bottom left page corner). */
signatureImageParameters.setRotation(VisualSignatureRotation.AUTOMATIC);

// Defines a zoom of the image. The value is applied to width and height of a
signature field.
// The value must be defined in percentage (default value is 100, no zoom is applied).
signatureImageParameters.setZoom(50);

// Specifies a background color for a signature field.
signatureImageParameters.setBackgroundColor(Color.GREEN);

// Defines the image scaling behavior within a signature field with a fixed size
/*
  STRETCH - the default behavior, stretches the image in both directions in order to
fill the signature field box;
  ZOOM_AND_CENTER - zooms the image to fill the signature box to the closest side, and
centers in another dimension;
  CENTER - centers the image in both dimensions.
*/
signatureImageParameters.setImageScaling(ImageScaling.CENTER);

// set the image parameters to signature parameters
padesSignatureParameters.setImageParameters(signatureImageParameters);
```

19.4.3.1.2. Dimensions

DSS framework provides a set of functions to manage the signature field size :

Visible signature dimensions

```
// Object containing a list of signature field parameters
SignatureFieldParameters fieldParameters = new SignatureFieldParameters();
signatureImageParameters.setFieldParameters(fieldParameters);

// Allows defining of a specific page in a PDF document where the signature must be
placed.
// The counting of pages starts from 1 (the first page)
// (the default value = 1).
fieldParameters.setPage(1);

// Absolute positioning functions, allowing to specify a margin between
// the left page side and the top page side respectively, and
// a signature field (if no rotation and alignment is applied).
fieldParameters.setOriginX(10);
fieldParameters.setOriginY(10);

// Allows specifying of a precise signature field's width in pixels.
// If not defined, the default image/text width will be used.
fieldParameters.setWidth(100);

// Allows specifying of a precise signature field's height in pixels.
// If not defined, the default image/text height will be used.
fieldParameters.setHeight(125);
```

19.4.3.1.3. Text Parameters

The available implementations allow placing of a visible text to a signature field :

List of available visible text parameters

```
// Instantiates a SignatureImageTextParameters object
SignatureImageTextParameters textParameters = new SignatureImageTextParameters();
// Allows you to set a DSSFont object that defines the text style (see more
information in the section "Fonts usage")
textParameters.setFont(font);
// Defines the text content
textParameters.setText("My visual signature \n #1");
// Defines the color of the characters
textParameters.setTextColor(Color.BLUE);
// Defines the background color for the area filled out by the text
textParameters.setBackgroundColor(Color.YELLOW);
// Defines a padding between the text and a border of its bounding area
textParameters.setPadding(20);
// TextWrapping parameter allows defining the text wrapping behavior within  the
signature field
/*
  FONT_BASED - the default text wrapping, the text is computed based on the given font
size;
  FILL_BOX - finds optimal font size to wrap the text to a signature field box;
  FILL_BOX_AND_LINEBREAK - breaks the words to multiple lines in order to find the
biggest possible font size to wrap the text into a signature field box.
*/
textParameters.setTextWrapping(TextWrapping.FONT_BASED);
// Set textParameters to a SignatureImageParameters object
imageParameters.setTextParameters(textParameters);
```

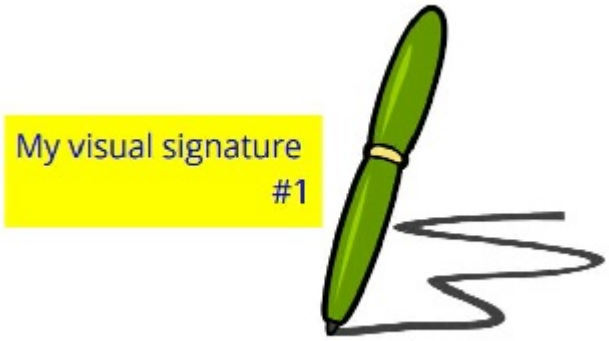
19.4.3.1.4. Text and image combination

DSS provides a set of functions to align a text respectively to an image. The parameters must be applied to a `SignatureImageTextParameters` object :

Combination of text and image parameters

```
// Specifies a text position relatively to an image (Note: applicable only for joint
image+text visible signatures).
// Thus with _SignerPosition.LEFT_ value, the text will be placed on the left side,
// and image will be aligned to the right side inside the signature field
textParameters.setSignerTextPosition(SignerTextPosition.LEFT);
// Specifies a horizontal alignment of a text with respect to its area
textParameters.setSignerTextHorizontalAlignment(SignerTextHorizontalAlignment.RIGHT);
// Specifies a vertical alignment of a text block with respect to a signature field
area
textParameters.setSignerTextVerticalAlignment(SignerTextVerticalAlignment.TOP);
```

The result of applying the foregoing transformations is provided on the image below:



19.4.3.1.5. Fonts usage

You can create a custom font as following, for a physical font:

Add a custom font as a file

```
// Initialize text to generate for visual signature
DSSFont font = new
DSSFileFont(getClass().getResourceAsStream("/fonts/OpenSansRegular.ttf"));
```

For a logical font:

Java font usage

```
SignatureImageTextParameters textParameters = new SignatureImageTextParameters();
DSSFont font = new DSSJavaFont(Font.SERIF);
font.setSize(16); // Specifies the text size value (the default font size is 12pt)
textParameters.setFont(font);
textParameters.setTextColor(Color.BLUE);
textParameters.setText("My visual signature");
imageParameters.setTextParameters(textParameters);
```

For a native font:

Native font usage

```
textParameters.setFont(new PdfBoxNativeFont(PDType1Font.HELVETICA));
```

19.4.4. JAdES

A typical example of a `JAdES-BASELINE-B` signature creation is represented below:

Signing a file with JAdES

```
// Prepare parameters for the JAdES signature
JAdESSignatureParameters parameters = new JAdESSignatureParameters();
// Choose the level of the signature (-B, -T, -LT, -LTA).
parameters.setSignatureLevel(SignatureLevel.JAdES_BASELINE_B);
// Choose the type of the signature packaging (ENVELOPING, DETACHED).
parameters.setSignaturePackaging(SignaturePackaging.ENVELOPING);
// Choose the form of the signature (COMPACT_SERIALIZATION, JSON_SERIALIZATION,
FLATTENED_JSON_SERIALIZATION)
parameters.setJwsSerializationType(JWSSerializationType.COMPACT_SERIALIZATION);

// Set the digest algorithm
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);
// Set the signing certificate
parameters.setSigningCertificate(privateKey.getCertificate());
// Set the certificate chain
parameters.setCertificateChain(privateKey.getCertificateChain());

// Create common certificate verifier
CommonCertificateVerifier commonCertificateVerifier = new CommonCertificateVerifier();
// Create JAdESService for signature
JAdESService service = new JAdESService(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 JAdESService to sign the document with the signature value obtained
in
// the previous step.
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

19.4.5. ASiC

19.4.5.1. ASiC-S

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

Sign a file within an ASiC-S container

```
// Preparing parameters for the AsicS signature
ASiCWithXAdESSignatureParameters parameters = new ASiCWithXAdESSignatureParameters();
// We choose the level of the signature (-B, -T, -LT, LTA).
parameters.setSignatureLevel(SignatureLevel.XAdES_BASELINE_B);
// We choose the container type (ASiC-S or ASiC-E)
parameters.aSiC().setContainerType(ASiCContainerType.ASiC_S);

// 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 ASiC service for signature
ASiCWithXAdESService service = new ASiCWithXAdESService(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);
```

19.4.5.2. ASiC-E

This is another example of the source code for signing multiple documents using ASiC-E based on CAdES-BASELINE-B:

Sign multiple files within an ASiC-E container

```
// Preparing the documents to be embedded in the container and signed
List<DSSDocument> documentsToBeSigned = new ArrayList<>();
documentsToBeSigned.add(new FileDocument("src/main/resources/hello-world.pdf"));
documentsToBeSigned.add(new FileDocument("src/main/resources/xml_example.xml"));

// Preparing parameters for the ASiC-E signature
ASiCWithCAdESSignatureParameters parameters = new ASiCWithCAdESSignatureParameters();

// We choose the level of the signature (-B, -T, -LT or -LTA).
parameters.setSignatureLevel(SignatureLevel.CAdES_BASELINE_B);
// We choose the container type (ASiC-S pr ASiC-E)
parameters.aSiC().setContainerType(ASiCContainerType.ASiC_E);

// 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 ASiC service for signature
ASiCWithCAdESService service = new ASiCWithCAdESService(commonCertificateVerifier);

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

19.5. Examples of SCA and SCDev Topology and Workflows

19.5.1. Hash computation

In order to avoid transfer of original or sensitive information, and also to reduce the amount of data by online protocols, a hash of a document or data to be signed can be computed.

Hash computation

```
// Compute hash on a DSSDocument
DSSDocument document = new InMemoryDocument("Hello World!".getBytes());
String base64Sha256HashOfDocument = document.getDigest(DigestAlgorithm.SHA256);

// Compute hash on a byte array
byte[] binaries = "Hello World".getBytes();
byte[] sha256HashOfBinaries = DSSUtils.digest(DigestAlgorithm.SHA256, binaries);
String base64Sha256HashOfBinaries = Utils.toBase64(sha256HashOfBinaries);
```

19.5.2. Detached signature based on digested document

When you want to keep your original documents private, a signature can be created in a detached way, by providing the digest of an original document only. You can find an example of a use case below:

Detached signature based on digested document

```
// Create a DigestDocument from original DSSDocument
DigestDocument digestDocument = new DigestDocument();
digestDocument.addDigest(DigestAlgorithm.SHA256,
originalDocument.getDigest(DigestAlgorithm.SHA256));

// Preparing parameters for a signature creation
CAdESSignatureParameters parameters = new CAdESSignatureParameters();
parameters.setSigningCertificate(privateKey.getCertificate());
parameters.setCertificateChain(privateKey.getCertificateChain());
parameters.setSignatureLevel(SignatureLevel.CAdES_BASELINE_B);

// Set the detached packaging, as a digest only will be included into the signature,
and the original content
parameters.setSignaturePackaging(SignaturePackaging.DETACHED);

// The same DigestAlgorithm shall be used as the one used to create the DigestDocument
parameters.setDigestAlgorithm(DigestAlgorithm.SHA256);

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

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

// Sign the ToBeSigned data
SignatureValue signatureValue = signingToken.sign(dataToSign,
parameters.getDigestAlgorithm(), privateKey);

// We invoke the signature service to create a signed document incorporating the
obtained Sig
DSSDocument signedDocument = service.signDocument(digestDocument, parameters,
signatureValue);

// Initialize the DocumentValidator
DocumentValidator documentValidator =
SignedDocumentValidator.fromDocument(signedDocument);

// Set the CertificateVerifier
documentValidator.setCertificateVerifier(commonCertificateVerifier);

// Provide the original or digested document as a detached contents to the validator
documentValidator.setDetachedContents(Arrays.asList(originalDocument));

// Validate the signed document
Reports reports = documentValidator.validateDocument();
```

19.5.3. Client-side signature creation with server-side remote key activation

When a private key signing is operated remotely, i.e. on an external server, then the document preparation and the actual signing can be separated. The signature document is created on client's side and the actual signature is made remotely. Refer to section Client-side signature creation with server-side remote key activation for a detailed description and visual illustration of the steps that take place in such a situation. See the code below for a code illustration:

Creation of the signature envelope on client side and signature value on server side

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

// Compute the hash of ToBeSigned data to send to the remote server
byte[] toBeSignedDigest = DSSUtils.digest(parameters.getDigestAlgorithm(),
dataToSign.getBytes());
Digest digest = new Digest(parameters.getDigestAlgorithm(), toBeSignedDigest);

// Provide the hash of ToBeSigned data to the remote server for signing
SignatureValue signatureValue = serverSignDigest(digest);
//SignatureValue signatureValue = serverSign(dataToSign,
parameters.getDigestAlgorithm());

// We invoke the signature service to create a signed document incorporating the
obtained Sig
DSSDocument signedDocument = service.signDocument(toSignDocument, parameters,
signatureValue);
```

19.6. Interpreting a detailed report

We will illustrate here how to read a detailed validation report where the validation succeeds even though the signing certificate has been revoked.

This will illustrate how to look up at which check failed and how the overall validation process can succeed even when a sub-process failed.

First, as explained in Detailed report the structure of the detailed validation report is based on ETSI EN 319 102-1 ([R09]). This means that the detailed report is structured in terms of:

- Validation processes; and
- Building blocks.

There are three validation processes specified in ETSI EN 319 102-1:

- The Validation process for Basic Signatures;
- The Validation process for Signatures with Time and Signatures with Long-Term Validation Material;
- The Validation process for Signatures providing Long Term Availability and Integrity of

Validation Material, abbreviated in the report as "Validation Process for Signatures with Archival Data".

Those validation processes in turn rely on building blocks, which are denoted in ETSI EN 319 102-1 as:

- The basic building blocks;
- The time-stamp validation building block;
- The additional building blocks.

DSS groups the basic building blocks and the additional building blocks related to the validation of a particular signature together. However, it separates the time-stamp validation builiding block from the rest and present it alongside the validation processes because this building block essentially consists in applying the validation process for basic signatures to a timestamp token taken as a CMS object.

Now let’s see how this looks in a validation report (we use here the HTML representation provided by the demonstration).

First, as mentioned, are the validation processes and the time-stamp validation building block, which are numbered in the figure below with:

1. The validation process for Basic Signatures;
2. The time-stamp building block in which appears the validation process of the timestamp;
3. The validation process for Signatures with Time and Signatures with Long-Term Validation Material;
4. The validation process for Signatures with Archival Data.

Signature S-1DBA602183912ECC2A0F59DADC7EB53D9051121B0831C72717652CFDC7A34790 TOTAL_PASSED

Validation Process for Basic Signatures (Best signature time : 2022-02-03 13:30:16 (UTC)) 1 REVOKED_NO_POE

Is the result of the 'Format Checking' building block conclusive?	OK
Is the result of the 'Identification of Signing Certificate' building block conclusive?	OK
Is the result of the 'Validation Context Initialization' building block conclusive?	OK
Is the result of the 'X.509 Certificate Validation' building block conclusive?	WARNING : The result of the 'X.509 Certificate Validation' building block is not conclusive!
Is the signing certificate not revoked at validation time?	WARNING : The signing certificate is revoked at validation time!
Is the validation time in the validity range of the signing certificate?	OK
Is the result of the 'Cryptographic Verification' building block conclusive?	OK
Is the result of the Basic Validation Process conclusive? Basic Signature Validation process failed with INDETERMINATE/REVOKED_NO_POE indication	NOT OK : The result of the Basic validation process is not conclusive!

Timestamp T-8D27E2D5B3B90E91C9341A5442FB7C123B26F0DB97043AED6CB431AC8F8AA231 2 PASSED

Validation Process for Time-stamps PASSED
SIGNATURE_TIMESTAMP (Production time : 2021-10-19 14:43:44 (UTC))

Is the result of the 'Identification of Signing Certificate' building block conclusive?	OK
Is the result of the 'X.509 Certificate Validation' building block conclusive?	OK
Is the result of the 'Cryptographic Verification' building block conclusive?	OK
Is the result of the 'Signature Acceptance Validation' building block conclusive?	OK

Time-stamp Qualification QTSA

Has a trusted list been reached for the certificate chain?	OK
Is the list of trusted lists acceptable? Trusted List : https://ec.europa.eu/tools/lotl/eu-lotl.xml	OK
Is the trusted list acceptable? Trusted List : https://ssi.gouv.fr/uploads/tl-fr.xml	OK
Has been an acceptable trusted list found?	OK
Is the certificate related to a TSA/QTST?	OK
Is the certificate related to a trust service with a granted status?	OK
Is the certificate related to a trust service with a granted status at the production time?	OK

Validation Process for Signatures with Time and Signatures with Long-Term Validation Data (Best signature time : 2021-10-19 14:43:44 (UTC)) 3 PASSED

Is the result of the Basic Validation Process acceptable?	OK
Is an acceptable revocation data present for the certificate? Latest acceptable revocation : R-927B40E8EB48E0DE537756B3B3C9282CEC7021A34E18365C219F256B1C2BF6BD	OK
Does the message-imprint match the computed value? Signature Timestamp with Id = T-8D27E2D5B3B90E91C9341A5442FB7C123B26F0DB97043AED6CB431AC8F8AA231, production time = 2021-10-19 14:43	OK
Is the result of basic time-stamp validation process conclusive? Signature Timestamp with Id = T-8D27E2D5B3B90E91C9341A5442FB7C123B26F0DB97043AED6CB431AC8F8AA231, production time = 2021-10-19 14:43	OK
Is the revocation time after best-signature-time? Best-signature-time : 2021-10-19 14:43, revocation time : 2021-10-27 11:23	OK
Are the time-stamps in the right order?	OK
Is the signed qualifying property: 'signing-time' present?	OK
Is the signing-time plus the time-stamp delay after best-signature-time?	IGNORED : The check is skipped by the validation policy
Is the signature acceptable?	OK

Certificate Revocation Data Selector
Id = C-528A80DE87B888CC70EAC34EDF14D46294346305D41E4BBC616BCB24EC3A01D5

Is the result of the revocation data basic validation process acceptable? Id = R-927B40E8EB48E0DE537756B3B3C9282CEC7021A34E18365C219F256B1C2BF6BD	OK
Is the revocation acceptance check conclusive? Id = R-927B40E8EB48E0DE537756B3B3C9282CEC7021A34E18365C219F256B1C2BF6BD, thisUpdate = 2022-02-03 13:25, production time = 2022-02-03 13:25	OK
Is an acceptable revocation data present for the certificate? Latest acceptable revocation : R-927B40E8EB48E0DE537756B3B3C9282CEC7021A34E18365C219F256B1C2BF6BD	OK

Validation Process for Signatures with Archival Data (Best signature time : 2021-10-19 14:43:44 (UTC)) 4 PASSED

Is the result of the LTV validation process acceptable?	OK
Is the result of the Time-stamp Validation Building Block acceptable? Signature Timestamp with Id = T-8D27E2D5B3B90E91C9341A5442FB7C123B26F0DB97043AED6CB431AC8F8AA231, production time = 2021-10-19 14:43	OK
Is the result of basic time-stamp validation process conclusive? Signature Timestamp with Id = T-8D27E2D5B3B90E91C9341A5442FB7C123B26F0DB97043AED6CB431AC8F8AA231, production time = 2021-10-19 14:43	OK
Is the digest algorithm reliable at lowest POE time for the time-stamp token? Digest algorithm SHA256 at validation time : 2022-02-03 13:30 for time-stamp message imprint with Id : T-8D27E2D5B3B90E91C9341A5442FB7C123B26F0DB97043AED6CB431AC8F8AA231	OK
Does the message-imprint match the computed value? Signature Timestamp with Id = T-8D27E2D5B3B90E91C9341A5442FB7C123B26F0DB97043AED6CB431AC8F8AA231, production time = 2021-10-19 14:43	OK

As further illustrated in the figure above:

- The validation process for Basic signature is executed against the first time which is the (current) validation time;
- The validation process for Signatures with Time and Signatures with Long-Term Validation Material is executed against a second time which is the "best signature time" that is determined using the signature timestamp;
- The validation process for Signatures with Archival Data is executed against a third time which is the "best signature time" determined using all time assertions present in the signature.

Additionally, each process has an associated indication, here:

1. `REVOKED_NO_POE` for the validation process for basic signatures;
2. `PASSED` for the timestamp validation block;
3. `PASSED` for the validation process for Signatures with Time and Signatures with Long-Term Validation Material;
4. `PASSED` for the validation process for Signatures with Archival Data.

Each of those indications is determined using the result of the associated building blocks, and applying additional checks.

Here we can see that the `REVOKED_NO_POE` indication arises from the fact that the result of the "X.509 Certificate Validation" building block is not conclusive.

Now delving into the building blocks themselves, we can see in the figure below that the building blocks are grouped by the signed objects to which they relate:

- `BBB SIG` are the building blocks used for the validation of the signature itself;
- `BBB TIMESTAMP` are the building blocks used for the validation of the timestamp;
- `BBB REVOCATION DATA` are the building blocks used for the validation of the OCSP responses taken as CMS objects.

Validation results

We saw previously that the "validation process for basic signature" resulted in the REVOKED_NO_POE indication because the result of the "X.509 Certificate Validation" building block was not conclusive.

To check what went wrong, we must therefore look at the "X.509 Certificate Validation" building block associated to the signature, that is the "X.509 Certificate Validation" building block that is in BBB SIG.

We see there that the check "Is the certificate validation conclusive?" has failed. Therefore, we now need to look at the "Certificate" sub-block of BBB SIG.

In the "Certificate" sub-block we can see that all checks succeeded except for "Is the certificate not revoked?". We can thus conclude that the validation process for basic signature resulted in the indication REVOKED_NO_POE because the signing certificate is revoked at validation time.

That being said, we saw before that although the validation process for basic signature failed with REVOKED_NO_POE, the other validation processes resulted in the PASSED indication. And in fact, the overall result of the validation process is TOTAL_PASSED.

To understand why that is so, we need to look back at the signature validation processes. There we can see in the "validation process for Signatures with Time and Signatures with Long-Term Validation Material" that specific checks differing from the building blocks are executed. Which checks are executed depends on the indication determined during the "validation process for basic signatures". In the present case, because the indication was REVOKED_NO_POE the specific check "Is the revocation time after best-signature-time" is executed.

As mentioned before, *best-signature-time* is determined, for that validation process, using the signature timestamp. Because here the validation of the signature timestamp succeeded, the time indicated in the timestamp is used as *best-signature-time*, and because this time is indeed before the time of revocation of the signing certificate, the check succeeds, and the whole "validation process for Signatures with Time and Signatures with Long-Term Validation Material" succeeds.

Now to understand why the overall result of the validation is TOTAL_PASSED, we need to go back to the procedures specified in ETSI EN 319 102-1 (cf. [R09]). The three validation processes specified in that standard are in fact not independent:

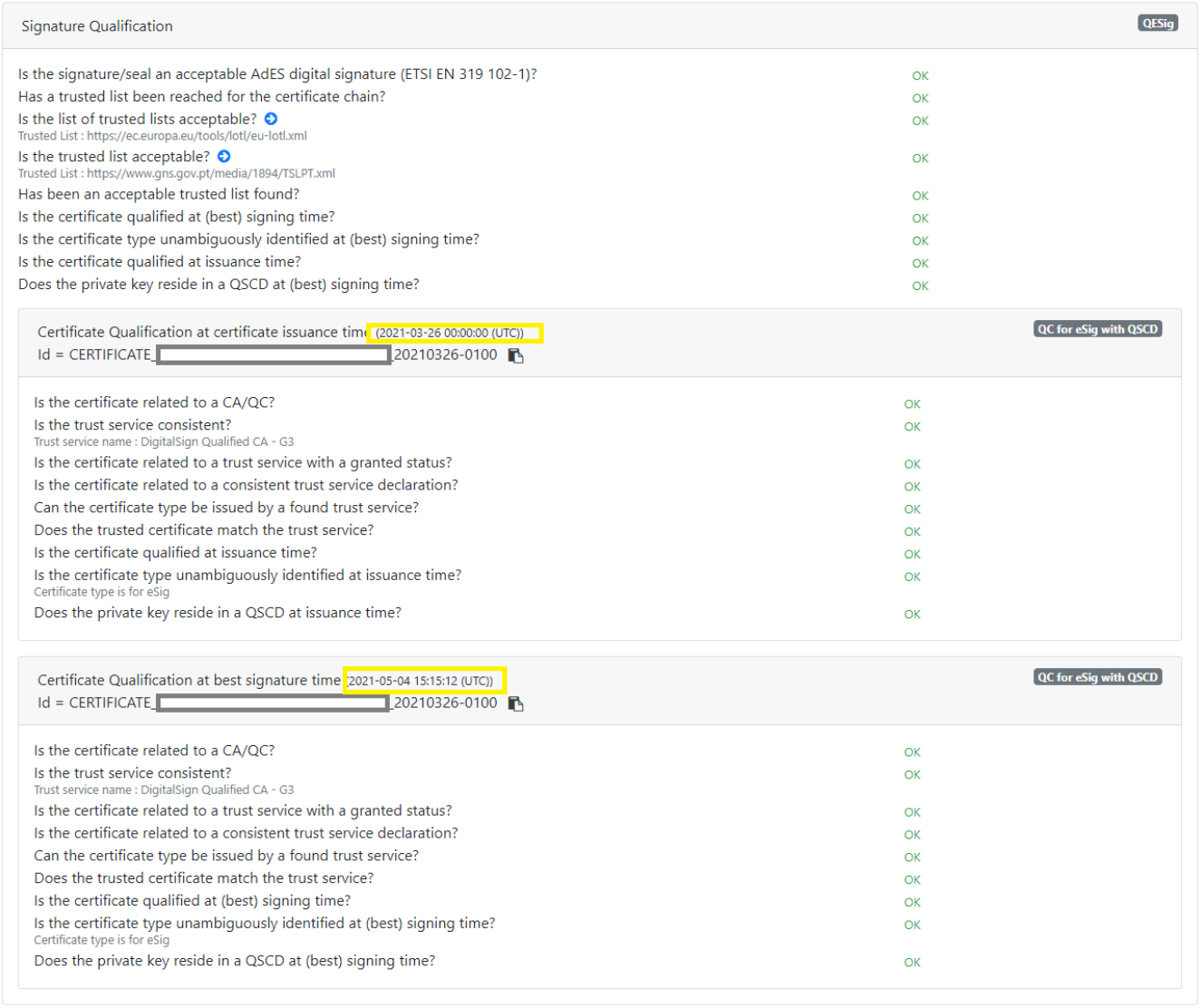
- The "validation process for Signatures providing Long Term Availability and Integrity of Validation Material" calls the "validation process for Signatures with Time and Signatures with Long-Term Validation Material"; and
- The "validation process for Signatures with Time and Signatures with Long-Term Validation Material" itself calls the "validation process for basic signatures".

The overall validation result is then provided as the indication returned by the validation process against which the validation was performed.

Although it is possible to only run the validation process for basic signature, in our case the process that was run was the "validation process for Signatures providing Long Term Availability and Integrity of Validation Material" which required to run the other two validation processes.

Therefore, because that validation process returned PASSED, the overall validation result is TOTAL_PASSED.

Finally, the report contains information on the determination of the qualification of the signature.



This determination is not specified in ETSI EN 319 102-1 ([R09]), but rather in ETSI TS 119 172-4 ([R10]).

Essentially, a signature can be determined as qualified if:

1. The result of running the "validation process for Signatures providing Long Term Availability and Integrity of Validation Material" defined in ETSI EN 319 102-1 is `TOTAL_PASSED`;
2. The signing certificate is determined as qualified at best-signature-time and at issuance time (the time when the certificate was issued i.e. the value of the "notBefore" field);
3. The private key corresponding to the signing certificate is determined as being held in a qualified signature creation device (QSCD).

We discussed above the validation processes defined in ETSI EN 319 102-1. The determinations of point 2 and 3 on the other hand rely on the procedures specified in ETSI TS 119 615 ([R14]).

Without going into details, ETSI TS 119 615 specifies procedures for interpreting the content of EUMS trusted lists, including procedures for validating EUMS trusted lists.

Illustrated in the figure above are the results of the main steps defined in that standard.