******

DG JUSTICE

Framework Contract No DI/06769-00

Specific Contract No 12960

**Digital Signature Service for DG Justice**

**Technical Analysis**

**Software Architecture Document - Public**



|  |  |
| --- | --- |
| Subject |  |
| Version |  |
| Release Date |  |
| Filename |  |
| Document Reference |  |

Document History

|  |  |  |  |
| --- | --- | --- | --- |
| Document History | | | |
| Version | Release Date | Author | Description |
| 0.01 | 12/05/2014 | Peter Macfarlane | Initial version created for QTM4 of Specific contract No 7712 under Framework Contract No DI/06769-00 |
| 1.00 | 13/06/2014 | Peter Macfarlane | QTM4 of Specific contract No 7712 under Framework Contract No DI/06769-00  Update according to customer comments |
| 1.01 | 25/02/2015 | Peter Macfarlane | QTM4 of SC12960  Securing communication between server and applet |
| 1.02 | 12/03/2015 | Peter Macfarlane | QTM4 of SC12960  Update according to customer comments |

|  |  |  |  |
| --- | --- | --- | --- |
| Reviews | | | |
| Version | Release Date | Reviewer | Position |
| 0.01 | 18/04/2014 | - | - |
| 1.00 | 13/06/2014 | - | - |
| 1.01 | 25/02/2015 | - | - |
| 1.02 | 12/03/2015 | - | - |

Table Of Contents

[1 Introduction 7](#_Toc413924857)

[1.1 Terminology 7](#_Toc413924858)

[1.2 Purpose of the Document 7](#_Toc413924859)

[1.3 Scope of the Document 7](#_Toc413924860)

[1.4 Intended Audience 7](#_Toc413924861)

[1.5 Structure of the Document 7](#_Toc413924862)

[2 Architectural Goals and Constraints 8](#_Toc413924863)

[2.1 Architectural Goals 8](#_Toc413924864)

[2.1.1 Multi-Platform Viability 8](#_Toc413924865)

[2.1.2 User-Friendly Operation 8](#_Toc413924866)

[2.1.3 Reliability 8](#_Toc413924867)

[2.1.4 Maintainability 8](#_Toc413924868)

[2.2 Extensibility 8](#_Toc413924869)

[2.3 Architectural Constraints 8](#_Toc413924870)

[2.3.1 Operating Environment 9](#_Toc413924871)

[2.3.2 Software Constraints 9](#_Toc413924872)

[3 Security 10](#_Toc413924873)

[3.1 Application Security 10](#_Toc413924874)

[4 Use-Case View 11](#_Toc413924875)

[5 Logical View 12](#_Toc413924876)

[5.1 Overview 12](#_Toc413924877)

[5.2 Architecturally Significant Design Packages 12](#_Toc413924878)

[5.2.2 Signing Controller 13](#_Toc413924879)

[5.2.3 DSS 13](#_Toc413924880)

[5.3 Use-Case Realisations 15](#_Toc413924881)

[5.3.1 Signing Service – Smart Card or Windows Certificate Store 15](#_Toc413924882)

[5.3.2 Signing Service – PKCS#12 19](#_Toc413924883)

[5.3.3 Signing Controller 20](#_Toc413924884)

[5.3.4 Server Seal 23](#_Toc413924885)

[6 Deployment View 24](#_Toc413924886)

[7 Implementation View 26](#_Toc413924887)

[8 Data View 28](#_Toc413924888)

[8.1 Card Profile 28](#_Toc413924889)

[8.1.1 Entities 28](#_Toc413924890)

[8.1.2 Choice of Algorithm 29](#_Toc413924891)

[8.2 Statistics Logging 29](#_Toc413924892)

[8.2.1 Entities 30](#_Toc413924893)

[9 Size and Performance 31](#_Toc413924894)

[10 Detailed Design 32](#_Toc413924895)

[10.1 Signing Applet 33](#_Toc413924896)

[10.1.1 Environment Detection 33](#_Toc413924897)

[10.1.2 Applet – Signing Controller Communication 34](#_Toc413924898)

[10.1.3 User Interface 35](#_Toc413924899)

[10.1.4 Signature Management 35](#_Toc413924900)

[10.2 Signing Controller 36](#_Toc413924901)

[10.2.1 Configuration 36](#_Toc413924902)

[10.2.2 Smart Card Signing Methods 36](#_Toc413924903)

[10.2.3 Smart Card Profile Management 38](#_Toc413924904)

[10.2.4 Document Management 40](#_Toc413924905)

[10.2.5 Enterprise Application Interface 40](#_Toc413924906)

[10.3 Enterprise Application 40](#_Toc413924907)

[11 Appendix 42](#_Toc413924908)

[11.1 Smart Card Profile XSD 42](#_Toc413924909)

List of Tables

[Table 1: Reference Documents 6](#_Toc413924910)

[Table 2: Applicable Documents 6](#_Toc413924911)

[Table 3: Architecturally Significant Use Case 11](#_Toc413924912)

[Table 4: DSS Token APIs 13](#_Toc413924913)

[Table 5: Signing Sequence Diagram (JRE 6) – Description 18](#_Toc413924914)

[Table 6: Applet-Server Interface 21](#_Toc413924915)

[Table 7: Signing Controller – Enterprise Application Interface 22](#_Toc413924916)

[Table 8: Signing Service Components 32](#_Toc413924917)

[Table 9: Signing Service Artifacts 33](#_Toc413924918)

[Table 10: Signing Controller – Configuration – XML Schema – Signing Methods 38](#_Toc413924919)

List of Figures

[Figure 1: Logical View - Overview 12](#_Toc413924920)

[Figure 2: Signing Sequence Diagram (Smart Card - JRE 6) 16](#_Toc413924921)

[Figure 3: Signing Sequence Diagram – Signing Controller 20](#_Toc413924922)

[Figure 4: Signing Service - Deployment View 24](#_Toc413924923)

[Figure 5: Signing Service - Implementation View 26](#_Toc413924924)

[Figure 6: Signing Service - Data View – Card Profile Repository 28](#_Toc413924925)

[Figure 7: Signing Service - Data View – Card Profile Repository 29](#_Toc413924926)

[Figure 8: Applet-Webapp communication - principle 34](#_Toc413924927)

[Figure 9: Signing Controller – Configuration – XML Schema – Top Level 36](#_Toc413924928)

[Figure 10: Signing Controller – Configuration – XML Schema – Signing Methods 37](#_Toc413924929)

[Figure 11: Smart card profile data store XML schema – top level 38](#_Toc413924930)

[Figure 12: Smart card profile data store XML schema – Card Profiles 39](#_Toc413924931)

[Figure 13: Smart card profile data store XML schema – Defaults 40](#_Toc413924932)

Reference and Applicable Documents

This section contains the lists of all reference and applicable documents. When referring to any of the documents below, the bracketed reference will be used in the text, such as [[R01](#R01)].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reference Documents | | | | |
| Ref. | Title | Reference | Version | Date |
| GLO | Digital Signature Service for DG Justice - Glossary | DSS4eJustice-GLO | 1.01 | 31/12/2013 |
| FS | Digital Signature Service for DG Justice - Functional Analysis - Public | DSS4eJustice-FSP | 1.00 | 13/06/2014 |

Table 1: Reference Documents

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Applicable Documents | | | | |
| Ref. | Title | Reference | Version | Date |
| A01 | Framework Contract No DI/06769-00 | N/A | N/A | 10/09/2010 |
| A02 | Specific contract No 12960 | JUST/2013/JCIV/FW-A/0170/A4/e-Justice portal IRI/ECLI/DSS | N/A | 28/03/2014 |

Table 2: Applicable Documents

# Introduction

## Terminology

The “product name” of the software is the Digital Signature Applet. This consists of a Java applet that runs in a user’s browser. This applet is supported by software that runs on the server. For clarity, we refer to the “signing service” when referring to the software in general, and “signing applet” or “applet” when referring to the applet component. We also use the term “enterprise application” by which we mean the overall application in which the signing service and signing applet are deployed.

## Purpose of the Document

This document provides an architectural overview of the signing service within the enterprise application. It aims to show the architectural choices made during the design of this service.

## Scope of the Document

The scope of the document includes the technical aspects of the signing service.

## Intended Audience

The document is addressed to:

* Any technical person having an interest in using the signing service.

## Structure of the Document

The document is organised as follows:

Chapter 1 **Introduction** is this introduction;

Chapter 2 **Architectural Goals and Constraints** describes the architectural goals and constraints arising from non-functional requirements;

Chapter 3 **Security** describes the security considerations;

Chapter 4 **Use-Case View** lists the architecturally significant use cases;

Chapter 5 **Logical View** presents the architecturally significant design packages and the main use case implementations;

Chapter 6 **Deployment View** describes the hardware and software nodes of the deployment;

Chapter 7 **Implementation View** describes the software layers of the signing service;

Chapter 8 **Data View** describes the data entities involved in the signing service;

Chapter 9 **Size and Performance**  considers the size and performance issues.

Chapter 10 **Detailed Design** describes the design of the software in more detail.

# Architectural Goals and Constraints

## Architectural Goals

This section presents the architectural goals arising from non-functional requirements.

### Multi-Platform Viability

The signing service must be usable with a comprehensive selection of operating system, Java runtime, browser vendor and version, and smart card types.  
This goal is achieved by environment detection and customising the applet to adapt to the different client environments, all validated by careful testing.

### User-Friendly Operation

The signing service must require the least possible input from the user. This is achieved by investing in the environment detection capabilities of the signing applet and a smart card fingerprinting mechanism which limits the amount of information that must be gathered from the user.

The signing applet provides only control and business logic, allowing seamless integration into the user interface.

### Reliability

The reliability of the signing service is achieved by delegating all signature operations to the DSS library.

Reliability of the applet is also a major point, especially regarding the interaction with the smart card. In particular, when the user tries to use an unsupported smart card, or if the communication between the applet and the smart card does not work properly, the applet has to clearly indicate to the user that there is a problem (not necessarily the exact cause) without crashing (and leaving the user without any idea of the problem).

### Maintainability

Maintainability and upgradeability of the signing service is achieved by respecting the following design principles:

* Encapsulation of functionality in different layers, e.g. separation of core business model, control logic and presentation logic.
* Re-use of well-established frameworks.
* Object oriented design (OOD).Separation of concern and sound documentation. The enterprise application and the Signing Service should be able to evolve independently, so they should be as loosely coupled as possible.

## Extensibility

The signing applet is loosely coupled to the presentation layer which allows it to be used in different enterprise applications and easily accommodate changes to the presentation layer design.

## Architectural Constraints

This section presents the architectural constraints.

### Operating Environment

The signing service must be deployable in commonly used Application Servers. The reference application server is JBoss.

The signing service client (the applet) should support a variety of operating system, browser and JRE.

### Software Constraints

As far as possible the signing service uses open source software.

The service uses the DSS library for signature operations. The PKCS11 signature API relies on third party drivers whose distribution is constrained by licensing. For certain smartcards in certain environments, the user may be required to provide the driver.

# Security

## Application Security

The security communication between the signing applet (via the web page) and the signing service in the enterprise application, and the communication between the signing service and external signature verification services can be assured by using HTTPS.

# Use-Case View

This following table lists the use cases which have been identified as having a significant architectural impact.

|  |  |
| --- | --- |
| **Use Case** | **Description** |
| Detect Environment | The system detects the technical environment of the signing service user and prepares the user interface for the electronic signature of PDF document. |
| Sign PDF Online | The signing service user applies his/her digital signature to a PDF form... |

Table 3: Architecturally Significant Use Case

# Logical View

## Overview

The following figure describes the logical layers of the signing service.



Figure 1: Logical View - Overview

The Web Layer includes the web page – and associated controller – used to get the signed PDF from the user. The Web Layer includes also an Applet that can interact with a signing device.

The Signing Applet and the Signing Controller both rely on the DSS library – the former for applying the digital signature and the latter for validating the digital signature.

The Service Layer represents the services of the portal that the Signing Service relies upon. These are exposed to the Signing Controller by the Portal Façade, which is a well-defined interface (refer to Table 7).

The External Online Signing Service is an external service to which the browser posts the PDF document to be signed and which returns the signed document to the Signing Controller.

## Architecturally Significant Design Packages

#### Signing Service Applet

To sign a document, an applet is loaded into the browser to interact with a smart card terminal and to sign the PDF. The applet communicates with the Signing Controller to establish the signature context, obtain the PDF to sign, deliver the signed PDF, and verify the signed PDF.

The applet does not *directly* communicate with the Signing Controller. Instead, to make a request to the server the applet calls a JavaScript function in the enclosing web page, which calls back to the applet with the response. This allows the applet to re-use the session established between the browser and the server and avoids potential problems with proxy servers.

The applet does not directly provide a user interface but manipulates the user interface components in the enclosing HTML page (for example, to list the available certificates from the smart card or to display status messages) or is triggered by JavaScript calls made from the enclosing web page (for example, the “onclick” event of a button can call a method in the applet). This allows the visual style of the enterprise application web layer to be fully respected.

### Signing Controller

The Signing Controller provides the following functionality:

* It has a data store of the APIs, digest algorithms and signature algorithms supported by particular smartcard types which it uses to construct the context of the electronic signature, that is, the capabilities of the signature device.
* It prepares a sealed PDF for signature by the signing service user.
* It provides validation of the signed PDF;
* It logs information concerning the use of the signing service;

### DSS

DSS is a software library that can create and validate PAdES forms up to LTV. In particular it supports the requirements in Commission Decision 2011/130/EU and for verification makes use of Member States' trusted lists of certificate authorities.

The DSS library is used by the Signing Applet to sign the PDF document.

The DSS library is used by the Signing Controller to verify the signature.

#### Signature Token APIs

The DSS library provides several APIs to connect to an SSCD. The following table lists the APIs and their platform and JRE constraints.

|  |  |  |
| --- | --- | --- |
| **API** | **Platform** | **Minimum JRE version** |
| MS-CAPI | Windows | 1.6 |
| PKCS11 | Windows, MacOS, Linux | 1.5 |
| MOCCA | Windows, MacOS, Linux | 1.6 |
| PKCS12 | Windows, MacOS, Linux | 1.5 |

Table 4: DSS Token APIs

The MS-CAPI API is available for the Windows platform only and requires at least JRE version 1.6. It relies on pre-installed middleware to access the smart card.

The PKCS11 API is available for all platforms. It requires the path to a device driver.

The MOCCA API is available for all platforms and requires at least JRE version 1.6. It is compatible with a restricted number of smart cards. It uses the *javax.smartcardio* package to directly communicate with the smart card. Note that MOCCA has particular licensing constraints and is not free to use.

The PKCS12 API is available for all platforms. The private key and certificate are stored in a file. The API requires the location of the file and, optionally, a password to open it.

#### MS-CAPI – Improved Connector

The default MS-CAPI connector provided in the DSS library has the unfortunate behaviour of trying to open cached certificates in the Windows certificate store that are linked to cards not currently inserted in the computer. In certain cases, this shows a modal dialog to the user with the message “Please insert card”, even though the user has inserted the card. This is sometimes true even if the user has inserted the card for the cached certificates.

The default MS-CAPI connector is based on the sun.security.mscapi.SunMSCAPI class that is included with the standard JRE. This class uses the sunmscapi.dll, also included with the standard JRE. In order to overcome the limitations of this implementation, the source code for these two artifacts is adapted.

##### Adapted sunmscapi.dll

The adapted sunmscapi.dll uses the following functions from the Windows Crypt32.lib library.

|  |  |
| --- | --- |
| **Function** | **Comment** |
| CertEnumCertificatesInStore | Enumerates all certificates in a windows certificate store. The function returns a pointer to the next entry in the certificate store. |
| CryptAcquireCertificatePrivateKey | Obtains a handle to the certificate private key. If called with the CRYPT\_ACQUIRE\_SILENT\_FLAG the call fails if a dialog is required and instead sets the NTE\_SILENT\_CONTEXT error code. |
| CryptGetUserKey | Retrieves a handle to one of the user’s public/private key pairs. The call fails if the handle to the certificate private key was obtained with the NTE\_SILENT\_CONTEXT flag and a dialog is required. |

The adapted sunmscapi.dll provides a method that lists certificates in the windows certificate store. The strategy is to enumerate over all certificates, and with each certificate:

1. Call CryptAcquireCertificatePrivateKey for the certificate using the CRYPT\_ACQUIRE\_SILENT\_FLAG; if the call fails, continue the enumeration;
2. Call CryptAcquireCertificatePrivateKey; if the call fails, continue the enumeration;
3. Call CryptAcquireCertificatePrivateKey again, without the NTE\_SILENT\_CONTEXT flag;
4. Call CryptGetUserKey;
5. Add this certificate to the list that is returned by the function.

In this way, certificates that require a user interface are excluded from the list of certificates, and the user interface modal dialog is never shown. This strategy is successful as a user interface dialog is not needed simply to list the certificates on the device

## Use-Case Realisations

### Signing Service – Smart Card or Windows Certificate Store

The signing service involves the interaction of four high-level components: Smart card, Applet, Browser and Server. The following UML sequence diagram provides an overview of the signing process.



Figure 2: Signing Sequence Diagram (Smart Card - JRE 6)

| Sequence Description | | |
| --- | --- | --- |
|  | navigateToSigningPage | The user accesses a function in the enterprise application that requires signature of a PDF document. |
|  | pageRequest | The browser requests the signing page from the server |
|  | jsp | The JSP generates the signing page. It contains options for smartcard signature, PKCS12 certificate file signature or signature by a certificate stored in the Windows certificate store. |
|  | selectSigningMethod(SC) | The user selects the Smart Card signing method |
|  | pageRequest | The browser requests the page for the Smart Card signing method. |
|  | jsp | The enterprise application provides the page for the Smart Card signing method. |
|  | initPage | The page initialises. A JavaScript function detects that the browser has a Java plug-in with JRE version at least 1.6. The applet is configured to load with JRE 6 capabilities (that is, smart card detection). The browser passes User-Agent and Platform information to the applet as applet parameters. |
|  | startApplet | The browser starts the applet. |
|  | init | The applet initialises |
|  | getSmartCardATR | Using SmardCard I/O, the applet   * detects if a smart card terminal is installed and if there is a card in the terminal; * reads the ATR bytes, which can be used to determine the type of the smart card. |
|  | smartcardATR | The smartcard returns its ATR to the applet. |
|  | getSigningContext | The applet sends the finger print to the browser. The finger print consists of smart card information, java version/vendor, and platform information. |
|  | getSigningContext | The browser makes an AJAX call to send the fingerprint to the server. |
|  | evaluateFingerprint | The server uses its database of finger prints to build the signing context. |
|  | signingContext | The server returns the signing context to the browser.  The signing context consists of the API that can be used to interact with the smartcard and the strongest digest algorithm supported by the smartcard. |
|  | signingContext | The browser returns the signing context to the applet. |
|  | updateUI | Using the signing context, the applet updates the UI via JavaScript (enables signing button, displays certificate ID, …). Additional input from the user is minimised as far as possible. |
|  | pageInitialised | The user sees the initialised page |
|  | signPDF | The user initiates the signature process from the browser, for example by clicking a button. |
|  | signPDF | The browser calls the applet to sign the PDF. |
|  | getPDF | The applet requests the browser for the PDF. |
|  | getPDF | The browser makes an AJAX call to the server to retrieve the PDF |
|  | createPDF | The server prepares the PDF. |
|  | createXML | The server creates the corresponding data file[[1]](#footnote-2). |
|  | attachXML | The server attaches the data file (if provided) to the PDF. |
|  | sealPDF | The server signs (seals) the PDF. |
|  | sealedPDF | The server returns the sealed PDF to the browser. |
|  | sealedPDF | The browser returns the sealed PDF to the applet. |
|  | createDigest | The applet creates a digest of the sealed PDF. |
|  | signDigest | The applet requests the smartcard to sign the digest. |
|  | signedDigest | The smartcard returns the signed digest to the applet. |
|  | wrapSignedDigestInPDF | The applet wraps the signed digest in the PDF, creating the signed PDF. |
|  | sendSignedPDF | The applet requests the browser to send the signed PDF to the server. |
|  | sendSignedPDF | The browser makes an AJAX call, sending the signed PDF to the server. |
|  | validateSignedPDF | The server verifies that the signed PDF originated in the server by validating the server signature.  If a data file is present, the server verifies that the signed PDF corresponds to the current form in the workflow by comparing the data file attached to the PDF with the workflow data file  Validation is completely configurable and can result in blocking errors, warnings or no reaction. |
|  | validateSignature | The server validates the user signature by checking that the content of the PDF matches what the user signed, that the certificate has not been revoked, and that the certificate chain can be traced to the trusted list. Validation is completely configurable and can result in blocking errors, warnings or no reaction. |
|  | tempStoreSignedPDF | The server stores the signed PDF. |
|  | status | The server returns a status to the browser. |
|  | status | The browser returns a status to the applet. |
|  | updateUI | The applet updates the UI according to the status. |
|  | status | The user sees the status of the signed document. |
|  | next | The user can give a decision on how to proceed. |
|  | next | The browser sends the command to the server |
|  | storePDF | If the signed PDF passed the configured validation checks, the server stores the signed PDF, that is, it makes it available to the next step in the workflow |
|  | next,next | The browser displays the page for the next step in the workflow. |

Table 5: Signing Sequence Diagram (JRE 6) – Description

#### Alternative Workflow – No Smart Card

At step 10, the applet fails to detect a smart card. The applet sends the fingerprint without a smart card ATR to allow the server to log the event.

#### Alternative Workflow – Multiple Smart Card

The applet detects more than one smart card. At step 10 the user is warned that he should remove one of the smart cards. The workflow continues.

#### Alternative Workflow – Smart Card ATR not in Data Store

At step 14, the ATR is not in the data store. The fingerprint information and user country (if available) are logged. The workflow continues. At step 19 the applet tries to connect to the smart card as follows:

1. Using the MOCCA connector, if it is available and the card is supported by MOCCA;
2. On a Windows operating system, using the MS-CAPI API;
3. If the MS-CAPI connection fails, or if the operating system is not Windows, using the PKCS#11 API. In this case, the user must be prompted to provide the location of the device driver. Note that it is not possible to know if the MS-CAPI API connected to a smart card, therefore the option to use the PKCS#11 API will always be available.

As the capabilities of the smart card are not known, the applet will use the algorithm choice strategy explained in section 8.1.2.

After successfully signing the document, the user must be prompted to provide the connection parameters, namely the device driver if provided by the user, as well as the smart card issuer. In this way, the information in the smart card data store can be crowd-sourced for the benefit of other users. This is subject to manual validation.

#### Alternative Workflow – PKCS#11 Driver not in expected location

According to the smart card data store, the PKCS#11 API must be used. The default location(s) of driver files are provided in the signing context. At step 19, before connecting to the smart card, the applet verifies that the driver is present in a default location or in a sub-folder. If the driver is not present, the user can provide the location.

The default locations of the driver files can be provided as paths that can include any number of wildcard characters (\*) in the folder or file name components of the path. The search for the driver takes the wildcard into consideration. Matching files are tried in the order they are discovered until one is found to work or all have been tried. The user that configures the default driver path is responsible for providing reasonable values (for example, a path of “c:/Windows/System32/\*.dll” could result in a large number of files for the applet to try).

#### Alternative Workflow – View PDF

The user will be always prompted to view the PDF before signing. If the user wants to view it, the applet then writes the PDF as a temporary file in the local file system and launches the default PDF viewer application to display this file to the user.

The user can also view the signed PDF. The PDF that is signed is held in the applet memory. The user clicks a button on the user interface. The applet prompts the user for a location to save the signed PDF.

### Signing Service – PKCS#12

If Java is available, the user is also able to select with the standard file choice dialog of the browser a PKCS#12 file containing the desired signature token. The applet manages the extraction of the certificate from the file by prompting for a password. The process follows the same sequence as Figure 2 except that:

* the selectSigningMethod operation selects the PKCS12 signing method;
* the signDigest operation is performed by the applet.

### Signing Controller

The signing controller implements the interface between the applet and the enterprise application, i.e.: it is the server-side logic of the signing service. In addition to providing services that are specific to signature (retrieve signing context…), it also acts as a proxy to all other portal services used by the applet, as defined in Table 6; this allows the enterprise application and the applet to evolve indendepently.

The following diagram represents the interaction between the Applet, the Signing Controller and the Enterprise Application.



Figure 3: Signing Sequence Diagram – Signing Controller[[2]](#footnote-3)

The following table defines the interface between the applet and the Signing Controller (mediated by the containing web page).

| **Method** | **Parameter** | **Return** |
| --- | --- | --- |
| getSigningContext | Fingerprint:  + OS  + Architecture  + JRE version  + JRE vendor  + JRE architecture  + ATR (optional) | Signing Context:  + Card Profile  (refer to section 8) |
| downloadPDF | - | Sealed PDF:  + PDF byte array  + PDF Name |
| uploadSignedPDF | Signed PDF:  + PDF byte array | Status:  + SignatureStatus |
| getLocalisedMessages | - | LocalisedMessages:  +language  +code/translation map |
| log | + Log entry | - |

Table 6: Applet-Server Interface

The following table defines the interface between the Signing Controller and the Enterprise Application.

| **Method** | **Parameter** | **Return** | **Description** |
| --- | --- | --- | --- |
| getLocalisedMessages | +HTTPServletRequest  + array of message codes | + Map (code, message) | Gets a list of translated messages corresponding to the provided codes. |
| getPDFDocument | + HTTPServletRequest | + byte array (the unsigned/signed PDF) | Gets the PDF document that must be signed. |
| getPDFDocumentXML | +HTTPServletRequest | + String (the data file of the PDF) | Gets the data file corresponding to the PDF document (optional). |
| getPDFDocumentName | +HTTPServletRequest | + String | Gets the name of the PDF document (to be used as a file name) |
| storePDF | +HTTPServletRequest  +  + SignedForm | - | Provides the signed PDF to the next stage of the workflow.  SignedForm is a wrapper for the PDF (sealed and possibly signed) as a byte array, the detached signature (if provided by the chosen signing method) and an object giving the result of signature validation; |
| getUserCountry  [CURRENTLY NOT USED] | + HTTPServletRequest | + country code associated with the form; the method will currently return null. | Gets the user country. |
| log | + HTTPServletRequest  + DssEvent | - | Logs information, for example statistical information concerning smart card usage. |
| getCardProfileXML | - | +String | Gets the card profile data store XML. The method is provided to allow the portal to manage changes in the data store. A fall back will be implemented to retrieve the XML from a location defined in a properties file. |
| getDocumentValidationConfig | - | DocumentValidationConfig | Configuration of signature validation i.e. which validations are enabled, are warnings, or are errors |
| getCrlSource | - | CRLSource | A CRLSource is a means of connecting to a certificate revocation list. |

Table 7: Signing Controller – Enterprise Application Interface

#### Validation using public key

In order to avoid “man in the browser” attacks, an additional validation mechanism has been put in place. When communicating with the applet, the server generates a hash (using the private key) of the data sent to the applet. The hash is sent in the server response together with the algorithm used for the hash generation.

On the applet side, before processing a server response, it validates the hash provided by the server using the public key which is embedded as a resource in the applet. An exception is thrown in case the hash coming from the server cannot be validated.

All communication from the Signing Controller to the applet is protected this way (that is, the localised messages, the signing context and the sealed PDF).

### Server Seal

A common feature of the use cases is that the eJustice portal server “seals” the PDF before delivering it to the applet or to the user to sign. The purpose is to ensure that the PDF has not been edited by the user.

The server can be configured in order to use one of three possible sealing methods:

* Custom seal;
* PAdES seal;
* No seal.

#### Custom Seal

The implementation of the feature is:

1. The SigningController obtains the PDF representation of the dynamic form from the eJustice portal;
2. The SigningController generates a SHA-1 hash of the PDF;
3. The SigningController digitally signs the digest using a signing certificate;
4. The SigningController stores the signed digest (base 64 encoded) in a text file and attaches the text file to the PDF (a standard PDF attachment)

In order to validate the seal when a signed document is received from the applet or user, the signed content is extracted, then the digest attachment is removed from the PDF, then the digest is re-calculated and compared with the digest that was received.

This approach to sealing the PDF is used in preference to simply signing it in order to avoid problems with signature validation by the ultimate user of the PDF. However, it is not guaranteed to be compatible with third party signing tools which can change the internal structure of the PDF during signature,

#### PAdES seal

The implementation of the feature is:

1. The SigningController obtains the PDF representation of the dynamic form from the eJustice portal;
2. The SigningController signs the document by creating a PADES signature using a signing certificate coming from a PKCS12 certificate store.

This approach to sealing the PDF is compatible with third party signing tools because the sealed part of the PDF is protected from modification during signature. However, the fact that the final document is signed twice could be problematic for the end processor, especially as the signature associated with the seal cannot be a qualified electronic signature.

#### No seal

The server does not seal the PDF. As a consequence, the server cannot validate the origin of the document.

This is the least secure method among the three options. In this approach, there is not guarantee that the user has signed the same PDF as the one he/she downloaded.

# Deployment View

The following diagram represents a deployment view of the signing service. The diagram shows the hardware and software nodes in the execution environment of the system that are relevant to the signing service.



Figure 4: Signing Service - Deployment View

The Signing Controller must have external access by HTTP(S) to OCSP, CRL, TSP, LOTL and MS Trusted list servers. The URL of the OCSP, CRL, TSP, and LOTL are fixed within the application and should be stable over the long term; the URLs of the MS Trusted list servers are taken from the LOTL and are subject to change without notice.

The Signing Applet is deployed in the browser on the user workstation. The browser must have HTTPS access to the enterprise application.

# Implementation View

The following figure is a component view of the enterprise application showing the software layers relevant to the signing service.



Figure 5: Signing Service - Implementation View

The signing controller component depicted in Figure 5 is the component specified in section “5.3.3 Signing Controller”. It is provided as a separate Java Archive (.jar). A small number of classes and configuration files that link the presentation layer to the Signing Controller must be deployed in the enterprise application Web Archive (.war).

BouncyCastle is a library of cryptography functions. PDFBox is a library that allows manipulation of the content of a PDF document.

The figure emphasises that the Signing Applet is a rather heavyweight component that relies on the substantial BouncyCastle and PdfBox libraries. The Signing Applet consists of a core component that can execute within a JRE version 1.5 or higher. A jar compiled for JRE version 1.6 provides the Signing Applet with additional capabilities when a JRE version 1.6 or higher is available[[3]](#footnote-4). The Smart Card Drivers component represents the middleware needed for the Applet to communicate with the smart card. These must be pre-installed on the user’s computer. The MSCAPI DLLs component are two dlls required by the improved MSCAPI connector (refer to section 5.2.3.2) – these are the improved mscapi.dll and a c++ runtime that is required for JRE version 1.6.

On the server side, the DSS library is used to seal the PDF and to validate the signed PDF.

The Presentation Layer Artifacts are the JSP, JavaScript, CSS, and images for the presentation layer. These can be customised to suit the enterprise application.

# Data View

## Card Profile

The following diagram shows the data view of the signing service card profile management. This is relevant only to the smart card signing method.



Figure 6: Signing Service - Data View – Card Profile Repository

### Entities

* The **Card\_Profile** entity represents a type of smart card, identified by its ATR. The “Name” attribute is a descriptive name and the “url” attribute is the web page of the provider or manufacturer of the card.
* The **API** entity represents an API that is supported by the **Card\_Profile**. This also depends on the operating system and architecture. The **API** includes an optional library path attribute which contains the common location(s) of the driver needed by the API.
* The **Digest Algorithm** entity represents an algorithm used to create the digest of a PDF (for example, SHA-1). The name attribute corresponds to the name of the algorithm as used by the DSS library. The **Digest Algorithm** is associated with the **Card\_Profile**, but also with the **API**. This allows a default **Digest Algorithm** to be defined for the card that can be overridden, in certain cases, at the level of the **API**.
* The **Card\_Profile** is associated with the strongest digest algorithm that it supports.

### Choice of Algorithm

A smart card can support several digest algorithms. This implies that a choice must be made when creating the digital signature. There are several scenarios:

1. The card profile is known. In this case the signing applet will take the strongest digest algorithms. This shall be defined in the card profile data store.
2. The card profile is not known. In this case the signing applet will attempt to use a default configuration defined in the card profile data store. There are two possible strategies:
   1. List strategy

The card profile data store contains an ordered list of algorithms that should be tried until the document has been signed or all listed algorithms have been tried.

* 1. Random strategy

The card profile data store contains a list of algorithms and a default algorithm. The algorithms in the list should be tried randomly until the document has been signed, all algorithms have been tried, or number of tries reaches the maximum permitted number (also defined in the card profile data store).

Each attempt to sign the document will be logged and will allow the capabilities of the card to be learned.

1. The PKS12 API is used. In this case the signing applet will use the SHA-512 digest algorithm, which is the strongest digest algorithm supported by the DSS library.

## Statistics Logging

The following diagram shows the data view of the signing service statistics logging.



Figure 7: Signing Service - Data View – Card Profile Repository

The SigningContextEvent and AppletCloseEvent, as well as the Fingerprint entity, are relevant only to the Smart Card signing method. SignatureEvent statistics are logged for all signing methods but the libraryPath, cardIssuer, digestAlgo and signatureAlgo attributes are only relevant to the Smart Card signing method.

### Entities

The following entities are involved in statistics logging.

* The DssEvent is an abstract class containing the relevant date for the event and the fingerprint of the applet environment.
* The SignatureEvent is logged whenever the user attempts to sign the document. It contains the parameters of the signature, indicates if the signature was successful, and provides error codes in the event of an unsuccessful signature;
* The AppletCloseEvent is logged whenever the applet closes. It contains an event log recording the operations that were performed by the applet.

# Size and Performance

The main dimensioning characteristic that impacts the architecture is the size of the applet. This must be kept as small as possible in order to reduce the download time for the user. The architectural choice to use the applet to construct the signed PDF inevitably increases its size, but provides more security to the enterprise by making the client computer wholly responsible for signing the PDF.

To minimise the applet size, it is structured such that features needed only for particular environments, such as API wrappers or dlls, can be lazily loaded.

# Detailed Design

This chapter provides details about the design of the signing service. It describes how the functional analysis is translated and structured in the architecture.

Table 8 lists the components that are discussed.

|  |  |
| --- | --- |
| Component | Description |
| Signing Applet | A java applet that is used to collect information about the user’s environment, prepare the user interface in the containing web page, interact with a signing device, obtain the PDF document to sign, and sign the PDF document. |
| Signing Controller | A server-side component deployed in the enterprise application that supports the operation of the Signing Applet. |
| Enterprise Application | The Enterprise Application uses the Signing Service, but also provides services to the Signing Service |

Table 8: Signing Service Components

The Signing Service consists of the Signing Applet and the Signing Controller; the Signing Service also has a small presence in the Enterprise Application as the JSP and supporting logic.

The Signing Service is built as several Maven artifacts under the eu.europa.ejusticeportal.dss group.

| Artifact ID | Description |
| --- | --- |
| digital-signature-applet | The parent of the applet-dependencies and portal-signing artifacts. |
| applet-dependencies | This parent of Signing Applet dependencies. The modules exist in order to sign the jar files automatically during the maven build, and in some cases to strip the jar files of unnecessary classes in order to reduce the size of the applet. |
| apache-commons | Some classes from the Apache commons projects, bundled in a single jar file |
| bcmail-jdk15on | Bouncycastle singnature and encryption libraries |
| bcpkix-jdk15on |
| bcprov-jdk15on |
| dss-common | Core DSS Library |
| dss-document |
| dss-spi |
| pdfbox | The pdfbox library for PDF manipulation |
| slf4j | Slf4j logging |
| xmlsec | XML security standards |
| sscd-mocca-adapter | The adapter between the DSS library and the MOCCA framework |
| Smcc | Required jar from the MOCCA framework |
| iaik | JCE implementation use by the MOCCA framework |
| portal-signing | The parent of the artifacts for the Signing Applet and Signing Controller |
| ej-portal-dss-common | Classes defining the objects exchanged between the Signing Applet and the Signing Controller and associated utilities |
| ej-portal-dss-applet | Core classes of the Signing Applet |
| ej-portal-dss-applet-6 | Classes of the Signing Applet relying on the javax.smartcardio package in JRE 6, including the MOCCA API classes |
| ej-portal-dss-controller | Classes of the Signing Controller |
| ej-portal-dss-mscapi | The classes and the dll for the improved MSCAPI connector |
| ej-portal-dss-mscapi-crt | The c++ runtime needed by JRE 1.6 when the improved MSCAPI connector is used (specifically, msvcr100.dll). |
| ej-portal-dss-mocca | The classes that interface with the MOCCA API |
| signature-page-demo | A simple web application that demonstrates how the Signing Applet and Signing Controller can be deployed. |

Table 9: Signing Service Artifacts

## Signing Applet

### Environment Detection

The Deployment Toolkit Script, a JavaScript provided by Oracle[[4]](#footnote-5), detects if the java plug-in is present in the browser and if it is at or above the minimum supported version, and launches an applet.

The applet collects information about the execution environment (operating system, architecture – whether 64 or 32 bit – Java JRE version and provider by requesting the standard properties from the java.lang.System class); this is supplemented with the User-Agent and platform provided by the browser as applet parameters.

If a smartcard is present, the applet can probe the card to obtain the card type. The card type can be derived from the content of the ATR (answer to reset) of the smart card that may be accessed by the getATR() method of the javax.smartcardio.Card class. The ATR is used as a unique identifier for the card type. This method to get the ATR exists only since JRE 1.6. The fall back for earlier JRE versions is to obtain more information from the user.

### Applet – Signing Controller Communication

Figure 8 illustrates the basis of communication between the applet and the server.



Figure 8: Applet-Webapp communication - principle

Using the JSObject[[5]](#footnote-6), the applet gets a reference to an object that represents the containing web page. This object allows the applet to manipulate the DOM and invoke JavaScript functions in the page.

To make a call to the server, the applet makes a call to a JavaScript function in the containing page. This JavaScript function in turn makes an asynchronous call to the server using the JQuery Ajax API[[6]](#footnote-7). This Ajax call provides a call-back function that is invoked when the response from the server is received. This function passes the result back to the applet by invoking a call-back method on the applet.

### User Interface

The applet does not provide a direct user interface but rather relies on the containing HTML page.

#### State and Event Management

The applet is responsible for managing the state of the user interface. For example, it determines which user interface components should be visible or enabled and what text they should contain.

The applet maintains a model of the user interface in the eu.europa.ejusticeportal.dss.applet.ui.UI class. This consists of a list of components having typical attributes (visible, enabled, text, etc.) and a unique id (taken from the user interface design in [FS]). The model is updated whenever some processing done in the applet requires a change to the user interface. The applet serialises the changes to the model as a JSON string and invokes the updateUI JavaScript function on the containing web page, providing the JSON string as a parameter; this JavaScript function then manipulates the HTML components accordingly.

Events originating on the web page (for example, selection of a certificate, request to view the PDF) are identified by a unique type id and channelled back to the applet through the handleUiEvent method.

#### Internationalisation

Internationalisation of the components in the web page relies on code-value-language tuples.

### Signature Management

The Signing Applet uses the DSS library[[7]](#footnote-8) for all signature operations. The DSS library is officially supported and compiled for JRE 6.

## Signing Controller

### Configuration

The signing controller is configured by means of an XML file compliant to an XML schema, depicted in Figure 9.

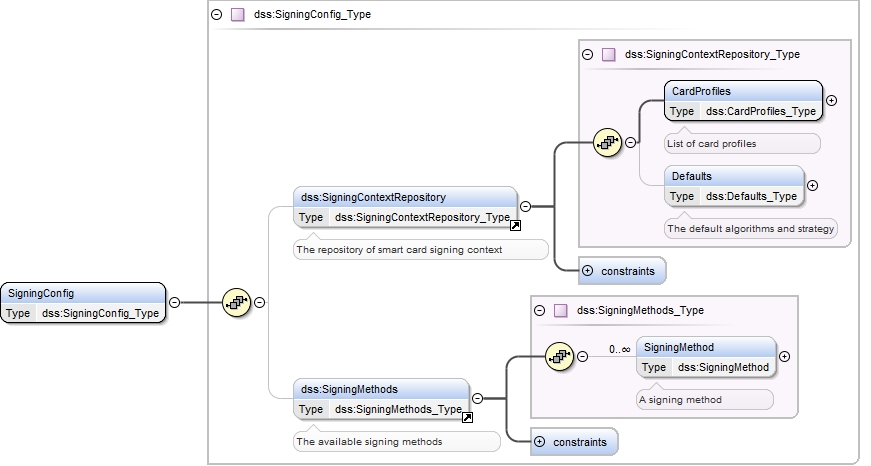


Figure 9: Signing Controller – Configuration – XML Schema – Top Level

The configuration consists of two main parts – the definition of the supported signing methods and the definition of the explicitly supported smart cards.

### Smart Card Signing Methods

Figure 10 depicts the configuration of the available signing methods.

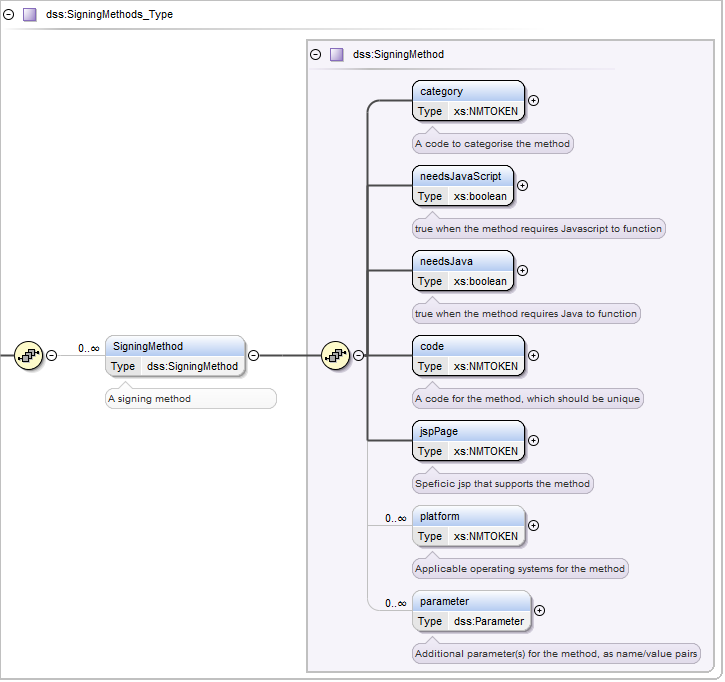


Figure 10: Signing Controller – Configuration – XML Schema – Signing Methods

The configuration includes Boolean flags to indicate if the method requires Java or JavaScript. Each method has a unique code

|  |  |
| --- | --- |
| Component | Description |
| needsJavaScript | A Boolean flag that indicates if the signing method requires JavaScript. Based on this value, the method can be disabled on the user interface. |
| needsJava | A Boolean flag that indicates if the signing method requires Java. Based on this value, a warning can be added in the user interface. |
| code | A unique code for the signing method. This code can be used for identifying translations for the method as well as in the logic of the controller. |
| jspPage | The jsp page that supports the signing method. |
| platform | A list of strings that identify platforms supported by the signing method. The logic for identifying a supported platform is based on partial matching of the User-Agent string from the HTTP request header.  If the attribute is empty, all platforms are assumed to be supported. |
| parameter | A list of name/value parameters that are available in the JSP. |
| category | Allows to group methods in categories, for better presentation on the user interface |

Table 10: Signing Controller – Configuration – XML Schema – Signing Methods

### Smart Card Profile Management

The profiles of smart cards explicitly supported by the Signing Service are held in a data store, implemented as an XML file. The Signing Controller uses information about the user’s environment provided by the Signing Applet to obtain the signing context from the data store. The signing context enables the applet to permit the user to sign the document with minimal effort.

This XML file also contains the default algorithms and strategy for use when the presented smart card is unknown.

Figure 11 depicts the part of the XML schema definition related to the smart card profile data store.

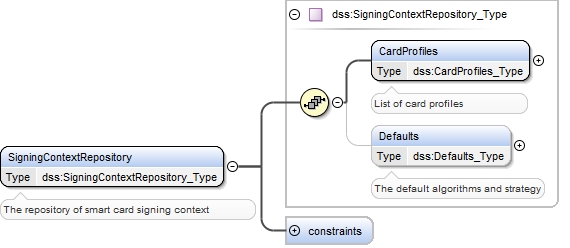


Figure 11: Smart card profile data store XML schema – top level

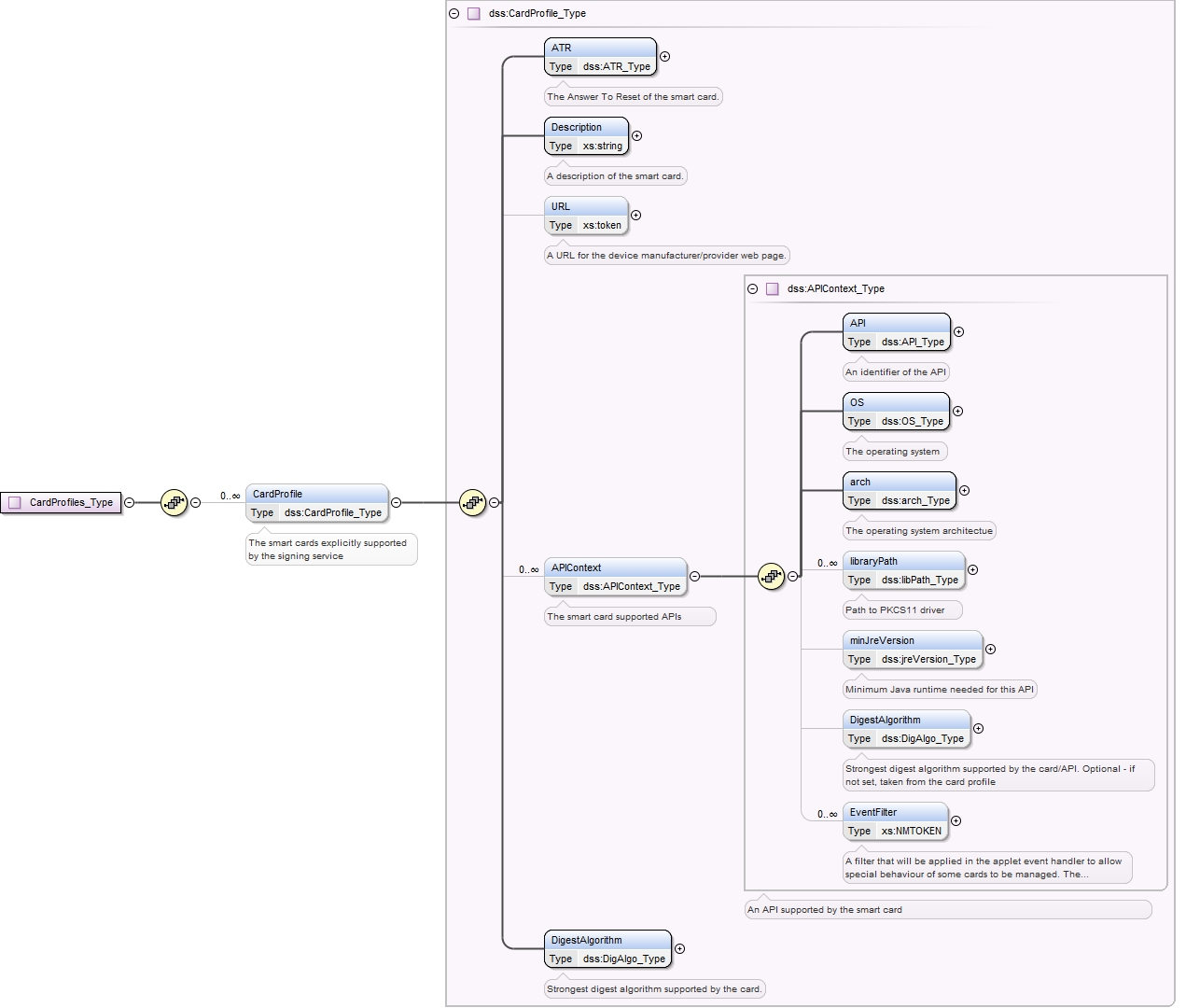


Figure 12: Smart card profile data store XML schema – Card Profiles

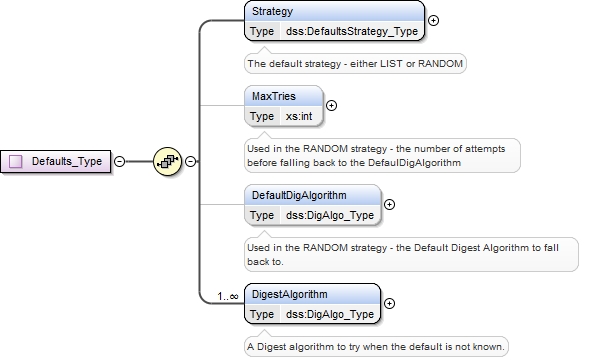


Figure 13: Smart card profile data store XML schema – Defaults

### Document Management

The Signing Controller prepares the PDF document to download to the user by sealing it according to the description in section 5.3.4.

The Signing Controller is also responsible for performing some very basic validation of the document uploaded by user. Depending on the configuration, the Signing Controller validates that the server signature applied to the PDF is intact, and validates that the PDF has later been digitally signed by another entity. It verifies the digital signature. Validation is completely configurable to give a blocking error, a warning or to be disabled.

The Signing Controller uses the DSS library[[8]](#footnote-9) for all signature operations.

### Enterprise Application Interface

The Signing Controller contains the eu.europa.ejusticeportal.dss.controller.PortalFacade interface, defining the services provided by the enterprise application to the Signing Controller, as detailed in Table 7.

## Enterprise Application

The Signing Service relies on a small number of artifacts in the enterprise application, specifically

* One main JSP providing the user interface (this JSP is composed of several JSP includes, depending on the chosen signing method) and associated style sheets and scripts;
* An implementation of the eu.europa.ejusticeportal.dss.controller.PortalFacade interface (refer to section 10.2), exposing enterprise application services to the Signing Controller.
* A “stub” controller to channel the HTTP requests to the Signing Controller, and to deliver the response back to the applet.

# Appendix

## Smart Card Profile XSD

The following box contains the XSD for the smart card profile data store.

<?xml version=**"1.0"** encoding=**"UTF-8"**?>

<xs:schema xmlns:xs=**"http://www.w3.org/2001/XMLSchema"** xmlns:dss=**"eu:europa:ejusticeportal:dynforms:signing:profile:v1"** targetNamespace=**"eu:europa:ejusticeportal:dynforms:signing:profile:v1"** elementFormDefault=**"qualified"** attributeFormDefault=**"unqualified"** version=**"1.0"**>

<xs:element name=**"SigningConfig"** type=**"dss:SigningConfig\_Type"**/>

<xs:complexType name=**"SigningConfig\_Type"**>

<xs:sequence>

<xs:element ref=**"dss:SigningContextRepository"** minOccurs=**"0"**/>

<xs:element ref=**"dss:SigningMethods"** minOccurs=**"0"**/>

</xs:sequence>

</xs:complexType>

<xs:element name=**"SigningContextRepository"** type=**"dss:SigningContextRepository\_Type"**>

<xs:annotation>

<xs:documentation>**The repository of smart card signing context**</xs:documentation>

</xs:annotation>

<xs:unique name=**"atr\_unique"**>

<xs:selector xpath=**"dss:CardProfiles/dss:CardProfile/dss:ATR"**/>

<xs:field xpath=**"."**/>

</xs:unique>

</xs:element>

<xs:complexType name=**"SigningContextRepository\_Type"**>

<xs:sequence>

<xs:element name=**"CardProfiles"** type=**"dss:CardProfiles\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**List of card profiles**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"Defaults"** type=**"dss:Defaults\_Type"** minOccurs=**"0"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The default algorithms and strategy**</xs:documentation>

</xs:annotation>

</xs:element>

</xs:sequence>

</xs:complexType>

<xs:complexType name=**"Defaults\_Type"**>

<xs:sequence>

<xs:element name=**"Strategy"** type=**"dss:DefaultsStrategy\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The default strategy - either LIST or RANDOM**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"MaxTries"** type=**"xs:int"** minOccurs=**"0"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Used in the RANDOM strategy - the number of attempts before falling back to the DefaulDigAlgorithm**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"DefaultDigAlgorithm"** type=**"dss:DigAlgo\_Type"** minOccurs=**"0"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Used in the RANDOM strategy - the Default Digest Algorithm to fall back to.**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"DigestAlgorithm"** type=**"dss:DigAlgo\_Type"** maxOccurs=**"unbounded"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**A Digest algorithm to try when the default is not known.**</xs:documentation>

</xs:annotation>

</xs:element>

</xs:sequence>

</xs:complexType>

<xs:complexType name=**"CardProfiles\_Type"**>

<xs:sequence>

<xs:element name=**"CardProfile"** type=**"dss:CardProfile\_Type"** minOccurs=**"0"** maxOccurs=**"unbounded"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The smart cards explicitly supported by the signing service**</xs:documentation>

</xs:annotation>

</xs:element>

</xs:sequence>

</xs:complexType>

<xs:complexType name=**"CardProfile\_Type"**>

<xs:sequence>

<xs:element name=**"ATR"** type=**"dss:ATR\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The Answer To Reset of the smart card.**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"Description"** type=**"xs:string"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**A description of the smart card.**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"URL"** type=**"xs:token"** minOccurs=**"0"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**A URL for the device manufacturer/provider web page.**

</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"APIContext"** type=**"dss:APIContext\_Type"** minOccurs=**"0"** maxOccurs=**"unbounded"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The smart card supported APIs**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"DigestAlgorithm"** type=**"dss:DigAlgo\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Strongest digest algorithm supported by the card.**</xs:documentation>

</xs:annotation>

</xs:element>

</xs:sequence>

</xs:complexType>

<xs:complexType name=**"APIContext\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>

**An API supported by the smart card**

</xs:documentation>

</xs:annotation>

<xs:sequence>

<xs:element name=**"API"** type=**"dss:API\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**An identifier of the API**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"OS"** type=**"dss:OS\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The operating system**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"arch"** type=**"dss:arch\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The operating system architectue**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"libraryPath"** type=**"dss:libPath\_Type"** minOccurs=**"0"** maxOccurs=**"unbounded"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Path to PKCS11 driver**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"minJreVersion"** type=**"dss:jreVersion\_Type"** minOccurs=**"0"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Minimum Java runtime needed for this API**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"DigestAlgorithm"** type=**"dss:DigAlgo\_Type"** minOccurs=**"0"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Strongest digest algorithm supported by the card/API. Optional - if not set, taken from the card profile**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"EventFilter"** type=**"xs:NMTOKEN"** minOccurs=**"0"** maxOccurs=**"unbounded"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**A filter that will be applied in the applet event handler to allow special behaviour of some cards to be managed.**

**The value must be a fully qualified Java class name implementing eu.europa.ejusticeportal.dss.applet.event\_api.EventFilter**</xs:documentation>

</xs:annotation>

</xs:element>

</xs:sequence>

</xs:complexType>

<xs:simpleType name=**"ATR\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The ATR (answer to reset), in hexadecimal form**

</xs:documentation>

</xs:annotation>

<xs:restriction base=**"xs:string"**>

<xs:pattern value=**"(((([0-9]|[A-F]){2})|(\.\.)|(\.([0-9]|[A-F]))|(([0-9]|[A-F])\.)|(\[.,.\]([0-9]|[A-F]))|(([0-9]|[A-F])\[.,.\])) ){0,31}((([0-9]|[A-F]){2})|(\.\.)|(\.([0-9]|[A-F]))|(([0-9]|[A-F])\.)|(\[.,.\]([0-9]|[A-F]))|(([0-9]|[A-F])\[.,.\]))"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The ATR is given as a regular expression of hexadecimal encoded bytes separated by spaces. Some positions in the string can take a range of values, for example: 1E 2F 24 .. 3D [1,5]F ..**</xs:documentation>

</xs:annotation>

</xs:pattern>

</xs:restriction>

</xs:simpleType>

<xs:simpleType name=**"DigAlgo\_Type"**>

<xs:restriction base=**"xs:string"**>

<xs:enumeration value=**"SHA1"**/>

<xs:enumeration value=**"SHA256"**/>

<xs:enumeration value=**"SHA384"**/>

<xs:enumeration value=**"SHA512"**/>

<xs:enumeration value=**"MD2"**/>

<xs:enumeration value=**"MD5"**/>

</xs:restriction>

</xs:simpleType>

<xs:simpleType name=**"API\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The unique identifier of the API**

</xs:documentation>

</xs:annotation>

<xs:restriction base=**"xs:string"**>

<xs:enumeration value=**"PKCS11"**/>

<xs:enumeration value=**"PKCS12"**/>

<xs:enumeration value=**"MSCAPI"**/>

<xs:enumeration value=**"MOCCA"**/>

</xs:restriction>

</xs:simpleType>

<xs:simpleType name=**"OS\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The operating system type.**

</xs:documentation>

</xs:annotation>

<xs:restriction base=**"xs:string"**>

<xs:enumeration value=**"windows"**/>

<xs:enumeration value=**"linux"**/>

<xs:enumeration value=**"macos"**/>

</xs:restriction>

</xs:simpleType>

<xs:simpleType name=**"CertElement\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Element of a certificate used in a hint.**

</xs:documentation>

</xs:annotation>

<xs:restriction base=**"xs:string"**>

<xs:enumeration value=**"issuer"**>

<xs:annotation><xs:documentation xml:lang=**"en"**>**The issuer of the certificate**</xs:documentation></xs:annotation>

</xs:enumeration>

<xs:enumeration value=**"crl"**>

<xs:annotation><xs:documentation xml:lang=**"en"**>**The CRL URL**</xs:documentation></xs:annotation>

</xs:enumeration>

</xs:restriction>

</xs:simpleType>

<xs:simpleType name=**"arch\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The architecture type (bitness)**

</xs:documentation>

</xs:annotation>

<xs:restriction base=**"xs:string"**>

<xs:enumeration value=**"32"**/>

<xs:enumeration value=**"64"**/>

<xs:enumeration value=**"either"**/>

</xs:restriction>

</xs:simpleType>

<xs:simpleType name=**"jreVersion\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The Java runtime environment version**

</xs:documentation>

</xs:annotation>

<xs:restriction base=**"xs:string"**>

<xs:pattern value=**"[0-9]{1,3}(\.[0-9]{1,2})(\.[0-9]{1,2}){0,1}(\.[0-9]{1,2}){0,1}"**/>

</xs:restriction>

</xs:simpleType>

<xs:simpleType name=**"libPath\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The path to PKCS11 library. Use "/" as path separator.**

</xs:documentation>

</xs:annotation>

<xs:restriction base=**"xs:token"**>

<xs:pattern value=**"[^\\]+"**/>

</xs:restriction>

</xs:simpleType>

<xs:simpleType name=**"DefaultsStrategy\_Type"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The defaults strategy decides which mechanism to use when deciding which algorithms to choose when the card profile is not known**</xs:documentation>

</xs:annotation>

<xs:restriction base=**"xs:NMTOKEN"**>

<xs:enumeration value=**"LIST"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The LIST strategy will try all algorithms in the list until one works or until the list is exhausted.**</xs:documentation>

</xs:annotation>

</xs:enumeration>

<xs:enumeration value=**"RANDOM"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**The RANDOM strategy will try a given number of algorithms before attempting the default.**</xs:documentation>

</xs:annotation>

</xs:enumeration>

</xs:restriction>

</xs:simpleType>

<xs:element name=**"SigningMethods"** type=**"dss:SigningMethods\_Type"**>

<xs:annotation>

<xs:documentation>**The available signing methods**</xs:documentation>

</xs:annotation>

<xs:unique name=**"code\_unique"**>

<xs:selector xpath=**"dss:SigningMethods/dss:SigningMethod/dss:Code"**/>

<xs:field xpath=**"."**/>

</xs:unique>

</xs:element>

<xs:complexType name=**"SigningMethods\_Type"**>

<xs:sequence>

<xs:element name=**"SigningMethod"** type=**"dss:SigningMethod"** minOccurs=**"0"** maxOccurs=**"unbounded"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**A signing method**</xs:documentation>

</xs:annotation>

</xs:element>

</xs:sequence>

</xs:complexType>

<xs:complexType name=**"SigningMethod"**>

<xs:sequence>

<xs:element name=**"category"** type=**"xs:NMTOKEN"** minOccurs=**"1"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**A code to categorise the method**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"needsJavaScript"** type=**"xs:boolean"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**true when the method requires Javascript to function**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"needsJava"** type=**"xs:boolean"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**true when the method requires Java to function**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"code"** type=**"xs:NMTOKEN"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**A code for the method, which should be unique**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"jspPage"** type=**"xs:NMTOKEN"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Speficic jsp that supports the method**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"platform"** type=**"xs:NMTOKEN"** minOccurs=**"0"** maxOccurs=**"unbounded"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Applicable operating systems for the method**</xs:documentation>

</xs:annotation>

</xs:element>

<xs:element name=**"parameter"** type = **"dss:Parameter"** minOccurs=**"0"** maxOccurs=**"unbounded"**>

<xs:annotation>

<xs:documentation xml:lang=**"en"**>**Additional parameter(s) for the method, as name/value pairs**</xs:documentation>

</xs:annotation>

</xs:element>

</xs:sequence>

</xs:complexType>

<xs:complexType name=**"Parameter"**>

<xs:annotation><xs:documentation xml:lang=**"en"**>**A parameter for a SigningMethod**

</xs:documentation></xs:annotation>

<xs:sequence>

<xs:element name=**"key"** type=**"xs:NMTOKEN"**/>

<xs:element name=**"value"** type=**"xs:string"**/>

</xs:sequence>

</xs:complexType>

</xs:schema>

*End of Document*

1. The data file could be an XML representation of the form data – this is an optional feature. [↑](#footnote-ref-2)
2. The browser is omitted for simplicity. [↑](#footnote-ref-3)
3. This split is for historical reasons – JRE version 1.5 is no longer officially supported. [↑](#footnote-ref-4)
4. [http://docs.oracle.com/javase/6/docs/technotes/guides/jweb/deployment\_advice.html#deplToolkit](http://docs.oracle.com/javase/6/docs/technotes/guides/jweb/deployment_advice.html%23deplToolkit) [↑](#footnote-ref-5)
5. <http://docs.oracle.com/javase/6/docs/technotes/guides/plugin/developer_guide/java_js.html> [↑](#footnote-ref-6)
6. <http://api.jquery.com/jQuery.ajax/> [↑](#footnote-ref-7)
7. <http://joinup.ec.europa.eu/software/sd-dss/home> [↑](#footnote-ref-8)
8. <http://joinup.ec.europa.eu/software/sd-dss/home> [↑](#footnote-ref-9)