



Ascertia Limited
40 Occam Road
Guildford
Surrey
GU2 7YG
United Kingdom

info@ascertia.com

www.ascertia.com

ADSS Server Developer's Guide

Document Version: 4.1.0.1

Document Issued: March 2010

©Copyright Ascertia Ltd, 2010

This document contains commercial-in-confidence material. It must not be disclosed to any third party without the written authority of Ascertia Limited.

Contents

1	Introduction	4
1.1	Scope	4
1.2	Intended Readership	4
1.3	Conventions	4
1.4	Technical support	4
1.5	Glossary	5
1.6	References to PKI Standards	6
2	ADSS Server Overview	7
2.1	Modes of Operation	7
2.2	ADSS Server Architecture	8
2.2.1	ADSS Enterprise Server Architecture	8
2.2.2	ADSS Infrastructure Server Architecture	9
2.3	ADSS Server Application Programming Interface	9
2.4	Interfacing to the OCSP and Timestamp Services	10
2.5	Essential Reading	10
3	Integration with a Business application	12
3.1	Generating/Registering a Key Pair and Certificate	12
3.2	Changing an end-user key Authorisation Code	13
3.3	Renewing a Key pair and Certificate	13
3.4	Deleting a Key pair and Certificate	14
3.5	Retrieving Private Key (PKCS#12 object) and Certificate	14
3.6	Server-side Document Signing	14
3.7	Client-Side Document Signing	15
3.7.1	Signing Documents with the GoSign Applet	15
3.7.2	Signing Documents with GoSign.NET	18
3.7.3	Hashing and Assembly	19
3.8	Verifying Signed Documents	19
3.9	Creating an Empty Signature Field on PDF Documents	20
3.10	XKMS Validate Service	21
3.11	LTANS Service	21
3.12	Decryption Service	22
4	Developing ADSS Web Service Clients	23
4.1	Using the ADSS Java API	25
4.1.1	Server-side Signing	25
4.1.2	Client-side Signing	26
4.1.3	Creating Empty Signature Fields	27
4.1.4	Verification	28
4.1.5	Certification	29
4.1.6	XKMS Validate	30
4.1.7	LTANS Archive	31
4.1.8	LTANS Export	31
4.1.9	LTANS Delete	32
4.1.10	Decryption	32
4.2	Using the .NET API	33
4.2.1	Server-side Signing	33
4.2.2	Client-side Signing	34
4.2.3	Creating Empty Signature Fields	36
4.2.4	Verification	36

4.2.5	Certification	37
4.2.6	XKMS Validate.....	38
4.2.7	LTANS Archive	39
4.2.8	LTANS Export.....	40
4.2.9	LTANS Delete.....	40
4.2.10	Decryption.....	41
4.3	Code Samples	41
5	ADSS Server Web Services XML Schemas	45
5.1	Verification Web Service.....	45
5.1.1	Request header for signature verification and certificate validation.....	47
5.1.2	Request body for signature verification	48
5.1.3	Request body for certificate validation.....	51
5.1.4	Response header for signature verification and certificate validation	53
5.1.5	Response body for signature verification and certificate validation	54
5.2	Signing Web Service	57
5.2.1	Document Signing Request.....	58
5.2.2	Document Signing Response	60
5.3	Certification Service.....	62
5.3.1	Certification request.....	62
5.3.2	Certification Response.....	64
5.4	Hashing Service.....	65
5.4.1	Hashing Request	66
5.4.2	Hashing Response Element	67
5.5	Assembly Service	68
5.5.1	Assembly Request.....	68
5.5.2	Assembly Response	69
5.6	Empty Signature Field Creation Service.....	69
5.6.1	Empty signature field generation request.....	70
5.6.2	Empty Signature Field Response	72
5.7	XKMS Service.....	72
5.8	LTAN Service.....	72
5.9	Decryption Service.....	72
5.10	Schema elements not currently supported in ADSS Server.....	72
5.11	Supported Algorithms in ADSS Server.....	75

1 Introduction

1.1 Scope

This document provides information on the Advanced Digital Signature Services (ADSS) Web Services interfaces and how business application developers can integrate these trust services into their applications. It also provides simple use cases on how an example Business Logic Application (BLA) should interact with ADSS Server.

1.2 Intended Readership

This guide is intended for developers who are interested in writing applications which will use the web services offered by ADSS Server. It is assumed that the reader has a basic knowledge of digital signatures, certificates and IT security.

1.3 Conventions

The following typographical conventions are used in this guide to help locate and identify information:

Bold text identifies menu names, menu options, items you can click on the screen, file names, folder names, and keyboard keys.

Courier font identifies code and text that appears on the command line.

Bold courier identifies commands that you are required to type in.

1.4 Technical support

If Technical Support is required, Ascertia has a dedicated support team providing debugging assistance, integration assistance and general customer support. Ascertia Support can be accessed in the following ways:

Support Email	support@ascertia.com
Support MSN Messenger	support@ascertia.com

In addition to the free support service describe above, Ascertia provides formal support agreements with all product sales. Please contact sales@ascertia.com for more details.

A Product Support Questionnaire should be completed to provide Ascertia Support with further information about your system environment. When requesting help it is always important to confirm:

- System Platform details;
- ADSS Server version number and build date;
- Details of specific issue and the relevant steps taken to reproduce it;
- Database version and patch level;
- The product log files.

1.5 Glossary

ADSS Server	Advanced Digital Signature Services (a server-side product from Ascertia for providing signature generation/verification, certificate validation and other e-trust services)
CA	Certificate Authority (logical entity responsible for issuing certificates and optionally also CRLs)
CAPI	Microsoft Crypto API
Cert	Digital Certificate
CRL	Certificate Revocation Lists
CMS	Cryptographic Message Syntax (a digital signature format)
DBMS	Database Management System
DNV	Det Norske Veritas (an Ascertia partner providing signature verification services using ADSS Server)
DSA	Digital Signature Algorithm
HSM	Hardware/Host Security Module
HTTP	Hyper Text Transfer Protocol
HTTP/S	HTTP over SSL/TLS connection
JDBC	Java Database Connectivity
IETF	Internet Engineering Task Force
LDAP	Lightweight Directory Access Protocol
LDAP/S	LDAP over SSL/TLS connection
PKCS	Public Key Cryptographic Standards
PKI	Public Key Infrastructure
RSA	Rivest, Shamir, Adleman (public key algorithm)
OCSP	Online Certificate Status Protocol (an IETF protocol for verifying the revocation status of a digital certificate)
SCVP	Server-based Certificate Validation Protocol
SHA	Secure Hash Algorithm (various different algorithms, e.g. SHA-1, SHA-256, SHA-512 etc.)
S/MIME	Secure MIME (standard for signing emails)
SSL	Secure Sockets Layer
TA	Trust Authority (authority trusted for issuing certificates, CRLs, OCSP responses and/or time stamps)
TLS	Transport Layer Security (a later version of SSL)
TSA	Time Stamp Authority (authority responsible for issuing timestamp tokens to prove that a document/data existed at a particular time)
TSP	Time Stamp Protocol
XKMS	XML Key Management Specifications
XML DigSig	XML Digital Signature standard

1.6 References to PKI Standards

CMS	http://tools.ietf.org/html/rfc3852
PDF Signatures	PDF Public Key Digital Signature and Encryption Specification v3.2 http://www.adobe.com/devnet/pdf/pdf_reference.html
PKCS#7	http://www.faqs.org/rfcs/rfc2315.html
S/MIME	http://www.ietf.org/rfc/rfc3851.txt
Timestamp	http://www.ietf.org/rfc/rfc3161.txt
XML DigSig	http://www.ietf.org/rfc/rfc3275.txt
OCSP	http://www.ietf.org/rfc/rfc2560.txt
XKMS	http://www.w3.org/TR/xkms2/
LTANS	http://tools.ietf.org/html/draft-ietf-ltans-ltap-07
OASIS DSS Decryption Profile	http://www.oasis-open.org/committees/download.php/25384/oasis-dss_profile-encryption_A-SIT_v0.1.doc

2 ADSS Server Overview

ADSS Server provides the following trust services:

- An optional Signing Service that supports these features:
 - Supports documents of various formats including PDF, XML, and other files
 - Creating digital signatures of various formats including PDF, XML, PKCS#7 / CMS and S/MIME as well as ETSI XAdES, CAdES and PAdES
 - Assembling signed hash values within PDF documents – useful when a hash of the document has been signed locally using GoSign Applet and it needs to be embedded
 - Creating blank signature fields in PDF documents – useful when using Certify Signatures
 - Working with ADSS GoSign Applet to offer local hash and sign functionality
- An optional Verification Service which supports these features:
 - Supports documents of various formats including PDF, XML, and other files
 - Verifying digital signatures of various formats including PDF, XML, PKCS#7 / CMS and S/MIME as well as ETSI XAdES, CAdES and PAdES
 - Validating X.509 digital certificates
- An optional Certification Service which supports these features:
 - Creating Public key pairs and certifying public keys using either a local CA configured within ADSS Server or external CAs
 - Renewing keys/certificates and changing associated authorisation codes
 - Revoking or suspending previous generated certificates (available via the ADSS Server's CMC interface)
- An optional XKMS Validate which supports these features:
 - Validating a single certificate or a full certificate chain
- An optional LTANS Service which supports these features:
 - Generating and retaining timestamp evidence that shows that data is (or is not) original over the long term
 - Automatically refreshes the timestamp evidence data
 - Optionally keeping and exporting the original data
 - Optionally automatically deleting the retained original data
- An optional Decryption Service which supports these features:
 - Decrypting encrypted documents with server held keys
- An optional Online Certificate Status Protocol (OCSP) service that supports these features:
 - Providing real-time information on the revocation status of a requested certificate, returned certificate status responses are GOOD, REVOKED or UNKNOWN. This service is compliant with the IETF RFC 2560 specifications.
- An optional Time Stamping Authority (TSA) service that supports these features:
 - Time stamping data to independently prove that it existed at (or before) a particular date and time. This is particularly useful for proving the time of signing and an important feature of long-term digital signatures. This service is compliant with the IETF RFC 3161 specifications.

ADSS Server is a J2EE application. It is a secure and scalable product that delivers trust for e-business applications and workflows. ADSS Server includes a flexible policy-based signing engine to suit synchronous and asynchronous business needs. This is discussed further in the **ADSS Server Admin Manual**. Only those services required by the business need to be deployed and licensed.

2.1 Modes of Operation

ADSS Server offers an XML web service interface that exposes the ADSS functionality on-demand to business applications. Using a single SOAP (Simple Object Access Protocol) request message,

business applications can programmatically pass parameters and data relevant to the service request to ADSS Server. The SOAP response message from ADSS Server returns results specific to the type of request it received e.g. if a document signing request is received then the signature or signed document is returned. Using this mode of operation the ADSS server can be tightly interwoven within document workflows.

The ADSS Signing Service also offers an HTTP based API for optimum performance –useful if web service calls are considered to be too much of an overhead for a specific project.

Note Ascertia also provides complete front-end business applications which integrate with ADSS Server in the background. These applications provide an out-of-the-box processing capability and zero integration effort for our customers. The current list of front-end applications for ADSS Server includes:

- **Auto File Processor (AFP):** this is a Watched Folder application which regularly monitors a set of input folders anywhere on the network for documents to be processed by ADSS Server (e.g. to be digitally signed etc.). AFP retrieves these documents and provides to ADSS Server for processing, the returned signed documents are then placed into a designated output (or error) folder. This is an ideal solution for bulk processing of documents in an unattended automated environment.
- **Ascertia Docs:** this is a document collaboration and approval application, ideal for electronic contract execution service. It allows documents to be uploaded and shared with other users for purposes of document review, sign-off and approval. Each collaborator is automatically notified of pending documents requiring their approval, and documents are shown and signed-off through the secure Ascertia GoSign applet viewer working in conjunction with ADSS Server.
- **Secure Email Gateway (SES):** this is a Mail Transfer Agent (MTA) server application that monitors emails as they flow in or out of an organisation. For outgoing emails it can filter emails that require digital signature or archiving and perform this automatically by making calls to ADSS Server. Note both signing of emails and/or attachments is supported. For incoming emails, SES can filter emails which are signed (or contained signed attachments) and can make calls to ADSS Server for automated signature verification services.

The above applications are described separately in their own manuals and not covered further in this document.



Note that OCSP and TSA services are not accessible over SOAP/XML interface rather they are accessible directly over HTTP/S. Consult the protocols specifications (i.e. the RFCs) to know more details about how to communicate with an OCSP or a TSA server. As a result OCSP and TSA Service interface is not discussed in this document as it only covers web services.

2.2 ADSS Server Architecture

As ADSS Server is a multi-function application providing a wide range of trust services, it can often be the case that not all services are required by every business application. Ascertia therefore licenses the ADSS service modules individually. Ascertia also groups the service modules into the following packages:

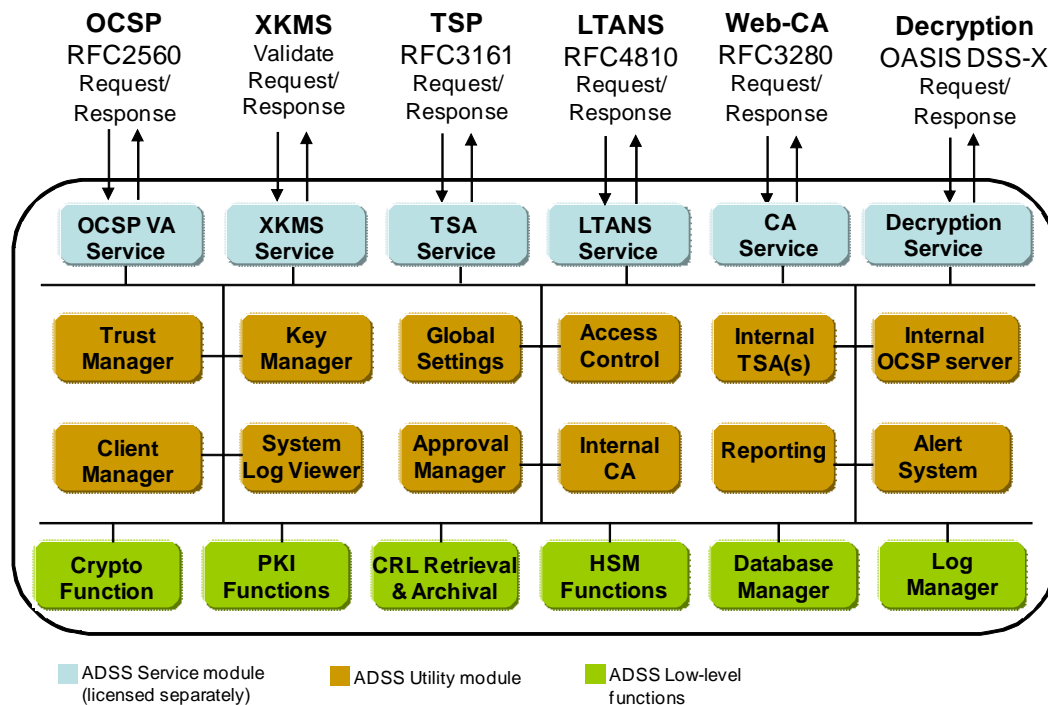
2.2.1 ADSS Enterprise Server Architecture

ADSS Enterprise Server is aimed at organisations wishing to primarily deploy digital signature creation and verification services. It comes with the following modules:

The ADSS Certification Service is also included in this package if it is required to create and certify end-user signing keys. These keys/certificates are then saved on ADSS Server (possibly inside an HSM) for providing server-side digital signature capability.

2.2.2 ADSS Infrastructure Server Architecture

ADSS Infrastructure Server is aimed at organisations wishing to deploy PKI infrastructure components such as CAs, OCSP Responders, TSA etc. The ADSS Infrastructure Server can be provided with the following modules:



2.3 ADSS Server Application Programming Interface

This document provides details on the XML Schema and how to integrate with business applications. Basic use-case examples are provided using a generic business application.

The following table provides a summary of the ADSS Server interfaces:

Service Name	Function
Digital signature service	Creates a digital signature according to a configured signing profile and any allowed application supplied parameters. Can create PDF, XML or PKCS#7 (File / Form) signatures. Various advanced profiles such as CAdES and XAdES are also supported.
Signature field creation	Create a blank signing field within a PDF document
Document/Data hashing	Calculates a hash of the data using a profile defined algorithm. This can be used with the GoSign Applet ¹ for multi-user zero-footprint client side signing (i.e. in this case hashing is done on the server, whilst signing is performed locally on the user's smartcard or USB token)
Document assembly	Embeds a digital signature within a PDF document (E.g. as in the above case, when GoSign applet returns the signed hash value from the user's smartcard, ADSS Server can embed this signature within the corresponding PDF document)
Verification service	Verifies one or more digital signatures. Various signature formats i.e. PDF, PKCS#7/CMS, XML, S/MIME, CAdES and XAdES are supported. As well as validation of digital

¹ The GoSign applet can be used with a centrally calculated hash or a locally calculated hash

Service Name	Function
	certificates
Certification service	Allows keys and certificates to be generated for multiple users which can be used later for server-side signing of documents
XKMS Service	Validates a single certificate or a complete certificate chain. The returned information can be used to create long-term signatures by embedding this validation information within a signature
LTAN Service	Securely archives data or documents for long-term preservation. Each archive object is timestamped to protect its integrity whilst in the archive. This timestamp evidence data can be automatically refreshed at configured timeframes. Archived objects can also be exported and deleted by calling applications.
Decryption Service	Decrypts encrypted document using private keys held by ADSS Server. Can be used in conjunction with GoSign applet, so that end-users can sign and encrypt their document submissions. These encrypted objects can then be decrypted through this web service.

2.4 Interfacing to the OCSP and Timestamp Services

The OCSP and Timestamp services can be accessed by existing external OCSP and TSA clients that request services, over HTTP or HTTPS, according to the RFC protocols detailed in 1.6. The following table describes the basic function of these service interfaces. There is no ADSS API available for these services as such clients are widely available in open source and commercial toolkits.

Service Name	Function
OCSP service	Allows clients to send an OCSP request for checking the revocation status of one or more certificates and in return get a OCSP response which provides "GOOD", "REVOKED" or "UNKNOWN" revocation status for each certificate whose status was requested.
TSA service	Allows clients to send timestamping request on any data object and in return get a signed timestamp response which proves the data object existed at the time provided by the TSA service

2.5 Essential Reading

ADSS Server is a powerful technology offering multiple services and multiple interfaces, to aid the reader we recommend the following sections based on your interests:

- **Server-side signing:** If you plan to use the ADSS Server to apply signature using keys held by ADSS Server (e.g. corporate signing keys or unique end-user keys held by ADSS server) then you should read section 5.2 which explains the request/response protocol items when requesting ADSS Server to sign a document.
- **Client-side signing:** You can use either GoSign applet OR GoSign.Net along with ADSS Server to create signatures at client side. You may need to understand the document hashing and document assembly request/response messaging flow in case you require document hashing to be conducted on the server, alternative option is to send the whole document to the GoSign applet and perform hashing and signing within the applet. To understand in detail the GoSign applet mode of signing separate developer and deployment guides and source code of a demo application is provided with ADSS Client SDK package.

- **Hashing and Assembly:** If you plan to use the ADSS server to hash documents and later provide signature made on the hash for final assembly of the document and signature then you should read section **5.4** and section **5.5** which explains the request/response protocol items.
- **Empty field creation and signing:** If you plan to use the ADSS server to create blank (i.e. empty) signature fields inside PDF documents and also optionally sign these blank signature fields in the same request message then you should read section **5.6** which explains the request/response protocol items.
- **Server-side verification:** If you plan to use the ADSS server to verify documents containing signatures or validate X509 digital certificates then you should read section **5.1** Using ADSS you can verify multiple file types (signed PDFs, signed XMLs, Signed emails in S/MIME format, Certificates, any file signed using PKCS#7 or CMS specification).
- **Certification services:** If you will be using the ADSS server to generate keys/certificate and manage the certification lifecycle we recommend that you read section **5.3** to understand the certification request/response message. This section also contains details of how to change authorisation codes and renewing keys/certificates etc.
- **XKMS Service:** if you plan to use ADSS Server to validate X509 digital certificates using XKMS Validate interface then you should read the section **5.7** of this document. This is a useful mechanism for gathering the revocation info for an entire certificate chain and this can then be embedded within a digital signature as part of the process of creating long-term signatures.
- **LTAN Service:** if you plan to use ADSS Server to archive your data securely as part of a long-term archive service then you should read the section **5.8** of this document
- **Decryption Service:** if you plan to use ADSS Server to decrypt encrypted documents using private keys held on the ADSS Server then you should read the section **5.9** of this document. Note the documents have been encrypted using a special GoSign applet module.

It is generally recommended that all readers should review section **3** and section **4** (information about writing ADSS clients)

3 Integration with a Business application

A business application can communicate with ADSS Server using standard web services calls. As explained above there are defined schemas for the different request/response messages between the business application and ADSS Server. There are several use cases for such interactions:

- Signing data
- Verifying signatures
- Hashing data
- Document assembly
- Generating a key pair and an associated digital certificate
- Deleting a key pair and its corresponding certificates
- Renewing a key pair and an associated digital certificate
- Changing the Authorisation Code associated with private key usage
- Validating X509 digital certificates
- Securely archiving, exporting and deleting data from the ADSS long-term archive
- Decrypting encrypted data using private decryption keys held on the ADSS Server

An application request can be wrapped in a SOAP envelope or wrapped in standard HTTP message then sent to ADSS Server. The request is processed and ADSS Server returns an XML response wrapped in a SOAP message or a wrapped in standard HTTP message.

If ADSS Server cannot successfully parse or action the request it responds with an error message.



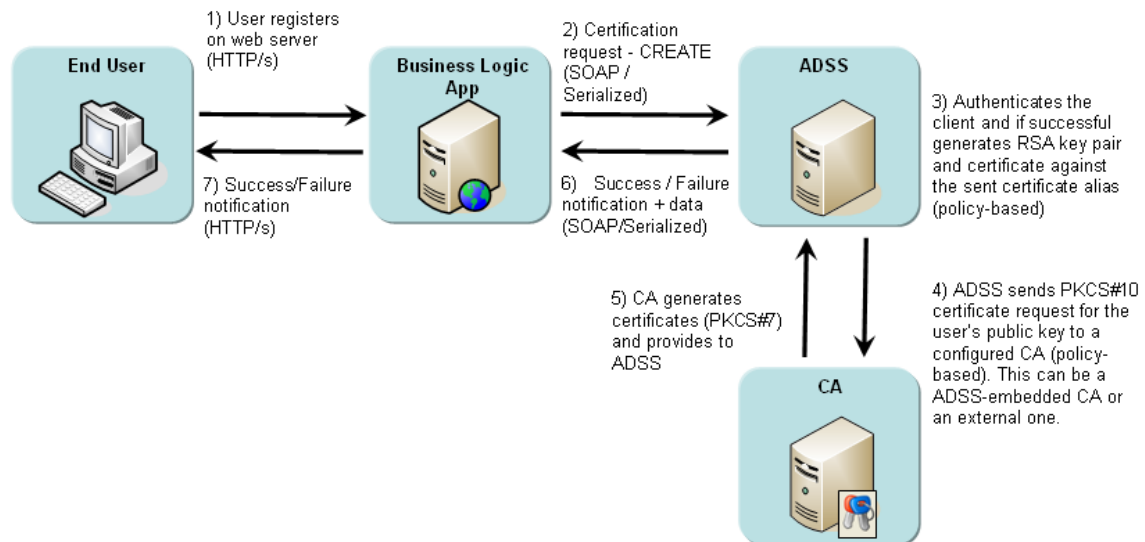
Consult the relevant schema section to learn know more about the request/response formats

3.1 Generating / Registering a Key Pair and Certificate

This section applies only to multi-user server-side signing mode. When an end-user requires a key pair and certificate then the application needs to send a certificate creation request to an ADSS Server. The server generates the key pair and obtains a certificate using either its embedded local CA or by communicating with an external CA. The keys and certificates generated can now be used to sign documents or data.

If the business application has already generated/obtained a key pair and a certificate signing request (CSR or PKCS#10) then this can also be sent to ADSS Server for certification. Alternatively if the key pair have already been generated and certified then the complete package (in the form of a PKCS#12 object) can still be registered with the ADSS Server, e.g. for server-side signature creation.

A high-level review of the process involving the user, business application, ADSS Server and the CA system is illustrated below.



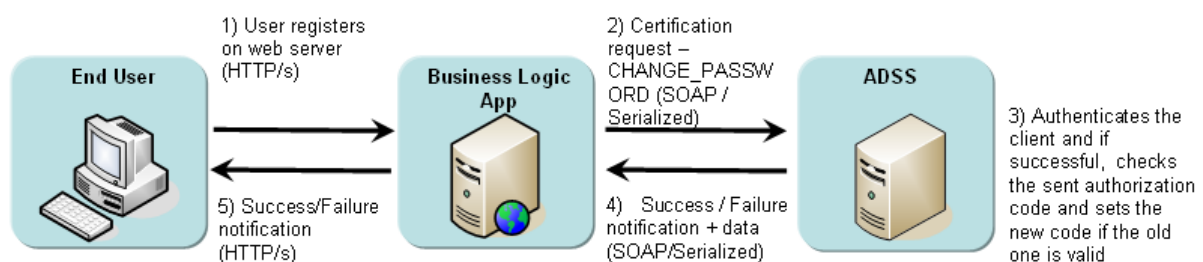
3.2 Changing an end-user key Authorisation Code

An Authorisation Code protects a user's private signing key from unauthorised use. The Authorisation code should only be known to the owner of the private signing key, such that only the owner can use their key on ADSS Server for signing purposes. For certain systems the business application may have strongly authenticated the user using a time-based token or mobile phone code etc and thus either release the unique authorisation code from a user information database or perhaps use a single code for all users relying on the application security to protect all such transactions.

Depending on the application clients may be able to change their authorisation code from time to time, e.g. routinely as part of their security policy or as required.

To change authorisation code the current code must be confirmed and a new authorisation code entered – all under the control of the business application. The application makes the call to the ADSS Server requesting this change. The authorisation code is checked with and applied to a PKCS#12 (PFX) file stored within the ADSS Server database. ADSS Server first verifies that the existing authorisation code is valid before changing the code to the new value.

Note that the authorisation code can only be changed when private signing keys are held in software and not when an HSM is used for holding ADSS Server private keys. The reason for this is that when an HSM is used PKCS#12 files are not required or maintained by ADSS Server.

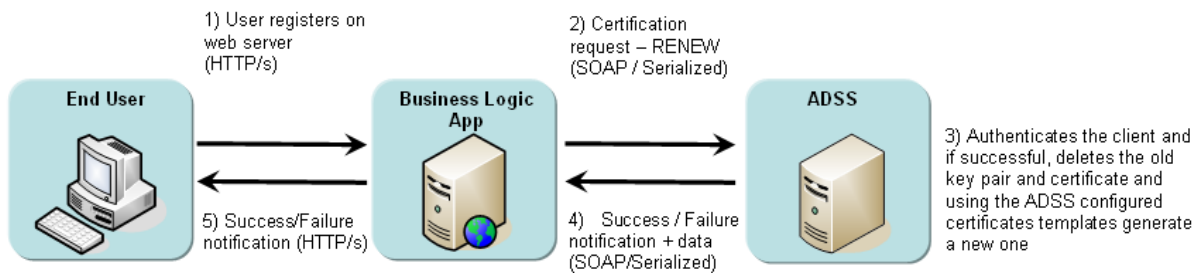


3.3 Renewing a Key pair and Certificate

ADSS Server provides various details about digital certificates in its response data. One attribute provides information on certificate expiry and this allows business applications to track impending expiry dates of client certificates. The business application has the ability therefore to check whether a certificate should be renewed. For example if a certificate has less than say 14 days before its expiry date then the application may choose to send a certificate renewal request to the ADSS Server. This enables a smooth migration from one key-pair to another avoiding a re-registration process and makes security services more user-friendly.

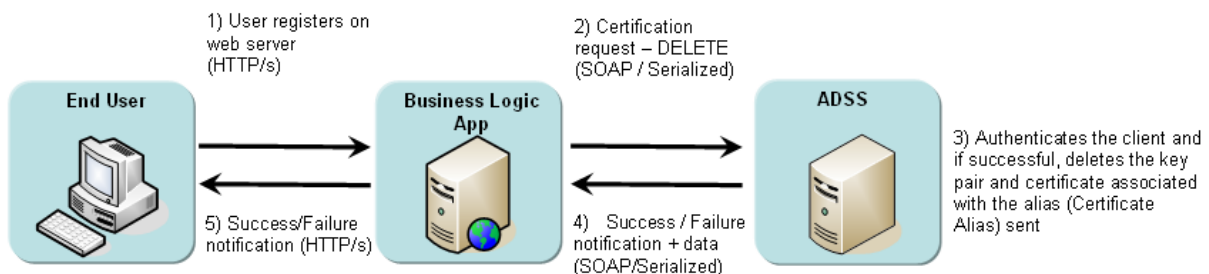
When ADSS Server receives a renewal instruction it generates a new key pair and then certifies the new public key using the appropriate CA. The old key pair and certificate are deleted. The new

certificate is generated according to the ADSS Server policy sent in the request (or the default certification policy).



3.4 Deleting a Key pair and Certificate

A business application can delete an existing key and certificate by sending a certificate deletion request to the ADSS Server. After authenticating the client, ADSS Server deletes the key pair and the certificate.



3.5 Retrieving Private Key (PKCS#12 object) and Certificate

The business application can also request the ADSS to return the PKCS#12 and the associated certificate chain when required by the business application. The business application itself or a client side plug-in (e.g. GoSign applet) can then open the PKCS#12 using locally provided password to extract the Private Key and then use the Private Key for client side signing operations

3.6 Server-side Document Signing

The ADSS Server provides flexible features for signing documents using ADSS Server stored keys or keys held by a desktop client. The facilities provided are:

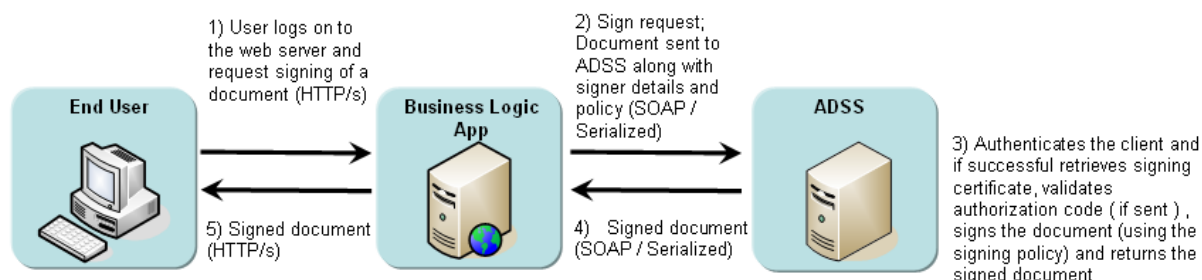
- Server-side signing – using the web services APIs
- Client-side signing – using the GoSign Applet with browser based clients OR using GoSign.NET with thick client applications along with web service APIs for server-side processing

This section describes server-side signing and subsequent sections describe the client-side signing process.

When a business application needs data or a document to be signed, it calls ADSS Server requesting it to sign the data using a specified signing certificate and associated private key held by ADSS Server. The application supplies the user's authorisation code for the signing certificate (not required if an HSM is used).

ADSS Server checks the authorisation code is correct for the target certificate, signs the data/document and returns the signature or signed document in its response back to the application. The certificate used to sign the document may be associated with an end-user or may be a corporate-level certificate registered in the ADSS Key Manager section. Note that if the signing certificate is not associated to an end-user i.e. present in Key Manager then the authorization code is not required at all as such a certificate is not tied with any end-user but assigned to business applications.

If PDF signatures are to be created, then the signature appearance is defined within the signature profile created on ADSS Server or is fully configurable by application call parameters sent in the request message. The signing process is illustrated below:



3.7 Client-Side Document Signing

Signing keys are often held in secure tokens such as a smart card, a USB token, or a soft token, under the direct control of the user. In this case as the signing key is available at the user end rather than on the ADSS Server, this means the signature must be created at the client-side. These keys can be accessed to sign documents via the Windows key store interface (Windows CAPI) or the PKCS#11 interface for other key stores like for Firefox or other browser types.

The ADSS Server provides two ways to access these client-side keys for signing:

1. The **GoSign Applet** for browser/web based applications
2. The **GoSign.NET** library calls for standalone applications or enterprise client/server applications

Both of the GoSign methods can use web service requests to generate document hashes to sign using the client keys and to assemble the signature with the document to produce the final signed document. The GoSign applet is also capable of signing a document by hashing and signing the document locally. GoSign Professional applet can also display PDF documents in its secure PDF viewer module. For further information on the GoSign applet please see its documentation set, provided within the ADSS Client SDK package.

3.7.1 Signing Documents with the GoSign Applet

Traditionally local software has been required to allow end-users to sign forms, data or documents. Increasingly this is viewed as unacceptable as organisations seek to tightly control their desktops and stop new third-party software being installed.

To address this, the Ascertia GoSign Applet can be used to provide a “zero-footprint” client-side signing solution. No installation is required since it uses the native ability of common browsers such as Internet Explorer and Firefox to run signed Java Applets². The applet can be signed by the Government, Bank or organisation running the service with which the end-user already has a trust relationship. Ascertia delivers the GoSign applet and required libraries already signed by a code signing certificate issued by a public CA.

The GoSign Applet is associated with a web-based business application for digitally signing data or documents using an end-user signing key held accessible with Windows CAPI or PKCS#11 interface³ - either in software or in a token or smartcard using appropriate third party Cryptographic service provider (CSP) software.

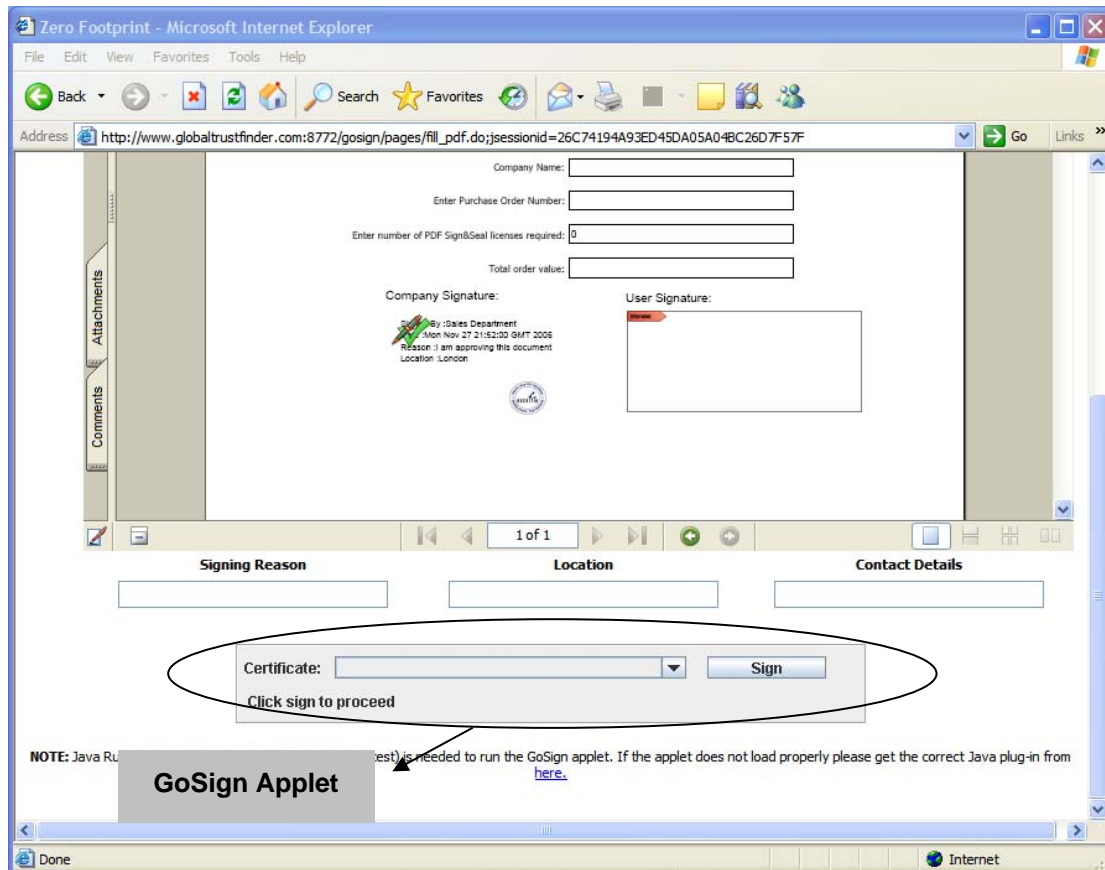
A new option in GoSign Professional applet is the use of roaming credentials. In this case the business application can provide the roaming credential secure container (which contains the user's Private Key and X509 certificate) from the ADSS Server to GoSign applet if it doesn't want to use a CAPI or PKCS#11 keys during signing operation. Note the GoSign applet initially generates these roaming credentials and provides them to the business application for sending to the ADSS Server for

² Note that ActiveX controls can also be provided on a project basis where these may be preferred, e.g. for internal systems.

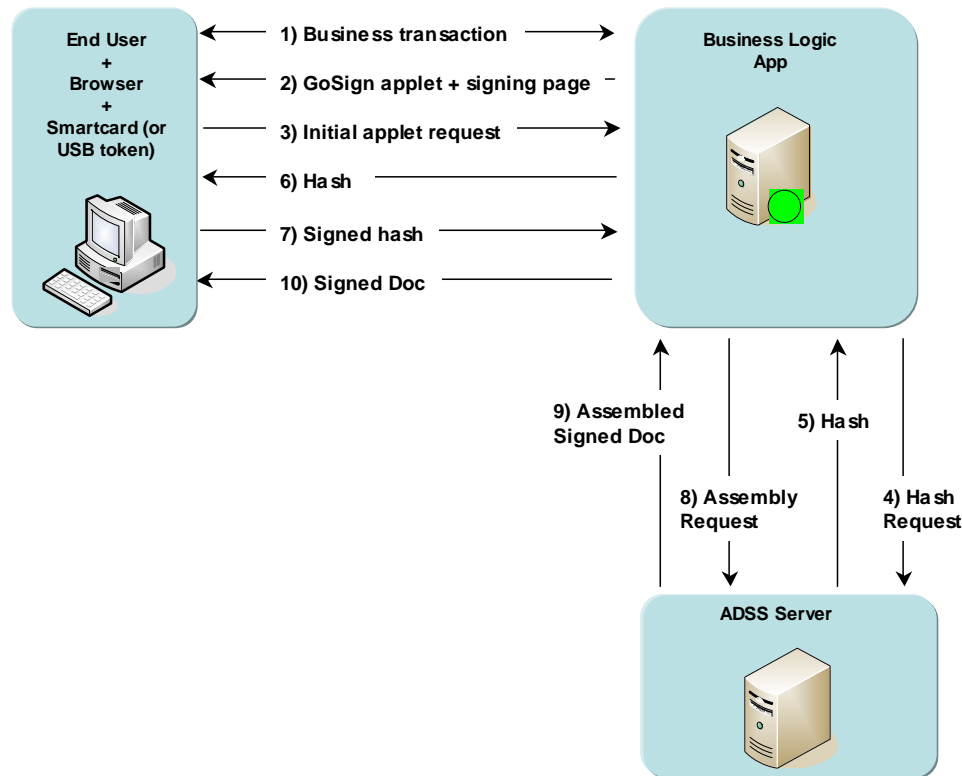
³ PKCS#11 interfaces to crypto-subsystems is primarily for Unix and MAC environments. Note that for MAC and Linux environments only the PKCS#11 interface can be used.

certification and secure storage. The details of integrating the applet into the web page are described in the GoSign Developer's Guide distributed with the ADSS Client SDK.

As one example of GoSign applet usage, the following screenshot illustrates a PDF being signed within a browser using Adobe® Reader. The HTML page contains fields for optional signing reason, location and contact details to be filled by the user. The certificate is to be selected using a drop-down list and then the "sign" button is pressed to sign the data. Local languages can be customised as needed. Note below that a company signature is already placed on the PDF. The user's digital signature will be applied within the currently blank 'User Signature' field:



A typical zero-footprint client-side signing process using the ADSS Server and the GoSign Applet proceeds as below. This assumes the document hash is calculated by the ADSS Server at the request of the business application. However note the latest versions of the GoSign applet can perform local hashing of the document within the applet.

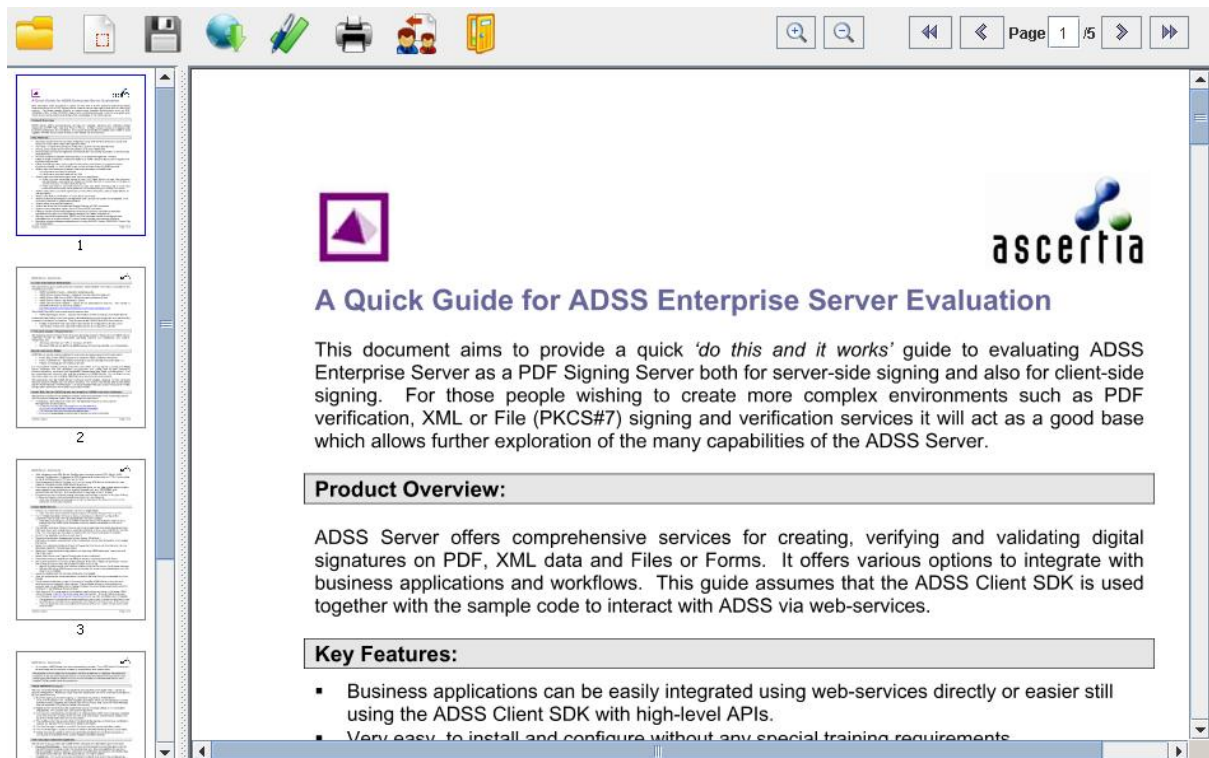


1. A user interacts with a web application and at some stage of the transaction; a PDF summarising the agreement is displayed using a browser embedded PDF Reader.
2. The web application requests the end-user to sign the document by adding the GoSign Applet to the web page. Optional details can be added as shown above; legal notices can be displayed as required by the business. The user can select a signing certificate using the drop-down list although this may be filtered on demand to only allow signing keys from trusted issuers
3. The GoSign Applet sends all of the data which needs to be set inside the PDF to the web application before signing
4. The web application makes a call to the ADSS Server to request a hash value to be calculated based on the PDF document and the above user supplied details
5. ADSS Server returns the hash value to the web application
6. The web application passes this hash value to the GoSign Applet to sign (depending on local settings the user may be asked to enter a password/PIN to access the private key at this point). The GoSign Applet returns the signed hash value (digital signature) to the web application i.e. PKCS#7/CMS
7. The web application then requests ADSS Server to assemble the digital signature within the PDF document and provides the signed hash value i.e. PKCS#7/CMS as input to this request
8. The web applications request the document assembly from the ADSS Server
9. ADSS Server returns the signed PDF document to the web application
10. The web application displays the user-signed PDF document within the browser session and may optionally ask the user to confirm that they confirm this represents their signed intent to proceed

An example demonstration application based on the above interaction is available at: <http://www.globaltrustfinder.com/ZerofootPrintSigningStep1.aspx>

GoSign Applet can also be used in several other scenarios where you can get GoSign Applet to sign PDF, XML and any file by locally hashing and signing all within the applet. GoSign Applet (Professional version only) also has a built-in PDF Viewer which can be used if required.

The PDF Viewer is fully integrated with the GoSign signing capabilities. Below is a screenshot of GoSign PDF Viewer:

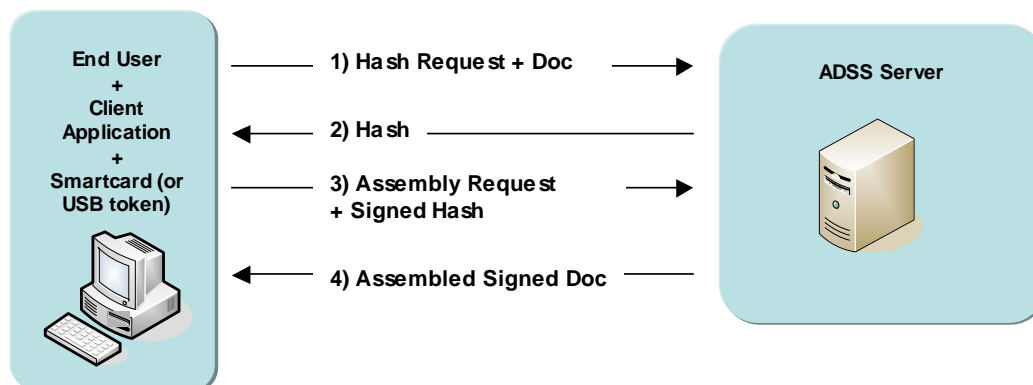


For further details about how to use the GoSign Applet with ADSS Server refer to the GoSign Developers Manual. The source code for the demonstration application is available.

3.7.2 Signing Documents with GoSign.NET

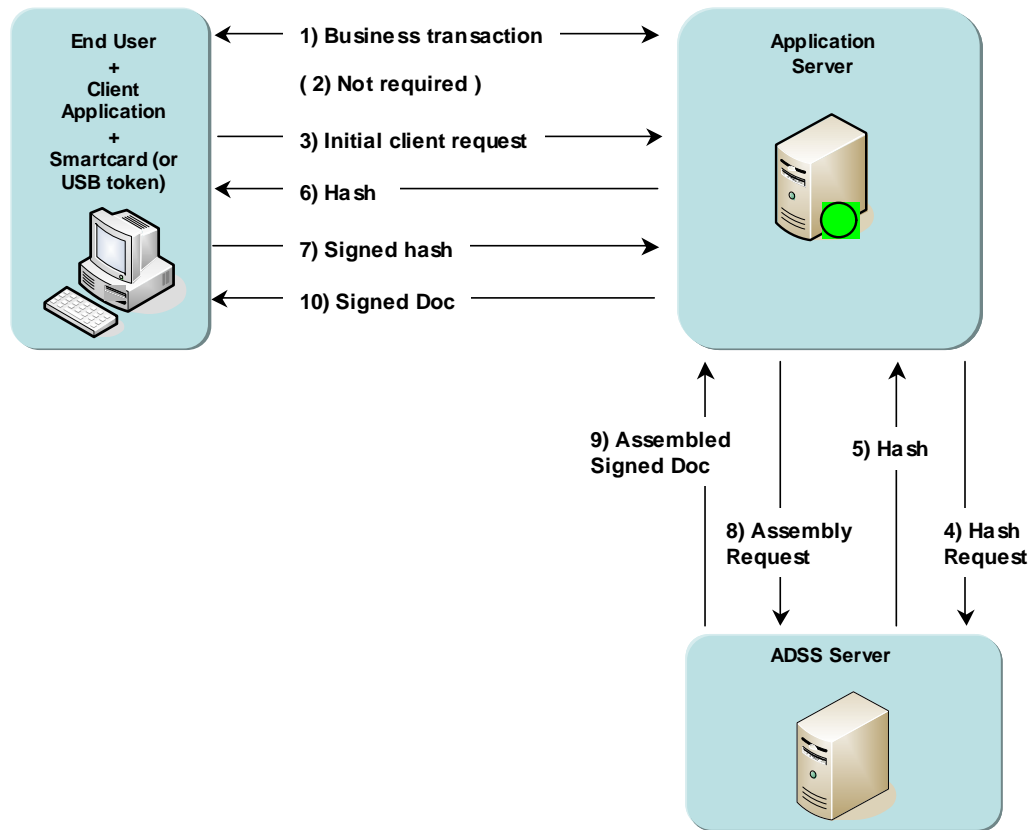
In the same way that the GoSign Applet accesses the Windows certificate store to sign documents in the browser, GoSign.NET can be integrated with a standalone desktop application or an enterprise client/server application.

For a standalone desktop application GoSign.NET is integrated directly on the desktop and accesses Windows certificate store, using the ADSS Client SDK KeyManager API (see section Error! Reference source not found.), to sign documents. The following diagram shows the necessary steps:



Here, as with the applet, requests are made to the ADSS Server for a hash that will be signed on the desktop and then the resulting signature is assembled by the ADSS Server with the original document. The web service calls are made directly from the desktop application to the ADSS Server.

It might be that the application architecture requirement is for a client/server model where the client application must be authenticated to the server which then carries out all secure transactions with the ADSS Server. Where a client/server model is being used, the GoSign.NET client signing functions are again integrated with the desktop application. But the calls to the ADSS Server for hashing and document assembly are made by the server application, as in the steps below:



These steps can be mapped onto the same steps listed for the GoSign Applet steps, where the desktop and server applications replace the browser and web server application.

3.7.3 Hashing and Assembly

When a business application wishes to hash data it calls ADSS Server, requesting it to hash the specified document or data. The most likely use of this call is to work in client-side signing mode when the applet/desktop application requires server-side hashing. In such scenarios once the hash is returned, the application interacts with the GoSign components to have this hash signed.

For PDFs the next stage of the signing process is to embed the signature within the PDF document. To do this the application makes an Assembly request to ADSS Server, sending the signature object so that it can be correctly embedded within the document. The process flow is similar to that shown in the previous diagrams.

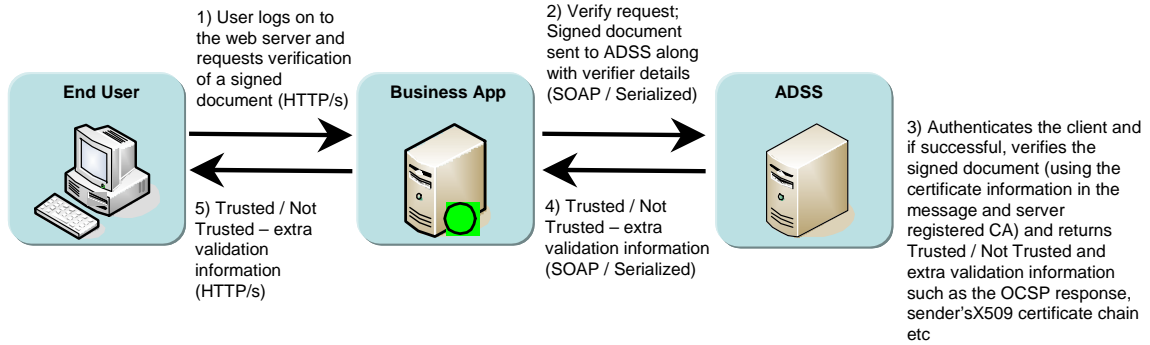
3.8 Verifying Signed Documents

When a business application needs signed data or a signed document to be verified or validated, it calls the ADSS Server requesting it to verify the data using certificate information held within the request in combination with trusted CA information registered with the server. The server verifies the signed data/document and returns the verification response back to the application.

The verification response is either Trusted or Not Trusted. Further information supplied in the response can be examined to appreciate why it is Trusted or Not Trusted such as the Quality Level of trust for a Trusted response or more detail on why an individual signature verification failed.

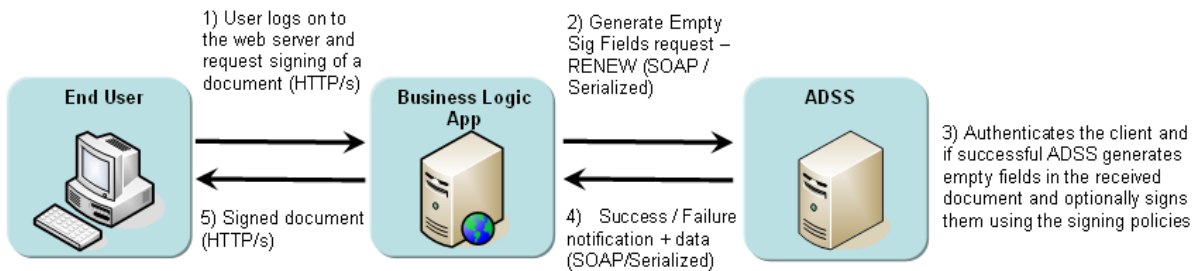
Signed data/documents can have multiple associated signatures and the response will contain an overall verification check for those signatures of Trusted / Not trusted. To be Trusted, all signatures must be valid and if any one signature fails the data/document is treated as Not Trusted. The response contains verification details for each of the signatures.

The verification process is illustrated below:



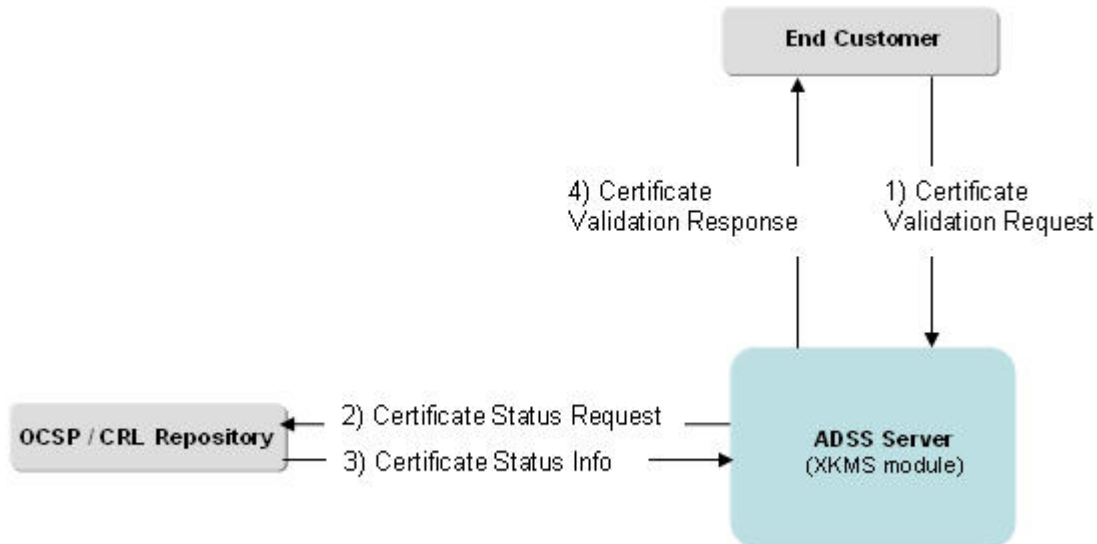
3.9 Creating an Empty Signature Field on PDF Documents

In some instances an empty signature field may be need to be inserted within a PDF document – for example when creating additional signature fields prior to certifying (and thus locking) a PDF. To achieve this, a business application can call ADSS Server requesting it to create an empty signature field within a target PDF document.



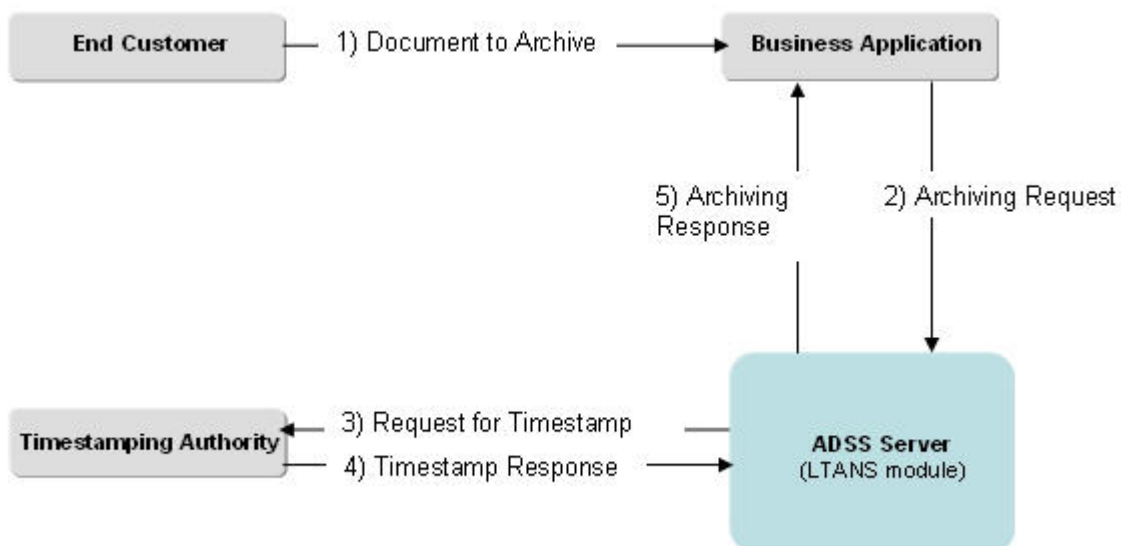
3.10 XKMS Validate Service

A business application may need to validate certificates as part of verifying a digital signature or as part of creating long-term signatures which contain embedded certificate validation information for the signer. To do this a business application can make an XKMS Certificate Validation request as depicted below:



3.11 LTANS Service

To utilise the ADSS LTANS Service, a business application sends a data object (document, data, signed transaction etc.) to the ADSS Server. The ADSS LTANS Service will create a secure archive object for the data object using the configured Time Stamp Authority (TSA). The archive object complies with the IETF XML Evidence Record Syntax (XMLERS) specification. The communication with the ADSS LTANS Service is conducted over the IETF Long Term Archive Protocol (LTAP) as illustrated below:



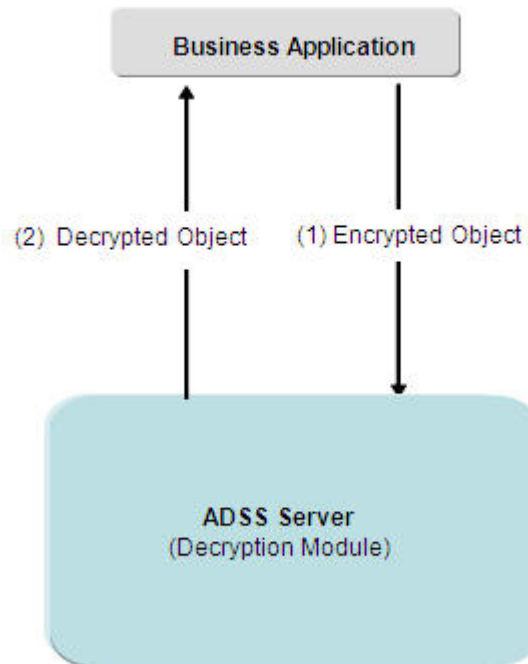
Note ADSS LTANS Service can either store the archive object internally in its database or return it to the business application for local storage (e.g. in case the business application is actually a Document Management System).

If ADSS LTAN Service is storing the archive object then the business application can at a later date export this out of the archive:

The business application may also request the ADSS LTANS Service to delete a particular archive object.

3.12 Decryption Service

End users may need to submit encrypted (possibly signed and encrypted) documents to a business application, e.g. an e-Tendering application. A special version of GoSign Applet can provide XML Encryption option. At a later date the business application will need to have the encryption removed through the ADSS Decryption Service. The following diagram illustrates the process:



4 Developing ADSS Web Service Clients

To use the ADSS signature/certificate verification, certification, signing, hashing, assembly and empty field creation (and signing) web services, calling applications need to send a SOAP request to the desired web service URL. ADSS then responds with a SOAP message containing the response which may be a valid response or a SOAP fault if ADSS could not parse the SOAP request.

The default web service URLs are:

Service	Default ADSS Address
Signing,	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/signing/dsi SSL with Server auth only: https:// <ADSS machinename>:8778/adss/signing/dsi Non SSL: http:// <ADSS machinename>:8777/adss/signing/dsi
Hashing	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/signing/dhi SSL with server auth only: https:// <ADSS machinename>:8778/adss/signing/dhi Non SSL: http:// <ADSS machinename>:8777/adss/signing/dhi
Assembly	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/signing/sai SSL with server auth only: https:// <ADSS machinename>:8778/adss/signing/sai Non SSL: http:// <ADSS machinename>:8777/adss/signing/sai
Empty Field creation	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/signing/esi SSL with server auth only: https:// <ADSS machinename>:8778/adss/signing/esi Non SSL: http:// <ADSS machinename>:8777/adss/signing/esi
Verification	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/verification/svi SSL with server auth only: https:// <ADSS machinename>:8778/adss/verification/svi Non SSL: http:// <ADSS machinename>:8777/adss/verification/svi
Certification	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/certification/csi SSL with server auth only: https:// <ADSS machinename>:8778/adss/certification/csi Non SSL: http:// <ADSS machinename>:8777/adss/certification/csi
OCSP	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/ocsp SSL with server auth only: https:// <ADSS machinename>:8778/adss/ocsp Non SSL: http:// <ADSS machinename>:8777/adss/ocsp
TSA	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/tsa SSL with server auth only: https:// <ADSS machinename>:8778/adss/tsa Non SSL: http:// <ADSS machinename>:8777/adss/tsa
XKMS	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/xkms SSL with server auth only: https:// <ADSS machinename>:8778/adss/xkms Non SSL: http:// <ADSS machinename>:8777/adss/xkms
LTANS	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/ltap SSL with server auth only: https:// <ADSS machinename>:8778/adss/ltap Non SSL: http:// <ADSS machinename>:8777/adss/ltap
Decryption	SSL with client-server auth: https:// <ADSS machinename>:8779/adss/decryption SSL with server auth only: https:// <ADSS machinename>:8778/adss/decryption Non SSL: http:// <ADSS machinename>:8777/adss/decryption

Ascertia provides easy to use libraries written in the JAVA and C# programming languages to minimise application integration development effort. Using these libraries business applications can be developed easily and quickly, whilst minimizing errors.

Please see the integration samples (Provided with ADSS Client SDK. That shows how you can write programs to integrate with ADSS using Java & C# APIs).



It is recommended that the ADSS Client SDK libraries are used in your development process. Their use considerably reduces development time because there is no need to create XML writers and parsers for the web-service protocol. If it is necessary to write the requests yourself OR using a development language which can't utilize Java or C# APIs then it is important to ensure that the requests comply with the corresponding schema file plus are compliant with the schema details mentioned in section 5

Ascertia has provided samples to guide developers in accessing ADSS services as part of ADSS Client SDK. The Readme.html file in the ADSS Client SDK folder gives further information on the sample directory structure and how to run the samples.



Note that Ascertia does not provide APIs for request creation or response parsing for the OCSP and TSA services, which are widely implemented protocols. You may use any commercial or free API for accessing these services. Some of the free APIs are:

OCSP

<http://www.bouncycastle.org/docs/docs1.4/org/bouncycastle/ocsp/package-summary.html>

<http://java.sun.com/j2se/1.5.0/docs/guide/security/pki-tiger-beta1.html>

Timestamping

<http://www.bouncycastle.org/docs/tspdocs1.3/org/bouncycastle/tsp/package-summary.html>

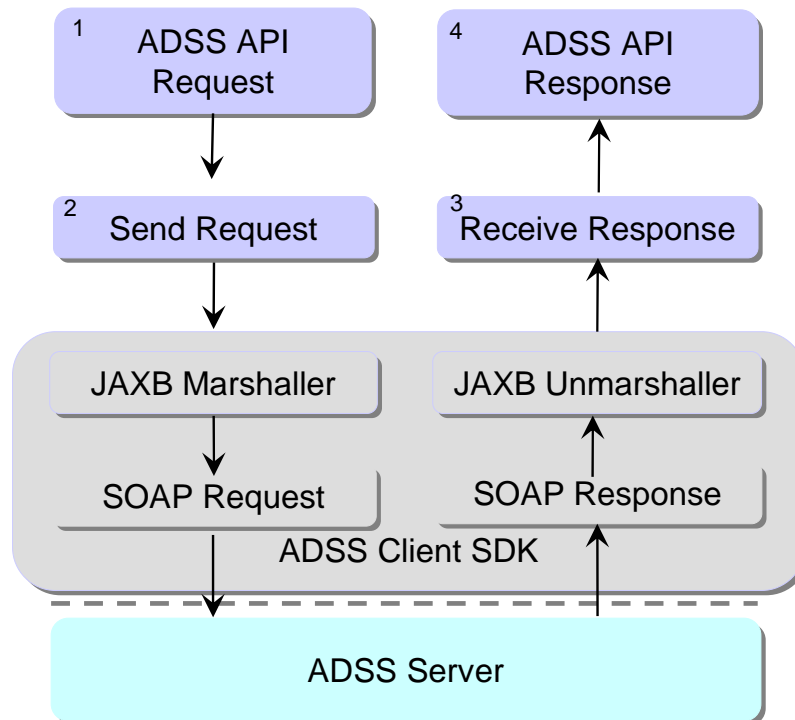
Note that Bouncy Castle also provides a .Net variant of their Java based cryptographic APIs which can be used with .Net clients.

The following lists the schema files associated with each ADSS Server service. These are located inside the **schema** folder provided with ADSS Client SDK.

Service	Schema File
Signing	adss_signmodule.xsd, oasis-dss.xsd
Verification	adss_verifymodule.xsd
Hashing, Assembly	adss_signmodule.xsd
Certification	adss_certmodule.xsd
OCSP	The protocol's ASN.1 specification is available from: http://www.ietf.org/rfc/rfc2560.txt
TSA	The protocol's ASN.1 specification is available from: http://www.ietf.org/rfc/rfc3161.txt
XKMS	The W3C specification is available at from: http://www.w3.org/TR/xkms2/ XKMS schema is also available under /Schema directory in Client SDK
LTANS	The XML specification is available from: http://tools.ietf.org/html/draft-ietf-ltans-ltap-07 LTANS schema is also available under /Schema directory in Client SDK
Decryption	The protocol specification is available at: http://www.oasis-open.org/committees/download.php/25384/oasis-dss_profile-encryption_A-SIT_v0.1.doc Decryption schema is also available under /Schema directory in Client SDK

4.1 Using the ADSS Java API

The ADSS Java library contains all the necessary components to create an XML structure that is understood by ADSS Server web services for Signing, Verification, Certification, Empty Signature Fields and Document Assembly. These components performs behind the scene XML marshalling using third party JAXB and send the request/receive response using SOAP communication packages (supplied with ADSS Client SDK) and thus the calling application has to write only few lines of code to get things done. The general program flow is represented in **Error! Reference source not found.**



The building blocks for this flow are outlined below and are demonstrated in detail by the samples provided with the ADSS Client SDK. This contains all the necessary packages to build a complete ADSS Server integrated business application.

The following sections provide code fragments for coding the ADSS API tasks 1, 2, 3, 4. Section 5 describes the XML element structure which is useful for identifying elements of the protocol while studying this code.

The ADSS Server Admin Manual describes how to configure the ADSS Server with keys, certificates, trust anchors, profiles etc that are required to process business application requests. Also, for the sample applications, a database script is provided (as part of ADSS Client SDK) to automate the server configuration. These server configurations are required for the sample application's code to work as it is.

4.1.1 Server-side Signing

A signing request is populated with various request information along with a PDF document and sent the server. Once the signed PDF is received, it is stored in a file.

```

// create the signing request object. Use OriginatorID and Certificate
// Alias which is already registered at ADSS
SigningRequest obj_signingRequest = new SigningRequest(OriginatorID,
"test_input_unsigned.pdf", SigningRequest.MIME_TYPE_PDF);
  
```

```

// set the Profile ID against which document will be signed
obj_signingRequest.setProfileId("adss:signing:profile:001");
// set a request id (provided by the application)
obj_signingRequest.setRequestId("1");
// send the request to the server
SigningResponse obj_signingResponse = (SigningResponse)
obj_signingRequest.send("http://www.test.com" + "/adss/signing/dsi");
if (obj_signingResponse.isSuccessfull()) {
    obj_signingResponse.publishDocument("test_input_signed.pdf");
    System.out.println("\nRequest has been processed successfully.");
}
else {
    System.out.println(obj_signingResponse.getErrorMessage());
}

```

4.1.2 Client-side Signing

Hashing and document assembly support is provided for use with GoSign Applet applications

4.1.2.1 Hashing and Document Assembly

See section 3.7.33.7.1 is an application overview of this process. The ADSS server side code performs the hashing and assembly tasks.

A hashing request is populated with various request information and a PDF document to be hashed. The hash is then extracted from the response, by the business application. The hash can then be signed either by server side code or by a user's signing software, GoSign Applet, smart card or secure USB token. The generated signature is then supplied to the server that was returned with the hash response.

```

// create HashRequest object
DocumentHashingRequest obj_documentHashingRequest = new
    DocumentHashingRequest("samples_test_client", "test_input_unsigned.pdf",
byte_cert);
obj_documentHashingRequest.setProfileId("adss:signing:profile:001");
// set the request Id - this is returned in the hash response
// to identify the document to be assembled in a later request
obj_documentHashingRequest.setRequestId("1");
// override the default signing reason, contact info set by ADSS Server
obj_documentHashingRequest.overrideProfileAttribute(
    DocumentHashingRequest.SIGNING_REASON, "Testing");
obj_documentHashingRequest.overrideProfileAttribute(
    DocumentHashingRequest.CONTACT_INFO, "123, Surrey");

// send the request and get the response
DocumentHashingResponse obj_documentHashingResponse = (DocumentHashingResponse)
obj_documentHashingRequest.send("http://www.test.com" + "/adss/signing/dhi");
if (obj_documentHashingResponse.isSuccessfull()) {
    System.out.println("\nDocument hash recieved successfully.");
    try {
        // create a PKCS#7 from the hash received from ADSS
        b_signature = createPKCS7(obj_documentHashingResponse.getDocumentHash());
    }
}

```

```

catch (Exception ex) {
    ex.printStackTrace();
    throw ex;
}
// create a Signature Assembly request with the help
// of the generated PKCS#7
SignatureAssemblyRequest obj_signatureAssemblyRequest = new
    SignatureAssemblyRequest("samples_test_client", b_signature,
        obj_documentHashingResponse.getDocumentId());
// set the request Id - which was earlier used
obj_signatureAssemblyRequest.setRequestId("1");
// send the Signature Assembly request to the server
SignatureAssemblyResponse obj_signatureAssemblyResponse =
    (SignatureAssemblyResponse)
    obj_signatureAssemblyRequest.send("http://www.test.com" +
        "/adss/signing/sai");

if (obj_signatureAssemblyResponse.isSuccessfull()) {
    obj_signatureAssemblyResponse.publishDocument (
        "test_input_signed.pdf".trim());
    System.out.println("\nRequest has been processed successfully.");
}
else {
    System.out.println(
        obj_signatureAssemblyResponse.getErrorMessage());
}
}
else {
    System.out.println(obj_documentHashingResponse.getErrorMessage());
}
}

```

4.1.3 Creating Empty Signature Fields

An empty signature field request is populated with various request information along with a PDF document and sent the server. Once the signed PDF is received, it is stored in a file.

```

// create EmptySignatureField request object
EmptySignatureFieldRequest obj_emptySigFieldRequest = new
EmptySignatureFieldRequest( "samples_test_client", "test_input_unsigned.pdf");
obj_emptySigFieldRequest.setProfileId("adss:signing:profile:002");
// set the Profile ID against which document will be signed
obj_emptySigFieldRequest.setSigningInfo("adss:signing:profile:001",
    "samples_test_signing_certificate");
// set a request id (provided by the application)
obj_emptySigFieldRequest.setRequestId("1");
// override the default signing reason set by ADSS Server
obj_emptySigFieldRequest.overrideProfileAttribute(
EmptySignatureFieldRequest.SIGNING_REASON, "Testing");

// send the request to the server
EmptySignatureFieldResponse obj_emptySigFieldResponse =
    (EmptySignatureFieldResponse) obj_emptySigFieldRequest.send(
        "http://www.test.com" + "/adss/signing/esi");

```

```
// process the response
if (obj_emptySigFieldResponse.isSuccessfull()) {
    obj_emptySigFieldResponse.publishDocument("test_input_signed.pdf");
    System.out.println("\nRequest has been processed successfully.");
}
else {
    System.out.println(obj_emptySigFieldResponse.getErrorMessage());
}
}
```

4.1.4 Verification

The ADSS API supports the verification of PDF signatures, PKCS#7/CMS or XML DigSig formats. The following code verifies a single signature on a PDF document.

```
//constructing request for pdf signature verification
SignatureVerificationRequest obj_signatureVerificationRequest = new
    SignatureVerificationRequest("samples_test_client",
    "PDF_v1.6_ACROBAT_RTC1_S1-1024-C1");
obj_signatureVerificationRequest.setCertificateQualityLevel("2");
obj_signatureVerificationRequest.setSignatureQualityLevel("2");

//add RespondWith items
obj_signatureVerificationRequest.addRespondWithItem(
    VerificationRequest.RESPOND_WITH_BASIC_CONSTRAINTS);
obj_signatureVerificationRequest.addRespondWithItem(
    VerificationRequest.RESPOND_WITH_KEY_USAGE);
obj_signatureVerificationRequest.addRespondWithItem(
    VerificationRequest.RESPOND_WITH_VALID_FROM);
obj_signatureVerificationRequest.addRespondWithItem(
    VerificationRequest.RESPOND_WITH_VALID_TO);
obj_signatureVerificationRequest.addRespondWithItem(
    VerificationRequest.RESPOND_WITH_ISSUER_NAME);

//constructing signature element so that it can be added into the signature
//verification request
SignatureInfo obj_signatureInfo = new SignatureInfo(
    "signature.id.001", args[0].trim(), SignatureInfo.SIGNED_DOCUMENT_TYPE_PDF);
obj_signatureInfo.setSignatureFormat(SignatureInfo.SIGNATURE_FORMAT_OTHER);
obj_signatureInfo.addRespondWithItem(
    VerificationRequest.RESPOND_WITH_BASIC_CONSTRAINTS);
obj_signatureInfo.addRespondWithItem(VerificationRequest.RESPOND_WITH_KEY_USAGE);
obj_signatureInfo.addRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_FROM);
obj_signatureInfo.addRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_TO);
obj_signatureInfo.addRespondWithItem(VerificationRequest.RESPOND_WITH_ISSUER_NAME);

//adding signature(s) into the signature verification request */
obj_signatureVerificationRequest.addSignatureInfo(obj_signatureInfo);

//sending the above constricted request to the ADSS Server
VerificationResponse obj_verificationResponse = (VerificationResponse)
obj_signatureVerificationRequest.send("http://www.test.com" +
"/adss/verification/svi");

//parsing the response
if (obj_verificationResponse.isSuccessfull()) {
```

```
        System.out.println("\nRequest has been processed successfully");
        System.out.println("\nResponse Id : " +
obj_verificationResponse.getRequestId());
        System.out.println("Response Type : " +
obj_verificationResponse.getResponseTypeId());
    }
    else {
        System.out.println("\nRequest hasn't been processed successfully.");
        System.out.println("\nError Code : " +
obj_verificationResponse.getErrorCode());
        System.out.println("Result Minor : " +
obj_verificationResponse.getResultMinor());
    }
}
```

4.1.5 Certification

Certification requests can be made to the ADSS Server for certificate/key generation, certificate renewal, certificate/key deletion and changing a password on the private key data.

```
// create the certification request object
// set the request type to create a certificate. Could also be
// RequestTypeEnum.DELETE, RequestTypeEnum.RENEW,
// RequestTypeEnum.CHANGE_PASSWORD for other certification actions
CertificationRequest obj_certificationRequest = new
    CertificationRequest("samples_test_client",
        CertificationRequest.REQUEST_TYPE_CREATE_CERTIFICATE,
        "Test Certificate");
// set the certification Profile ID used for certificate generation
obj_certificationRequest.setProfileId("adss:certification:profile:001");
// set the password of PKCS#12 generated at the server
obj_certificationRequest.setPkcs12Password("password");
// application generated request ID
obj_certificationRequest.setRequestId("1");

// indicate what information the ADSS server should respond with
// respond with the generated certificate in base64 format
obj_certificationRequest.addRespondWithItem(
    CertificationRequest.RESPOND_WITH_CERTIFICATE);
// respond with a PKCS#12 object containing the certificate and private key
obj_certificationRequest.addRespondWithItem(
    CertificationRequest.RESPOND_WITH_PKCS_12);
// respond with a PKCS#7 object containing the certificate
obj_certificationRequest.addRespondWithItem(
    CertificationRequest.RESPOND_WITH_PKCS_7);
// respond with the certificate expiry data
obj_certificationRequest.addRespondWithItem(
    CertificationRequest.RESPOND_WITH_EXPIRY_DATE);

// set other certificate request information
obj_certificationRequest.overrideProfileAttribute(
    CertificationRequest.SUBJECT_DN, "CN=Alice");

// send the certification request to the ADSS server
CertificationResponse obj_certificationResponse = (CertificationResponse)
    obj_certificationRequest.send("http://www.test.com" +
```

```

        "/adss/certification/csi");
if (obj_certificationResponse.isSuccessfull()) {
    obj_certificationResponse.publishCertificate("certificate.cer");
    obj_certificationResponse.publishPKCS12("certificate.pfx");
    obj_certificationResponse.publishPKCS7("certificate.p7b");
    System.out.println("\nRequest has been processed successfully.");
}
else {
    System.out.println(obj_certificationResponse.getErrorMessage());
}

```

4.1.6 XKMS Validate

XKMS Validate request can be sent to ADSS XKMS Service to get the validation status of a certificate.

```

//Constructing xkms validate request
ValidateRequest obj_validateRequest = new
    ValidateRequest(ValidateRequest.RESPONDWITH_X509CERT);
//Setting xkms request attributes
obj_validateRequest.setRequestId("adss:xkms:validaterequest:001");
obj_validateRequest.setRespondAddress("http://www.test.com");
//adding RespondWith items
obj_validateRequest.addRespondWith(ValidateRequest.RESPONDWITH_KEYNAME);
obj_validateRequest.addRespondWith(ValidateRequest.RESPONDWITH_KEYVALUE);
obj_validateRequest.addRespondWith(ValidateRequest.RESPONDWITH_SPKI);
obj_validateRequest.addRespondWith(ValidateRequest.RESPONDWITH_X509CHAIN);
obj_validateRequest.addRespondWith(ValidateRequest.RESPONDWITH_X509CRL);
//reading and setting certificate file
obj_validateRequest.setKeyInfo(Util.readFile("certificate.cer"));
//set the time at which request is being created
obj_validateRequest.setTimeInstant(new Date());
//set KeyUsage for provided certificate
obj_validateRequest.addKeyUsage(ValidateRequest.KEYUSAGE_SIGNATURE);
//set UseKeyWith for provided certificate
obj_validateRequest.addUseKeyWith(ValidateRequest.USEKEYWITH_SMIME,
    "alice@example.com");
//Sending the request to the ADSS server
ValidateResult obj_validateResult = (ValidateResult)
    obj_validateRequest.send("http://www.test.com"+ "/adss/xkms");
if (obj_validateResult.isSuccessfull()) {
    System.out.println("Validate Status : "+obj_validateResult.getResultMajor());
    System.out.println("\nRequest has been processed successfully.");
}
else {
    System.out.println(obj_validateResult.getErrorMessage());
}

```

4.1.7 LTANS Archive

A document can be securely archived for longer period using ADSS LTANS Service. Client application can create a document archiving request using ADSS Client SDK as follows:

```
/* Constructing archiving request */
ArchivingRequest obj_archivingRequest = new ArchivingRequest("samples_test_client",
    ArchivingRequest.SERVICE_TYPE_ARCHIVE, "mydata.dat");
obj_archivingRequest.setVersion("1.0");
obj_archivingRequest.setRequestId("Request_001");
obj_archivingRequest.setPolicyIdentifier("adss:ltan:profile:001");
obj_archivingRequest.setTransactionIdentifier("trans.id.001");
obj_archivingRequest.setRespondAddress("http://www.test.com");
obj_archivingRequest.setNonce("Nonce:001");
obj_archivingRequest.setRequestTime(new Date().toString());
//Constructing data string element so that it can be added into the archiving
request
byte[] byteArrArchiveData = Util.readFile("test.pdf");
obj_archivingRequest.setData(byteArrArchiveData);
obj_archivingRequest.setMetaDataValue("PDF");
//Sending the above constructed request to the ADSS server
ArchivingResponse obj_archivingResponse = (ArchivingResponse)
    obj_archivingRequest.send("http://www.test.com" + "/adss/ltap");
if ( obj_archivingResponse.isSuccessfull() ) {
    System.out.println("\nRequest has been processed successfully.");
    System.out.println("Reference Id : " +obj_archivingResponse.getReference());
}
else {
    System.out.println(obj_archivingResponse.getErrorMessage());
}
```

4.1.8 LTANS Export

An archived document can be retrieved from ADSS LTANS Service by using following code:

```
//Constructing archiving request
ArchivingRequest obj_archivingRequest = new ArchivingRequest("samples_test_client",
    ArchivingRequest.SERVICE_TYPE_EXPORT, "http://www.test.com");
obj_archivingRequest.setVersion("1.0");
obj_archivingRequest.setRequestId("Request_001");
obj_archivingRequest.setPolicyIdentifier("adss:ltan:profile:001");
obj_archivingRequest.setTransactionIdentifier("trans.id.001");
obj_archivingRequest.setRespondAddress("http://www.test.com");
obj_archivingRequest.setNonce("Nonce:001");
obj_archivingRequest.setRequestTime(new Date().toString());
//Set reference id to export the archived data
obj_archivingRequest.setReference("1234567890");
//Sending the above constructed request to the ADSS server
ArchivingResponse obj_archivingResponse = (ArchivingResponse)
```

```

        obj_archivingRequest.send("http://www.test.com" + "/adss/ltap");
    if (obj_archivingResponse.isSuccessfull()) {
        System.out.println("\nRequest has been processed successfully.");
        Util.writeFile(obj_archivingResponse.getData(), "exported_data.pdf");
    }
    else {
        System.out.println(obj_archivingResponse.getErrorMessage());
    }
}

```

4.1.9 LTANS Delete

An archived document can be deleted on ADSS LTANS Service by using following code:

```

//Constructing archiving request
ArchivingRequest obj_archivingRequest = new ArchivingRequest("samples_test_client",
    ArchivingRequest.SERVICE_TYPE_DELETE, "http://www.test.com");
obj_archivingRequest.setVersion("1.0");
obj_archivingRequest.setRequestId("Request_001");
obj_archivingRequest.setPolicyIdentifier("adss:ltan:profile:001");
obj_archivingRequest.setTransactionIdentifier("trans.id.001");
obj_archivingRequest.setRespondAddress("http://www.test.com");
obj_archivingRequest.setNonce("Nonce:001");
obj_archivingRequest.setRequestTime(new Date().toString());
//Set reference id to delete the archived data
obj_archivingRequest.setReference("1234567890");
//Sending the above constructed request to the ADSS server
ArchivingResponse obj_archivingResponse = (ArchivingResponse)
    obj_archivingRequest.send("http://www.test.com" + "/adss/ltap");
if (obj_archivingResponse.isSuccessfull()) {
    System.out.println("\nRequest has been processed successfully.");
}
else {
    System.out.println(obj_archivingResponse.getErrorMessage());
}
}

```

4.1.10 Decryption

An encrypted document can be decrypted using ADSS Decryption Service by using following code:

```

//Constructing Decryption request
DecryptionRequest obj_decryptionRequest = new
    DecryptionRequest("samples_test_client", "encrypted-XML.xml", "XML");
obj_decryptionRequest.setProfileId("adss:decryption:profile:001");
obj_decryptionRequest.setRequestId("req-1");
obj_decryptionRequest.setCertificateAlias("samples_tsa_service_certificate");
obj_decryptionRequest.writeTo("../data/encryption/encryptionXML-request.xml");
System.out.println("\nA request has been sent to verify the PDF signature. Waiting
    for response...");
}

```

```

/* Sending the above constructed request to the ADSS server */
DecryptionResponse obj_decryptResponse = (DecryptionResponse)
    obj_decryptionRequest.send(args[1] + "/adss/decryption");
/* Writing response to disk */
obj_decryptResponse.writeTo("../data/encryption/encryptionXML-response.xml");
/* Parsing the response */
if (obj_decryptResponse.isSuccessfull()) {
    System.out.println("\nRequest has been processed successfully.");
}
else {
    System.out.println(obj_decryptResponse.getErrorMessage());
}

```

4.2 Using the .NET API

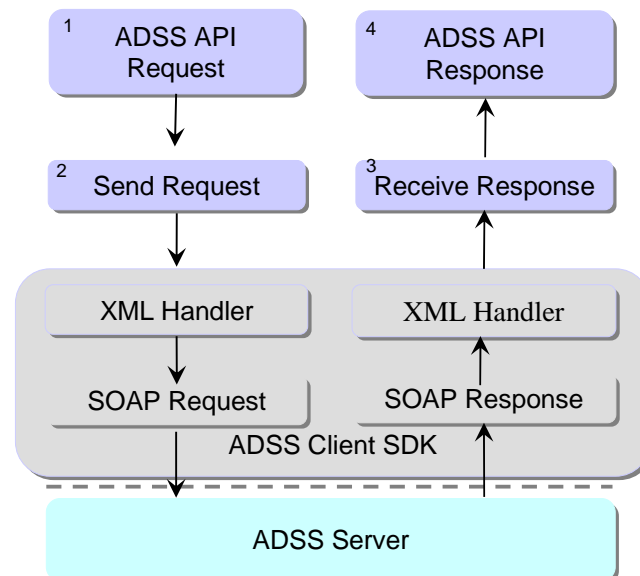


Figure - C# ADSS API program Flow

The building blocks for this flow are outlined below and are demonstrated in detail by the samples provided with the ADSS Client SDK. This contains all the necessary libraries to build a complete ADSS Server integrated business application.

The following sections provide code fragments for coding the ADSS API tasks 1-4. Section 5 describes the XML element structure which is useful for identifying elements of the protocol while studying this code.

Note that the code fragments are not intended to be functioning pieces of code. Sometimes values are read, but not used, for demonstration purposes.

The ADSS Server Admin Manual describes how to configure the ADSS Server with keys, certificates, trust anchors, profiles etc that are required to process business application requests. Also, for the sample applications, a database script is provided to automate the server configuration. These server configurations are required for the sample application's code to work as it is.

4.2.1 Server-side Signing

A signing request is populated with request parameters and a PDF document. Then this request is sent to ADSS Signing service using sent method of Request object. This will create the XML/SOAP

request, send it to ADSS Signing service and receive an XML/SOAP response. PDF is then extracted from the response.

```

/* Constructing request for pdf signing */
PdfSigningRequest obj_signingRequest = new
PdfSigningRequest("samples_test_client", "sample_input_file.pdf");
obj_signingRequest.SetProfileId("adss:signing:profile:001");
obj_signingRequest.SetCertificateAlias("samples_test_signing_certificate");
/* Writing request to disk */
obj_signingRequest.WriteTo("../data/signing/SignPDF-request.xml");
Console.WriteLine("\nA request has been sent to sign the PDF. Waiting for
response...");/* Sending the above constructed request to the ADSS Server */
PdfSigningResponse obj_signingResponse =
(PdfSigningResponse)obj_signingRequest.Send("http://testserver/adss/signing/dsi");
/* Writing response to disk */
obj_signingResponse.WriteTo("../data/signing/SignPDF-response.xml");
/* Parsing the response */
if (obj_signingResponse.IsSuccessful())
{
    obj_signingResponse.PublishDocument("sample_output_file.pdf");
    Console.WriteLine("\nRequest has been processed successfully.");
}
else
{
    Console.WriteLine(obj_signingResponse.GetErrorMessage());
}

```

4.2.2 Client-side Signing

4.2.2.1 Hashing and Document Assembly

See section 3.7 for an application overview of this process. The ADSS server side code performs the hashing and assembly tasks.

A hashing request is populated with various request parameters and a PDF document to be hashed. The XML/SOAP request is delivered to the ADSS server, which processes the PDF. The hash is extracted from the response, and is signed by the application and then added to an assembly request object. An XML/SOAP request is then delivered to the server. The supplied signature and server stored document are assembled by ADSS and sent to the calling application. The signed document is then finally extracted from the assembly response by the business application.

```

byte[] byte_cert = null;
byte[] byte_signature = null;
X509Certificate2 obj_cert = null;
try
{
    obj_cert = new X509Certificate2("../data/signing/adss_default_admin.pfx",
"password");
    byte_cert = obj_cert.GetRawCertData();
}
catch (Exception ex)
{
    Console.WriteLine(ex.StackTrace);
}
/* Constructing request for document hashing */
DocumentHashingRequest obj_documentHashingRequest = new
DocumentHashingRequest("samples_test_client", "sample_input_doc.pdf", byte_cert);
obj_documentHashingRequest.SetProfileId("adss:signing:profile:001");

```

```

obj_documentHashingRequest.OverrideProfileAttribute(DocumentHashingRequest.SIGNING_
REASON, "Testing");
obj_documentHashingRequest.OverrideProfileAttribute(DocumentHashingRequest.CONTACT_
INFO, "Alice");
obj_documentHashingRequest.OverrideProfileAttribute(DocumentHashingRequest.SIGNING_
FIELD, "Signature1");
/* Writing request to disk */
obj_documentHashingRequest.WriteTo("../data/signing/HashAndAssemblyManager-
request.xml");
Console.WriteLine("\nA request has been sent to calculate the document hash.
Waiting for response...");
/* Sending the above constructed document hashing request to the ADSS Server */
DocumentHashingResponse obj_documentHashingResponse =
(DocumentHashingResponse)obj_documentHashingRequest.Send("http://testserver/adss/sig
ning/dhi");
/* Writing response to disk */
obj_documentHashingResponse.WriteTo("../data/signing/HashAndAssemblyManager-
response.xml");
/* Parsing the document hashing response */
if (obj_documentHashingResponse.IsSuccessfull())
{
    Console.WriteLine("\nDocument hash recieved successfully.");
    try
    {
        // Compute the Pkc7 of the hash taken from the response
        ContentInfo contentInfo = new
ContentInfo(obj_documentHashingResponse.GetDocumentHash());
        SignedCms signedCms = new SignedCms(contentInfo);
        CmsSigner cmsSigner = new CmsSigner(obj_cert);
        cmsSigner.IncludeOption = X509IncludeOption.EndCertOnly;
        signedCms.ComputeSignature(cmsSigner, false);
        Byte[] pkc7Message = signedCms.Encode();
        byte_signature = pkc7Message;
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.StackTrace);
    }
}
/* Constructing request for signature assembly */
SignatureAssemblyRequest obj_signatureAssemblyRequest = new
SignatureAssemblyRequest("samples_test_client", byte_signature,
obj_documentHashingResponse.GetDocumentId());
obj_signatureAssemblyRequest.SetProfileId("adss:signing:profile:001");
Console.WriteLine("\nA request has been sent for signature assembling. Waiting
for response...");
/* Sending the above constructed signature assembly request to the ADSS Server */
SignatureAssemblyResponse obj_signatureAssemblyResponse =
(SignatureAssemblyResponse)obj_signatureAssemblyRequest.Send("http://testserver/ads
s/signing/sai");
/* Parsing the signature assembly response */
if (obj_signatureAssemblyResponse.IsSuccessfull())
{
    obj_signatureAssemblyResponse.PublishDocument("sample_output_doc.pdf");
    Console.WriteLine("\nRequest has been processed successfully.");
}
else
{
    Console.WriteLine(obj_signatureAssemblyResponse.GetErrorMessage());
}
}

```

```

}
else
{
    Console.WriteLine(obj_documentHashingResponse.GetErrorMessage());
}

```

4.2.3 Creating Empty Signature Fields

An empty signature field request is populated with request parameters and a PDF document. Then this request is sent to ADSS Signing service using send method of Request object. This will create the XML/SOAP request, send it to ADSS Signing service and receive an XML/SOAP response. The PDF extracted from the response contains a field that was created as an empty signature field and then signed by the server.

```

/* Constructing request for blank signature field(s) creation on the pdf document
*/
EmptySignatureFieldRequest obj_emptySigFieldRequest = new
EmptySignatureFieldRequest("samples_test_client", "sample_input_doc");

//obj_emptySigFieldRequest.SetRequestMode(EmptySignatureFieldRequest.DSS);
obj_emptySigFieldRequest.SetProfileId("adss:signing:profile:005");
obj_emptySigFieldRequest.SetSigningInfo("adss:signing:profile:001",
"samples_test_signing_certificate");
obj_emptySigFieldRequest.OverrideProfileAttribute(EmptySignatureFieldRequest.SIGNIN
G_REASON, "Testing");
/* Writing request to disk */
obj_emptySigFieldRequest.WriteTo("../data/signing/CreateEmptySigFields-
request.xml");
Console.WriteLine("\nA request has been sent to create blank signature(s) on the
PDF. Waiting for response...");
/* Sending the above constructed request to the ADSS Server */
EmptySignatureFieldResponse obj_emptySigFieldResponse =
(EmptySignatureFieldResponse)obj_emptySigFieldRequest.Send("http://testserver/adss/
signing/esi");
/* Writing response to disk */
obj_emptySigFieldResponse.WriteTo("../data/signing/CreateEmptySigFields-
response.xml");
/* Parsing the response */
if (obj_emptySigFieldResponse.IsSuccessfull())
{
    obj_emptySigFieldResponse.PublishDocument("sample_output_doc.pdf");
    Console.WriteLine("\nRequest has been processed successfully.");
}
else
{
    Console.WriteLine(obj_emptySigFieldResponse.GetErrorMessage());
}

```

4.2.4 Verification

The ADSS API supports the verification of PDF signatures, PKCS#7/CMS or XML DigSig formats. The following code verifies a single signature on a PDF document.

```

/* Constructing request for pdf signature verification */
SignatureVerificationRequest obj_signatureVerificationRequest = new
SignatureVerificationRequest("samples_test_client", "PKCS7_Type_RT-C1_S1-1024-C1");
obj_signatureVerificationRequest.SetCertificateQualityLevel("5");

```

```
obj_signatureVerificationRequest.SetSignatureQualityLevel("5");
obj_signatureVerificationRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_BASIC_CONSTRAINTS);
obj_signatureVerificationRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_KEY_USAGE);
obj_signatureVerificationRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_FROM);
obj_signatureVerificationRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_TO);
obj_signatureVerificationRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_ISSUER_NAME);
/* Constructing signature element so that it can be added into the signature verification request */
SignatureInfo obj_signatureInfo = new
SignatureInfo("signature.id.001", "smample_test_input.pkcs7",
SignatureInfo.SIGNED_DOCUMENT_TYPE_PKCS7);
obj_signatureInfo.SetSignatureFormat(SignatureInfo.SIGNATURE_FORMAT_OTHER);
obj_signatureInfo.AddRespondWithItem(VerificationRequest.RESPOND_WITH_BASIC_CONSTRAINTS);
obj_signatureInfo.AddRespondWithItem(VerificationRequest.RESPOND_WITH_KEY_USAGE);
obj_signatureInfo.AddRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_FROM);
obj_signatureInfo.AddRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_TO);
obj_signatureInfo.AddRespondWithItem(VerificationRequest.RESPOND_WITH_ISSUER_NAME);
/* Adding signature element(s) into the signature verification request */
obj_signatureVerificationRequest.AddSignatureInfo(obj_signatureInfo);
/* Writing request to disk */
obj_signatureVerificationRequest.WriteTo("../data/verification/VerifyPKCS7-request.xml");
Console.WriteLine("\nA request has been sent to verify the PDF signature. Waiting for response...");
/* Sending the above constructed request to the ADSS Server */
VerificationResponse obj_verificationResponse =
(VerificationResponse)obj_signatureVerificationRequest.Send(args[1].Trim() +
"/adss/verification/svi");
/* Writing response to disk */
obj_verificationResponse.WriteTo("../data/verification/VerifyPKCS7-response.xml");
/* Parsing the response */
if (obj_verificationResponse.IsSuccessfull())
{
    Console.WriteLine("\nRequest has been processed successfully.");
    Console.WriteLine("\nResponse Id : " +
obj_verificationResponse.GetRequestId());
    Console.WriteLine("Response Type : " +
obj_verificationResponse.GetResponseTypeId());
}
else
{
    Console.WriteLine("\nRequest hasn't been processed successfully.");
    Console.WriteLine("\nError Code : " + obj_verificationResponse.GetErrorId());
    Console.WriteLine("Result Minor : " +
obj_verificationResponse.GetResultMinor());
}
}
```

4.2.5 Certification

Certification requests can be made to the ADSS Server for certificate/key generation, certificate renewal, certificate/key deletion and changing a password on the private key data.

```

/* Constructing request for getting certificate(s) from the ADSS certification
service */
CertificationRequest obj_certificationRequest = new
CertificationRequest("samples_test_client",
CertificationRequest.REQUEST_TYPE_CREATE_CERTIFICATE, "Test Alice");

obj_certificationRequest.SetProfileId("adss:certification:profile:001");
obj_certificationRequest.SetPkcs12Password("password");
obj_certificationRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_CERTI
FICATE);
obj_certificationRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_
12);
obj_certificationRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_
7);
obj_certificationRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_EXPIR
Y_DATE);
obj_certificationRequest.OverrideProfileAttribute(CertificationRequest.SUBJECT_DN,
"CN=" + args[1].Trim());
/* Writing request to disk */
obj_certificationRequest.WriteTo("../data/certification/GenerateCertificate-
request.xml");
Console.WriteLine("\nA request has been sent to get certificate. Waiting for
response...");
/* Sending the above constructed request to the ADSS Server */
CertificationResponse obj_certificationResponse =
(CertificationResponse)obj_certificationRequest.Send("http://testserver/adss/certif
ication/csi");
/* Writing response to disk */
obj_certificationResponse.WriteTo("../data/certification/GenerateCertificate-
response.xml");
/* Parsing the response */
if (obj_certificationResponse.IsSuccessfull())
{
obj_certificationResponse.PublishCertificate("../data/certification/certificate.cer
");
obj_certificationResponse.PublishPKCS12("../data/certification/certificate.pfx");
obj_certificationResponse.PublishPKCS7("../data/certification/certificate.p7b");
    Console.WriteLine("\nRequest has been processed successfully.");
}
else
{
    Console.WriteLine(obj_certificationResponse.GetErrorMessage());
}
}

```

4.2.6 XKMS Validate

XKMS Validate request can be sent to ADSS XKMS Service to get the validation status of a certificate.

```

/* Constructing xkms validate request */
ValidateRequest obj_validateRequest = new
ValidateRequest(ValidateRequest.RESPONDWITH_X509CERT);
obj_validateRequest.SetRequestID("adss:xkms:validaterequest:001");
obj_validateRequest.SetRespondAddress(args[0]);
obj_validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_KEYNAME);
obj_validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_KEYVALUE);
obj_validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_SPKI);
obj_validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_X509CHAIN);

```

```

obj_validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_X509CRL);
obj_validateRequest.SetKeyInfo(Util.Util.ReadFile("sample_cert.cer"));
obj_validateRequest.SetTimeInstant(DateTime.Now);
obj_validateRequest.AddKeyUsage(ValidateRequest.KEYUSAGE_SIGNATURE);
obj_validateRequest.AddUseKeyWith(ValidateRequest.USEKEYWITH_SMIME,
"alice@example.com");
/* Writing request to disk */
obj_validateRequest.WriteTo("../data/xkms/Validate-Request.xml");
Console.WriteLine("\nA request has been sent. Waiting for response...");
/* Sending the above constructed request to the ADSS Server */
ValidateResult obj_validateResult =
(ValidateResult)obj_validateRequest.Send(args[1].Trim() + "/adss/xkms");
/* Writing response to disk */
obj_validateResult.WriteTo("../data/xkms/Validate-Result.xml");
if (obj_validateResult.GetResultMajor().Contains("Success"))
{
    Console.WriteLine("Validate Status : " + obj_validateResult.GetResultMajor());
    Console.WriteLine("Request Id : " + obj_validateResult.GetId());
    Console.WriteLine("List : " + obj_validateResult.GetCertificateChain().Count);
    Console.WriteLine("validate reason " +
obj_validateResult.GetValidReason().Count);
    Console.WriteLine("\nRequest has been processed successfully.");
}
else
{
    Console.WriteLine(obj_validateResult.GetErrorMessage());
}
}

```

4.2.7 LTANS Archive

A document can be securely archived for longer period using ADSS LTANS Service. Client application can create a document archiving request using ADSS Client SDK as follows:

```

/* Constructing archiving request */
ArchivingRequest obj_archivingRequest = new ArchivingRequest("samples_test_client",
ArchivingRequest.SERVICE_TYPE_ARCHIVE, "test_content.dat");
obj_archivingRequest.SetVersion("1.0");
obj_archivingRequest.SetRequestID("Request_001");
obj_archivingRequest.SetPolicyIdentifier("adss:ltan:profile:001");
obj_archivingRequest.SetTransactionIdentifier("trans.id.001");
obj_archivingRequest.SetRespondAddress(args[1]);
obj_archivingRequest.SetNonce("Nonce:001");
obj_archivingRequest.SetRequestTime(new DateTime().ToString());
/* Constructing data string element so that it can be added into the archiving
request */
byte[] byteArrArchiveData = Util.Util.ReadFile(args[0].Trim());
obj_archivingRequest.SetData(byteArrArchiveData);
obj_archivingRequest.SetMetaDataValue("PDF");
/* Writing request to disk */
obj_archivingRequest.WriteTo("../data/ltan/Archiving-Request.xml");
Console.WriteLine("\nA request has been sent. Waiting for response...");
/* Sending the above constructed request to the ADSS Server */
ArchivingResponse obj_archivingResponse =
(ArchivingResponse)obj_archivingRequest.Send("http://testserver/adss/ltap");
/* Writing response to disk */
obj_archivingResponse.WriteTo("../data/ltan/Archiving-Response.xml");
if (obj_archivingResponse.IsSuccessfull())
{

```

```

        Console.WriteLine("\nRequest has been processed successfully.");
        Console.WriteLine("Reference Id : " + obj_archivingResponse.GetReference());
    }
    else
    {
        Console.WriteLine(obj_archivingResponse.GetErrorMessage());
    }
}

```

4.2.8 LTANS Export

An archived document can be retrieved from ADSS LTANS Service by using following code:

```

/* Constructing archiving request */
ArchivingRequest obj_archivingRequest = new ArchivingRequest("samples_test_client",
ArchivingRequest.SERVICE_TYPE_EXPORT, "http://testserver.com");
obj_archivingRequest.SetVersion("1.0");
obj_archivingRequest.SetRequestID("Request_001");
obj_archivingRequest.SetPolicyIdentifier("adss:ltan:profile:001");
obj_archivingRequest.SetTransactionIdentifier("trans.id.001");
obj_archivingRequest.SetRespondAddress("http://testserver.com");
obj_archivingRequest.SetNonce("Nonce:001");
obj_archivingRequest.SetRequestTime(new DateTime().ToString());
/* Set reference id to export the archived data */
obj_archivingRequest.SetReference("123456");
/* Writing request to disk */
obj_archivingRequest.WriteTo("../data/ltan/Export-Request.xml");
Console.WriteLine("\nA request has been sent. Waiting for response...");
/* Sending the above constructed request to the ADSS Server */
ArchivingResponse obj_archivingResponse =
(ArchivingResponse)obj_archivingRequest.Send("http://testserver/adss/ltap");
/* Writing response to disk */
obj_archivingResponse.WriteTo("../data/ltan/Export-Response.xml");
if (obj_archivingResponse.IsSuccessfull())
{
    Console.WriteLine("\nRequest has been processed successfully.");
    Util.Util.WriteToFile(obj_archivingResponse.GetData(), "archive_export.dat");
}
else
{
    Console.WriteLine(obj_archivingResponse.GetErrorMessage());
}
}

```

4.2.9 LTANS Delete

An archived document can be deleted on ADSS LTANS Service by using following code:

```

/* Constructing archiving request */
ArchivingRequest obj_archivingRequest = new ArchivingRequest("samples_test_client",
ArchivingRequest.SERVICE_TYPE_DELETE, "http://testerver.com");
obj_archivingRequest.SetVersion("1.0");
obj_archivingRequest.SetRequestID("Request_001");
obj_archivingRequest.SetPolicyIdentifier("adss:ltan:profile:001");
obj_archivingRequest.SetTransactionIdentifier("trans.id.001");
obj_archivingRequest.SetRespondAddress("http://testerver.com");
obj_archivingRequest.SetNonce("Nonce:001");
obj_archivingRequest.SetRequestTime(new DateTime().ToString());
/* Set reference id to delete the archived data */

```

```

obj_archivingRequest.SetReference("123456");
/* Writing request to disk */
obj_archivingRequest.WriteTo("../data/ltan/Delete-Request.xml");
Console.WriteLine("\nA request has been sent. Waiting for response...");
/* Sending the above constructed request to the ADSS Server */
ArchivingResponse obj_archivingResponse =
(ArchivingResponse)obj_archivingRequest.Send("http://testerver.com/adss/ltap");
/* Writing response to disk */
obj_archivingResponse.WriteTo("../data/ltan/Delete-Response.xml");
if (obj_archivingResponse.IsSuccessfull())
{
    Console.WriteLine("\nRequest has been processed successfully.");
}
else
{
    Console.WriteLine(obj_archivingResponse.GetErrorMessage());
}

```

4.2.10 Decryption

An encrypted document can be decrypted using ADSS Decryption Service by using following code:

```

/* Constructing request for pdf signing */
DecryptionRequest obj_decryptionRequest = new
DecryptionRequest("samples_test_client", "sample_input.xml",
DecryptionRequest.MIME_TYPE_XML);
obj_decryptionRequest.SetProfileId("adss:decryption:profile:001");
obj_decryptionRequest.SetCertificateAlias("samples_test_signing_certificate");
/* Writing request to disk */
obj_decryptionRequest.WriteTo("../data/encryption/Decryption-request.xml");
Console.WriteLine("\nA request has been sent to decrypt the file. Waiting for
response...");
/* Sending the above constructed request to the ADSS Server */
DecryptionResponse obj_signingResponse =
(DecryptionResponse)obj_decryptionRequest.Send("http://testerver/adss/enc");
/* Writing response to disk */
obj_signingResponse.WriteTo("../data/encryption/Decryption-response.xml");
/* Parsing the response */
if (obj_signingResponse.IsSuccessfull())
{
    obj_signingResponse.PublishDocument("sample_output.xml");
    Console.WriteLine("\nRequest has been processed successfully.");
}
else
{
    Console.WriteLine(obj_signingResponse.GetErrorMessage());
}

```

4.3 Code Samples

The following tables explain what each of the provided samples provided in ADSS Client SDK demonstrates when using ADSS web services.

ADSS Service	Sample File Name (In Java and C#)
--------------	-----------------------------------

Signing**SignPDF**

This sample demonstrates how to get a PDF signed using an ADSS Server profile **adss:signing:profile:001**. This example creates a visible signature on the first page of the PDF at the top left corner. Note that the business application can also override the profile attributes in the code and set signing reason, signing location etc. The input and output file path and ADSS Server name are configurable and taken as command line parameters. The sample uses the ADSS Client API's XML mode to communicate with ADSS Signing Service. To achieve optimum performance you can also utilize pure HTTP mode instead of XML mode for all signing interface requests. To do so, you just need to call **setRequestMode(SigningRequest.HTTP)** function of signing API and rest of the code would remain unchanged.

SignPdfDSS

This sample demonstrates how to get a PDF signed using DSS protocol by ADSS Server profile **adss:signing:profile:001**. This example creates a visible signature on the first page of the PDF at the top left corner. Note that the business application can also override the profile attributes in the code and set signing reason, signing location etc. The input and output file path and ADSS Server name are configurable and taken as command line parameters. The sample uses the ADSS Client API's DSS mode to communicate with ADSS Signing Service.

SignPdfHTTP

This sample demonstrates how to get a PDF signed using an ADSS Server profile **adss:signing:profile:001** via a pure HTTP POST request without using ADSS Client API Signing classes. This example creates a visible signature on the first page of the PDF at the top left corner. Note that the business application can also override the profile attributes in the code and set signing reason, signing location etc. The input and output file path and ADSS Server name are configurable and taken as command line parameters.

CreateEmptySigFields

This sample demonstrates how to get ADSS Server to generate empty signature fields on the top left corner of all the pages of a PDF. It then also demonstrates a request for ADSS Server to sign one of the generated blank signature fields using the ADSS profile **adss:signing:profile:003**. The signature is a visible signature created on the first page of the PDF at the top left corner position - created using ADSS profile **adss:signing:profile:001**. Note that the business application can also override the signing profile attributes in the code and set signing reason, signing location etc. The input and output PDF file path and ADSS server name are configurable and taken as command line parameters.

SignPDFUsingPreferences

This sample demonstrates how an application can send a PDF file to an ADSS Server and have it signed using the XML signature appearance preferences set within the ADSS Server signing policy. XML preferences enable signatures to be created on multiple pages, in custom locations and with custom appearances. In this example all pages are signed. The location of the signature is defined to be slightly left of centre. The ADSS Server signing profile used in the sample is **adss:signing:profile:001**. See the ADSS Server Admin Guide to find out more about the capabilities of XML preferences and how to configure them. The input and output PDF file path and ADSS Server name are configurable and taken as command line parameters.

HashAndAssemblyManager

This sample demonstrates how an application can send a PDF to an ADSS Server and have it signed by a generated user key. In this example the ADSS signing profile **adss:signing:profile:001** will be used, although in this case the ADSS Server will not sign the PDF, rather it creates an empty signature field and returns a hash of the PDF document. The sample then generates a key-pair and a self signed certificate, signs the hash using this key, generates and sends a PKCS#7 back to ADSS for assembly. The input and output PDF file path and ADSS server name are

	<p>configurable and taken as command line parameters.</p> <p><u>SignPDFUsingIEKeyStore</u></p> <p>This sample demonstrates how an application can sign the PDF using the certificate from an IE Key Store. The user can select the certificate by configuring the filter on the basis of Subject DN, Issuer DN, Expiry Date, Key Usages and Signature Algorithm. This sample will select the first certificate if more than one certificate is returned after applying the filter. In this example the ADSS signing profile adss:signing:profile:001 will be used, although in this case the ADSS Server will not sign the PDF, rather it creates an empty signature field and returns a hash of the PDF document. The sample then signs the hash using this certificate selected from the IE Key store and sends a PKCS#7 back to ADSS for assembly. The input and output PDF file path and ADSS server name are configurable and taken as command line parameters.</p>
Verification	<p><u>VerifyPDF</u></p> <p>This sample demonstrates how an application can send a signed PDF to an ADSS Server and have the embedded signatures verified. The input PDF file path and ADSS server name are configurable and taken as command line parameters.</p> <p><u>VerifyPdfDSS</u></p> <p>This sample demonstrates how an application can send a signed PDF to an ADSS Server and have the embedded signatures verified using DSS interface. The input PDF file path and ADSS server name are configurable and taken as command line parameters.</p> <p><u>VerifyXMLEnveloped</u></p> <p>This sample demonstrates how an application can send a signed XML file to an ADSS Server and have the embedded signatures verified. The input XML file path and ADSS server name are configurable and taken as command line parameters. The XML must only have one signature.</p> <p><u>VerifyPKCS7</u></p> <p>This sample demonstrates how an application can send a PKCS#7 signed object to an ADSS Server and have its signature verified. The input PKCS#7 file and ADSS server name are configurable and taken as command line parameters. The PKCS#7 must only have one signature. CMS signatures can also be sent using this sample.</p> <p><u>VerifyCertificate</u></p> <p>This sample demonstrates how an application can send an X509 certificate to an ADSS server and have the certificate validated and its trust checked. The input certificate file and ADSS server name are configurable and taken as command line parameters.</p> <p><u>Historical Verification</u></p> <p>Similar program samples to those above are present inside java\src\com\ascertia\adss\samples\verification\historical. These perform historical verification and validation. In these samples the signature and certificate is verified and validated at a date/time in the past.</p>
Certification	<p><u>GenerateCertificate</u></p> <p>This sample demonstrates how an application can send a Certification request to an ADSS Server and receive the PKCS#12, Certificate, PKCS#7 etc. The ADSS server name is configurable and taken as command line parameter.</p>
XKMS	<p><u>XkmsValidate</u></p> <p>This sample demonstrate how ADSS Client SDK can be used to compose and send a certificate validation request to ADSS Server</p>

LTANS	<p><u>LtanArchive</u></p> <p>This sample demonstrates how a document can be securely archived at ADSS Server for a long period. ADSS Client SDK uses LTANS archive command to store the document at server. This program takes input document path as command line parameter and returns the reference Id of the stored archive object</p> <p><u>LtanExport</u></p> <p>This sample demonstrates how an archived document can be exported out of ADSS database through LTANS export command. This program takes exported file path and reference Id of stored archive as command line parameters</p> <p><u>LtanDelete</u></p> <p>This sample demonstrates how an archived document can be deleted from ADSS database through LTANS delete command. This program takes reference Id of stored archive as command line parameters</p>
--------------	---

5 ADSS Server Web Services XML Schemas

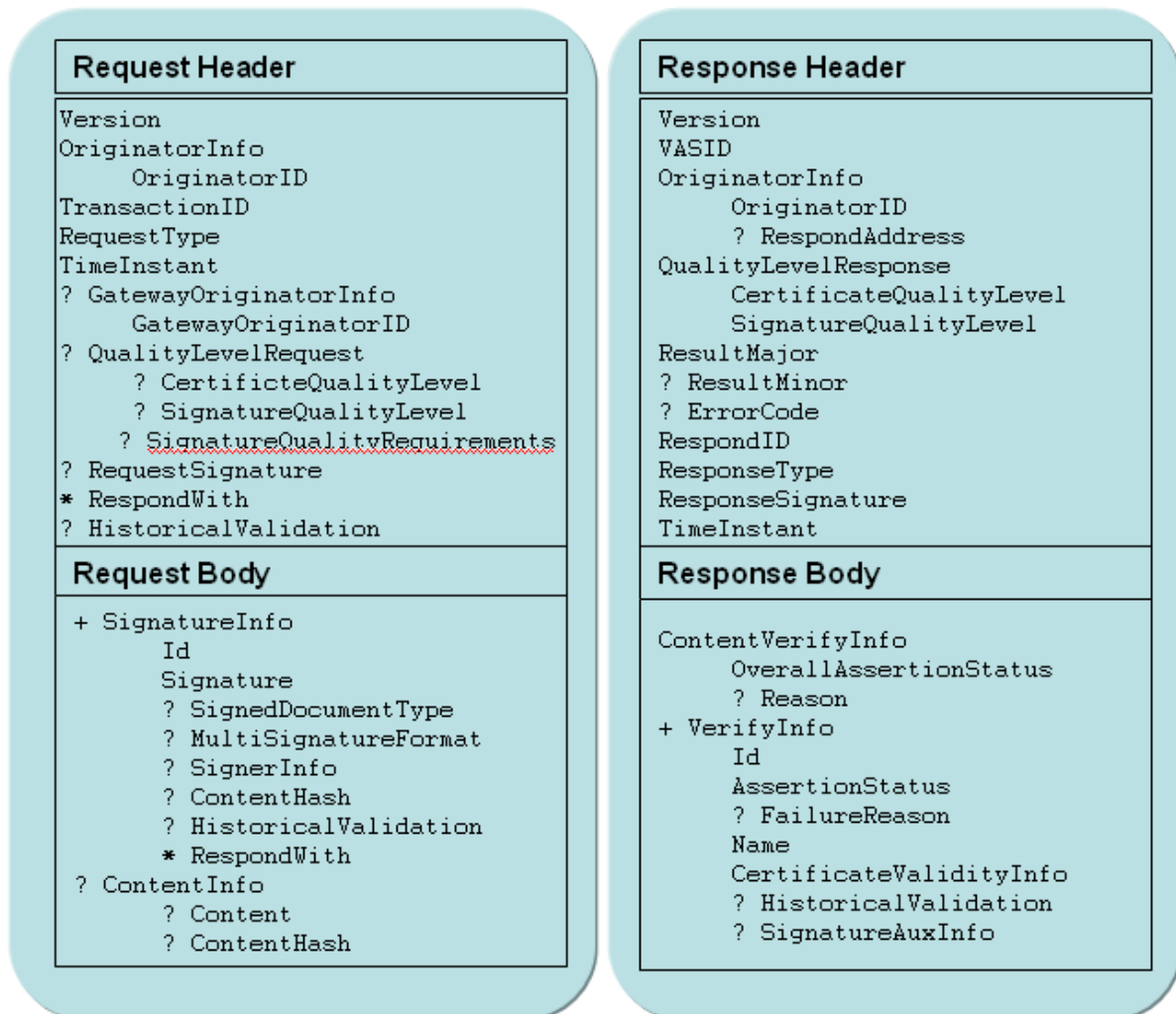
5.1 Verification Web Service

The Verification web service XML schema is a custom XML schema that enables business applications to send signatures to be verified and receive a response that contains the status of those signatures. In addition the application may optionally receive evidential information by identifying what should be returned in the request message e.g. RespondWith: CRL, OCSP, Public Key Value, etc.

The same Verification Web service XML schema also handles certificate verification (validation) requests and responses. The schema file **adss_verifymodule.xsd** contains the signature / certificate verification schema. All requests and response related to verification MUST comply with this schema. The ADSS Server rejects any request that do not comply.

The verification schema can be best understood by logically splitting it into two elements, the Header and the Body. The Header contains the basic information regarding the entity sending the request, request identifiers etc and the Body contains the signature/certificate to be verified.

Furthermore the schema defined in **adss_verifymodule.xsd** caters for both Signature verification and Certificate validation. The XML structures for both types of requests/responses are very much the same with little differences. The signature verification schema has the following structure:

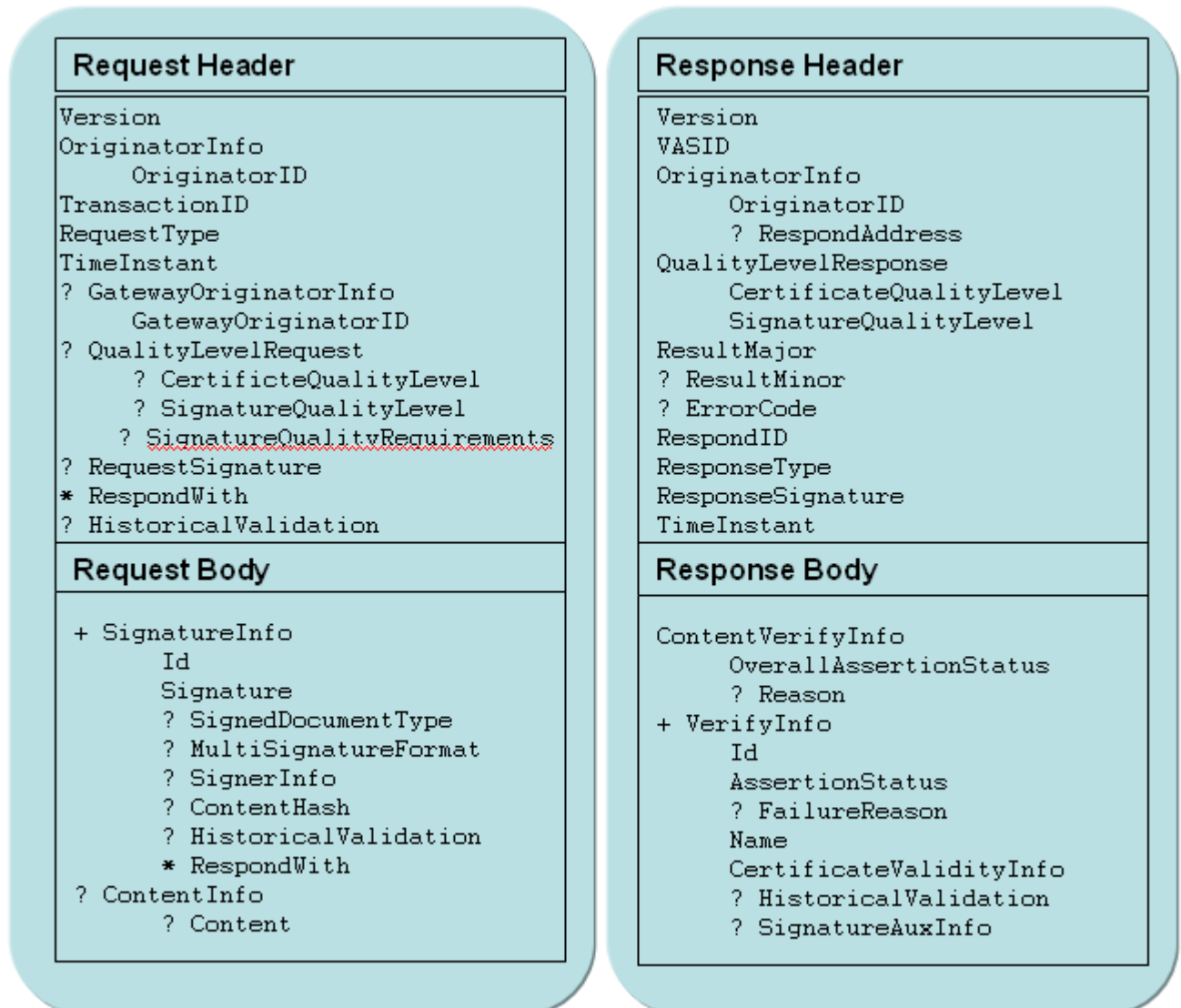


The key for the symbols used in the schema above is shown below:

? Zero or one occurrence
 + One or more occurrence
 * Zero or more occurrence
 For the remaining only one occurrence

As can be seen both request and response have a header and a body. Each of these is explained in further detail in the sections that follow.

The high-level structure of the Certificate Validation Schema is similarly described:



As before the key for the symbols used in the schema above is shown below:

? Zero or one occurrence
 + One or more occurrence
 * Zero or more occurrence
 For the remaining only one occurrence

5.1.1 Request header for signature verification and certificate validation

The header is the same for both signature verification and certificate validation requests. It is important to note that if the request is for certificate validation then the term "signature" means the processing of the signature on the Certificate to be validated.

ADSS Gateway is mentioned in the information below. This is a special version of ADSS Server used at remote sites to provide privacy for documents that need to have signatures verified. Privacy is enforced by having ADSS Gateway process the document locally, stripping out the signature objects, sending these for verification and then returning the received response to the client as it is. Separate documentation describes the ADSS Gateway.

All elements for the request are under a top level VerifyRequest XML element.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
Version (M) - (String)	Indicates the version of the validation interface. The default value is 1.0 and may change in future versions if the protocol is updated.
OriginatorInfo (M) - (Container)	Contains details of the business application that is sending the request. Also may contain the signature document name to be verified (When a remote ADSS Gateway is used in asynchronous Mode).
	OriginatorID (M) - (String) Contains a unique ID for originator (or the document name when using remote ADSS Gateways). If it contains the client's unique ID then this must be registered within the ADSS Server and if client-authenticated SSL is turned on within ADSS then it must match the Common Name within the Client's SSL Certificate. If an ADSS Gateway is sending the request on behalf of an asynchronous client e.g. a client communicating using email with an ADSS Gateway then this field automatically contains the document name sent in the email.
	RespondAddress (O) - (String) Contains an address e.g. email address or web server address to support asynchronous processing of requests i.e. the response is sent back some time later without requiring the client to remain connected. This field is currently only used by ADSS Gateway, when it is accepting asynchronous requests from clients so that it can respond in due course. ADSS Server does not process this element.
GatewayInfo (O) - (Container)	Only generated when an ADSS Gateway is sending a request to ADSS Server.
	GatewayOriginatorID (M) String Contains a unique ID for the ADSS Gateway. This must be registered within ADSS Server and if client-authenticated SSL is enabled then this value must match the ADSS Gateway client SSL Client Certificate CN. This field is populated when the request is sent from Gateway asynchronously.
TransactionID (M) - (String)	Contains a unique ID for the transaction. It is up to the client to set this value. ADSS Gateway generates Transaction IDs as GatewayOriginatorID + "_" + 80 bit random number e.g. Gateway1_45454556998f98989934343433.
QualityLevelRequest (O) - (Container)	Identifies the minimum Quality Levels for the whole request. All signatures will need to meet these minimum values. If a particular quality level is not defined in the request message then the default value is used from this client's account configured within ADSS Server.
	CertificateQualityLevel (O) - (String) Contains the minimum certificate quality level that the <u>signer's</u> certificate must meet. Must be in the range 0-10
	SignatureQualityLevel (O) - (String) Contains the minimum signature quality level that the <u>signature being verified</u> must meet (allowable range is 0-10)

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
SignQualityRequirements (O) (Container)	KeyUsage (O) - (String) - (Multiple) Contains required KeyUsage, if any, such as ContentCommitment, DigitalSignature, etc as required by the business application. If populated, then ADSS Server ensures that the signer certificate has them. The values will be in accordance with RFC 3647. Multiple XML elements of KeyUsage can exist each having a single KeyUsage value e.g. DigitalSignature etc to check against.
RequestType (M) - (Enumeration)	Contains the type of Request, i.e. Signature Verification (SV) or Certificate Validation (CV). Permitted values are thus: SV and CV
RequestSignature (O) - (Container)	Contains an XML DSig based Signature on the entire Request XML. ADSS Server policies define whether the request message must be signed or not.
TimeInstant (M) - (DateTime)	Contains the date/time when the message was created. E.g. 2007-02-27T18:36:33.171. Zulu time must be used so if the client is in a non GMT zone then the date shift must be appended e.g. 2007-02-27T18:36:33.171+02:00
HistoricalValidation (O) - (Container)	Contains the date-time that should be used as the basis for performing historical signature verification, i.e. verify the status of the signature and certificate at this date-time. This can also be set for per signature basis where each signature is to be verified at a different date-time e.g. 2007-02-27T18:36:33.171+02:00
RespondWith (O) - (String) - (Multiple)	Provide details of the data required back from ADSS Server e.g. the actual content which was signed, calculated signature quality, OCSP, CRL etc. RespondWith can be set at an overall level for all signatures. RespondWith can also be set separately for each signature. The values which can be returned are SignatureQualityLevel, CertificateQualityLevel, Content, SignHash, ContentHash, HashAlgorithm, X509CertificateChain, OCSP, X509CRL, SKI, KeyValue, Timestamp, KeyUsage, ExtendedKeyUsage, BasicConstraints, ValidFrom, Valid To, IssuerName, CertificateSerialNumber, CRLUrl, and CRLNumber. A detailed explanation for each of these is defined below in the request.

5.1.2 Request body for signature verification

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
SignatureInfo (M) – (Container) (Multiple)	Represents each signature inside the request. SignatureInfo may be present more than once in a request to support multiple signature verification per request. If multiple occurrences of SignatureInfo are present then ContentInfo's Content element should be populated – see below. Note that ADSS Server only supports multiple occurrences of SignatureInfo's ContentHash via ADSS Gateways. If the Content is within the SignatureInfo's Signature element i.e. it is data with an attached signature then there can't be more than one SignatureInfo element.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values

	<p>Id (M) – (ID) Contains a unique Identifier e.g. “signature-1” for each Signature present inside the SignatureInfo. Each Signature must contain a unique identifier. The identifier is returned in the response.</p>
	<p>Signature⁴ (M) – (Base64/XML) Contains the signature to be verified. This element contains XML if an XML based signature is sent for verification. It contains Base64 data if PDF, PKCS#7/CMS or S/MIME signatures are sent for verification.</p>
	<p>SignerInfo (O) - (String) The signer's certificate or a reference to it. This must be the certificate associated with the private key used to create the signature. In future if the signature itself contains an embedded signer certificate e.g. within a PKCS#7 /CMS signature, then if present this will be used first before using a SignerInfo provided certificate.</p>
	<p>SignedDocumentType (O) – (String) Defines the type of signature to be verified. Although optional, this information reduces the load on ADSS Server since this does not need to be determined. Possible values are:</p> <ul style="list-style-type: none"> → PDF → XML → PKCS#7 → CMS → S/MIME → Other (Not yet supported) <p>If absent or “Other” is specified then ADSS Server attempts to intelligently determine the signature type. Read the ADSS Server Supported Signature Formats document to understand more about signature types.</p>
	<p>MultiSignatureFormat (O) - (String) If the data has been signed multiple times, this element identifies the format used for applying multiple signatures. Possible values are:</p> <ul style="list-style-type: none"> → A → C → Other (Not yet supported) <p>If absent or “Other” is used ADSS will intelligently determine the signature format (see the ADSS Server Supported Signature Formats document).</p>
	<p>HistoricalValidation (O) – (Container) Contains a date-time reference to enable the historical verification of signatures. This is a per signature element and over-rides the overall Historical Validation element if it is present in the Request Message Abstract. e.g. 2007-02-27T18:36:33.171+02:00</p>

⁴ With ADSS Server a Signature doesn't mean a raw PKCS#1 data, it could also mean a PKCS#7/CMS, S/MIME, PDF, or XML file that contains a PKCS#1 signature. Currently PKCS#1 is not supported

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	<p>Content Hash (O) - (Container) Contains:</p> <ul style="list-style-type: none"> → Hash (base64) → Hashing algorithm (String) <p>This element is used when the signed content is not sent, perhaps because of confidentiality reasons. The Hash Algorithm is a string denoting the algorithm used e.g. SHA1, sha256WithRSAEncryption, sha512WithRSAEncryption etc. See section 5.11 in the developer's guide for more information. This is currently only supported for requests sent from an ADSS Gateway.</p>
	<p>RespondWith (O) - (String) (Multiple) Specifies those items which MUST be returned within the response from the ADSS Server. If for any reason a particular RespondWith item can't be returned then N/A is returned instead.</p> <p>Note: either the Request Header's RespondWith element can be used or if different RespondWith items are needed per signature then use the Request's Body RespondWith item. Note that the former is the default and the latter takes precedence.</p>
	<p>SignatureQualityLevel (O) – (Boolean) If set to true then the calculated Signature quality level is returned. Values ranges from 0-10</p>
	<p>CertificateQualityLevel (O) – (Boolean) If set to true then the calculated Certificate quality level is returned. Values ranges from 0-10</p>
	<p>Content (O) – (Boolean) If set to true then the plaintext message (unsigned) is returned.</p> <p>This might be originally sent in the request in the ContentInfo element OR extracted from the signed object only if it is an attached signature. The returned value is Base64 encoded.</p>
	<p>ContentHash (O) – (Boolean) If set to true then the hash of the request Content is returned using Base64 encoding.</p>
	<p>HashAlgorithm (O) – (Boolean) If set to true then the hash algorithm used to hash the content is returned.</p> <p>An example returned value in the response is sha1WithRSAEncryption. Other possible values are: sha256WithRSAEncryption, sha512WithRSAEncryption etc., see section 5.11 for more details.</p>
	<p>X509CertificateChain (O) – (Boolean) If set to true then the chain of the validated signer's certificate is returned including the trusted CA using Base64 encoding.</p>
	<p>OCSP (O) – (Boolean) If set to true then the OCSP response that validated the signer's certificate is returned using Base64 encoding.</p>
	<p>X509CRL (O) – (Boolean) If set to true then the current CRL used to validate the signers certificate is returned using Base64 encoding.</p>
	<p>SKI (O) – (Boolean) If set to true then the SubjectPublicKey Identifier of the Signer's certificate is returned using Base64 encoding.</p>
	<p>KeyValue (O) – (Boolean) If set to true then the Signer's Public Key is returned using Base64 encoding</p>
	<p>KeyUsage (O) – (Boolean) If set to true then all values inside KeyUsage of the signer's certificate are returned as a formatted string e.g. digitalSignature nonRepudiation keyEncipherment dataEncipherment keyAgreement</p>
	<p>ExtendedKeyUsage (O) – (Boolean) If set to true then all values inside ExtendedKeyUsage of the signer's certificate are returned as a formatted string e.g. sslClientAuthentication sslServerAuthentication</p>

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	BasicConstraints (O) – (Boolean) If set to true then the basic constraints set in the signer's certificate is returned as a formatted string e.g. Type=End Entity or Certificate Authority. Path Length constraint is also returned if present.
	ValidFrom (O) – (Boolean) If set to true then the Valid From date/time of the signer's certificate is returned as a formatted string, e.g. Mon Jun 30 16:44:47 GMT+01:00 2005
	Valid To (O) – (Boolean) If set to true then the Valid To date/time of the signer's certificate is returned as a formatted string. e.g. Mon Jun 30 16:44:47 GMT+01:00 2007
	IssuerName (O) – (Boolean) If set to true then the Issuer Distinguish Name of the signer's certificate is returned as a formatted string e.g. CN=Test L2 CA1,OU=SQA,O=Ascertia,C=GB
	CertificateSerialNumber (O) – (Boolean) If set to true then the serial number of the signer's certificate is returned as an integer, e.g. 24735
	CRLNumber (O) – (Boolean) If set to true then CRL number of the CRL used to check the revocation of signer certificate is returned as integer
	CRLUrl (O) – (Boolean) If set to true then URL from where the CRL was downloaded is returned as a string
	LongTerm (O) – (Boolean) If set to true then it converts the provided signature to a long term signature
ContentInfo (M) - (Container)	<p>Content (O) – (Base64, URI, XML) Contains original message that was signed (if not included with signature). The Content must be in Base64 format. Note that only signatures related to same Content can be sent in the request.</p> <p>ContentHash (O) – (Container) Contains</p> <p style="padding-left: 20px;">Hash (base64); hash of original Content</p> <p style="padding-left: 20px;">Hashing algorithm (String)</p> <p>This is used in cases where the end-entity / relying party does not wish to send the content, e.g. for confidentiality reasons. The Hash Algorithm will be string format identifier e.g. sha256WithRSAEncryption, sha512WithRSAEncryption etc</p>

5.1.3 Request body for certificate validation

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
CertificateInfo (M) - (Container)	Represents each certificate inside the request. Multiple instances of CertificateInfo is not currently supported
	Id (M) - (ID) Contains a unique Identifier e.g. "cert-1" for the certificate sent. This identifier will then be returned in the response
	CertificateType (M) (String) Contains the type of certificate which was sent, e.g. X509
	CertificateToValidate (M) - (Container) (String) Contains a reference to the signer's certificate in base64 format.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values

	<p>HistoricalValidation (O) - (Container) Contains date and time information if historical validation of a certificate is required. This is per certificate Historical Validation element and takes precedence over any overall Historical Validation element if present in the Request Message Abstract. e.g. 2007-02-27T18:36:33.171+02:00</p> <p>RespondWith (O) - (Container) Specifies those items which MUST be returned within the response from the ADSS Server. If for any reason a particular RespondWith item can't be returned then N/A is returned instead.</p> <p>SignatureQualityLevel (O) - (Boolean) If set to true then the calculated quality level of the certificate's signature is returned in the range 0-10</p> <p>CertificateQualityLevel (O) - (Boolean) If set to true then the calculated quality level of the certificate is returned in the range 0-10</p> <p>Content (O) - (Boolean) If set to true then the contents of the certificate. Are returned as Base64 encoded data</p> <p>ContentHash (O) - (Boolean) If set to true then the hash of the certificate is returned as Base64 encoded data</p> <p>HashAlgorithm (O) - (Boolean) If set to true then the hash algorithm used to hash the content is returned as a formatted string, e.g. sha1WithRSAEncryption or sha256WithRSAEncryption, sha512WithRSAEncryption etc., see section 5.11 for further details.</p> <p>X509CertificateChain (O) - (Boolean) If set to true the chain of the validated certificate including the trusted CA is returned as Base64 encoded data</p> <p>OCSP (O) - (Boolean) If set to true then the OCSP response used to validate the target certificate is returned as Base64 encoded data</p> <p>X509CRL (O) - (Boolean) If set to true then the CRL used to validate the target certificate is returned as Base64 encoded data</p> <p>SKI (O) - (Boolean) If set to true then the SubjectPublicKey Identifier of the target certificate is returned as Base64 encoded data</p> <p>KeyValue (O) - (Boolean) If set to true, returns the sent certificate's Public Key Returned value in response is Base64 encoded</p> <p>KeyUsage (O) - (Boolean) If set to true then all values inside KeyUsage of the target certificate are returned value as a formatted string e.g. digitalSignature nonRepudiation keyEncipherment dataEncipherment keyAgreement</p> <p>ExtendedKeyUsage (O) - (Boolean) If set to true then all values inside ExtendedKeyUsage of the target certificate are returned as a formatted string e.g. sslClientAuthentication sslServerAuthentication</p> <p>BasicConstraints (O) - (Boolean) If set to true then the basic constraints set in the target certificate are returned as a formatted string, e.g. Type=End Entity or Certificate Authority</p>
--	--

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	<p>ValidFrom (O) - (Boolean) If set to true then the Valid From date / time of the target certificate is returned as a formatted string e.g. Mon Jun 30 16:44:47 GMT+01:00 2005</p> <p>ValidTo (O) - (Boolean) If set to true then the Valid To date / time of the target certificate is returned as a formatted string e.g. Mon Jun 30 16:44:47 GMT+01:00 2007</p> <p>IssuerName (O) - (Boolean) If set to true then the Issuer Distinguish Name of the target certificate is returned as a formatted string e.g. CN=Test L2 CA1,OU=SQA,O=Ascertia,C=GB</p> <p>CertificateSerialNumber (O) - (Boolean) If set to true then the serial number of the target certificate is returned as an integer value, e.g. 23785</p> <p>CRLNumber (O) - (Boolean) If set to true then CRL number of the CRL used to check the revocation of the certificate is returned as integer</p> <p>CRLUrl (O) - (Boolean) If set to true then URL from where the CRL was downloaded is returned as a string</p>

5.1.4 Response header for signature verification and certificate validation

All elements for the request are under a top level VerifyResponse XML element.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
Version (M) - (String)	Indicates the version of the validation interface. The default value is 1.0 and may change in future versions if the protocol is updated.
VASID (M) - (String)	Contains the unique ID for the ADSS Server that signed the response. This is the Common Name contained within the ADSS Server's SSL certificate used for establishing mutual Authenticated SSL. If ADSS does not have an SSL certificate then VASID is ADSS friendly name configured in ADSS. See ADSS Server Admin Manual for more details.
OriginatorInfo (M) - (Container)	This is the same value as provided in the request. This is typically used by ADSS Gateway to store the document name and the end-users email address. When the response is received by ADSS Gateway these values are used to respond to the client.
	OriginatorID (Y) (String) Contains the same value sent in the request
	RespondAddress (O) (String) Contains the same value sent in the request
QualityLevelResponse (M) - (Container)	This will either contain the values set in the original request OR the default values taken from the client's configurations held in the ADSS Server. This is typically used for information purposes, e.g. to show the minimum quality levels used during the verification process. Note: The actual calculated value for the signature or certificate quality level would be set in per signature XML element of the response.
	CertificateQualityLevel (M) (String) In the range 0-10
	SignatureQualityLevel (M) (String) In the range 0-10

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
ResultMajor (M) (ResultMajorEnum)	- Specifies a code that identifies whether the request was processed successfully or if it failed, the possible values are: <ul style="list-style-type: none"> → Success: ADSS Server processed the request → VersionMismatch: Unsupported version number sent by the requestor → Sender: A problem was found in the request → Receiver: A problem occurred at the ADSS Server
ResultMinor (O) (ResultMinorEnum)	- Specifies a code that identifies why the failure occurred. Populated only when the ResultMajor is not "Success". Permitted values are: <ul style="list-style-type: none"> → Incomplete: Request is not completed according to the specification → TryLater: The server is busy → SigRequired: There is no signature in the request → Failure: Not yet supported → Refused: Not yet supported → NoAuthentication: The requestor can't be authenticated → MessageNotSupported: The format is not supported → TimelInstantOutOfRange: The TimelInstant entered by the client was not valid
ErrorCode (Integer)	(O) This contains the high level error code. Following error codes are supported: <ul style="list-style-type: none"> 2000: Transaction ID length has exceed the supported length 2010: Originator ID authentication failure 2020: Request contains invalid version number 2030: Originator ID is either inactive or access to the requested service is not allowed 2040: Request does not contain signature and content 2050: Requested signature type is not supported 2060: Request is not according to schema
ResponseID (String)	(M) - This is set to the TransactionID received in the request.
ResponseType (String)	(M) - Contains type of Response, i.e. Signature Verification or Certificate Validation. Permitted values are: SV and CV
ResponseSignature (M) - (Container)	Contains the ADSS Signature on the entire Response XML DSig format signature (signed using the Verification Response Signing key set within the ADSS configuration). The signing certificate is provided in the XML DSig
TimelInstant (Date)	(M) - Contains the date/time when response message was formed E.g. 2007-02-27T18:36:33.171. If the client is in a non-GMT zone then the date shift is also appended e.g. 2007-02-27T18:36:33.171+02:00

5.1.5 Response body for signature verification and certificate validation

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
ContentVerifyInfo (M)	Contains details of the signatures received/identified

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
<p>- (Container)</p>	<p>OverallAssertionStatus (M) - (String) Provides the overall result of the signature verification(s) of all signature(s) sent in the request. The possible values are:</p> <ul style="list-style-type: none"> → Trusted: For all signatures sent in the request <ul style="list-style-type: none"> a) Signature was verified successfully b) Signer certificates were trusted c) Signature quality/certificate quality was equal to or above the minimum d) Certificate status was good → NotTrusted: If any one of the above failed for any of the signatures → Indeterminate: If the signature can't be verified because the Content or Content Hash can't be found for a signature or the revocation status is unknown <p>The Content may be provided in the ContentInfo OR within the signed object.</p> <p>Reason (O) - (String) When the OverallAssertionStatus is Not Trusted contains the list of all Signature IDs that were not valid. Details of why they were invalid can be found in each signature response area. E.g. Signature-1, Signature-2, Signature-3.</p>
<p>VerifyInfo (M) - (Container) (Multiple)</p>	<p>Represents each signature processed by ADSS.</p> <p>Normally the number of VerifyInfo elements would be same as the number of SignatureInfo in the request. However this may change if the signature in the SignatureInfo is either Type C or a PDF or XML containing one or more signatures.</p> <p>Id (M) - (String) Each individual signature is given a unique ID.</p> <p>The format is: SignatureInfo > Id taken from the request + “-“+ n where n is an incrementing number per signature-document. e.g.</p> <p>SignatureInfo >Id in request is 'sign-1', VerifyInfo >Id will be sign-1-1</p> <p>Example 1: If Two Type-C signatures (having Id sign-1 and sign-2 in SignatureInfo >Id) are sent that result in identifying 3 more signatures each., then the Id in each VerifyInfo will be:</p> <pre> sign-1-1 sign-1-2 sign-1-3 sign-2-1 sign-2-2 sign-2-3 </pre> <p>Example 2: A PDF containing 3 signatures, SignatureInfo >Id in request is 'PDF', the Id in each VerifyInfo will be</p> <pre> PDF-1 PDF-2 PDF-3 </pre> <p>AssertionStatus (M) - (String) Contains status information for a particular signature. The status values returned can be:</p> <ul style="list-style-type: none"> Valid: Signature processed okay Invalid: An error occurred while processing the signature. Details can be found within FailureReason. Indeterminate: Revocation status could not be found or is unknown

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	<p>FailureReason (O) - (String) Contains error codes that explain why signature verification failed.</p> <p>One individual signature may fail for a number of reasons; (a) invalid Certificate Quality, (b) invalid Signature Quality, (c) Certificate Expired or (d) Certificate Revoked, etc. The returned values can be:</p> <p>1000: Invalid Signature Format 1001: Unsigned signed document found e.g. a PDF, XML with no signatures 1010: No valid X509 based Signer Certificate Found 1020: Path Building Failed 1030: Path Validation Failed due to certificate expiry 1040: Path Validation Failed as revocation status of the certificate is REVOKED 1050: Path Validation Failed as revocation status of the certificate is UNKNOWN 1051: CRL for the CA is marked PENDING and it is configured not to use PENDING CRLs 1060: Path Validation Failed due to Signature Verification Failure 1061: Content is not available to verify the signature 1070: Certificate Quality Level is not acceptable 1080: Signature Quality Level is not acceptable</p>
	<p>Name (M) - (String) Contains the Distinguished Name within the signer's certificate associated with the signature. This may be "N/A" if it cannot be found for some reason.</p>
	<p>CertificateValidityInfo (M) - (String) Contains details of the certificate and validity information of the certificate associated with the signature</p> <p>Certificate (M) - (String): Signer's Certificate in Base64 encoded value</p> <p>ThisUpdate (M) - (String): The time at which the status of the certificate is known to be correct. e.g. 2007-02-26 19:13:02.0</p> <p>NextUpdate (M) - (String): The time at which fresh information will be available about the status of the certificate. e.g. 2007-02-26 21:13:02.0</p> <p>ThisUpdate/Next update dates and times are taken from the CRL / OCSP Response ThisUpdate</p>
	<p>HistoricalValidation (O) - (Container) Contains date and time information if historical validation of a certificate was required.</p> <p>e.g. 2007-02-27T18:36:33.171+02:00</p>
	<p>SignatureAuxInfo (O) - (Container) Contains RespondWith element(s) that were asked for in the request</p>
	<p>SignatureQualityLevel (O) - (String) Contains the calculated Signature quality level. Value ranges from 0-10. Note: the actual calculated value for the signature or certificate quality level is always returned if requested by the client application or Relying Party.</p> <p>CertificateQualityLevel (O) - (String) Contains the calculated Certificate quality level. Value ranges from 0-10. Note: In future releases of ADSS Server, if the Assertion status is Invalid or Indeterminate then the actual calculated value for the signature or certificate quality level is always returned regardless of whether it is requested by the client application or Relying Party.</p>
	<p>Content (O) - (String) Contains the plaintext message (unsigned). This data</p>

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values

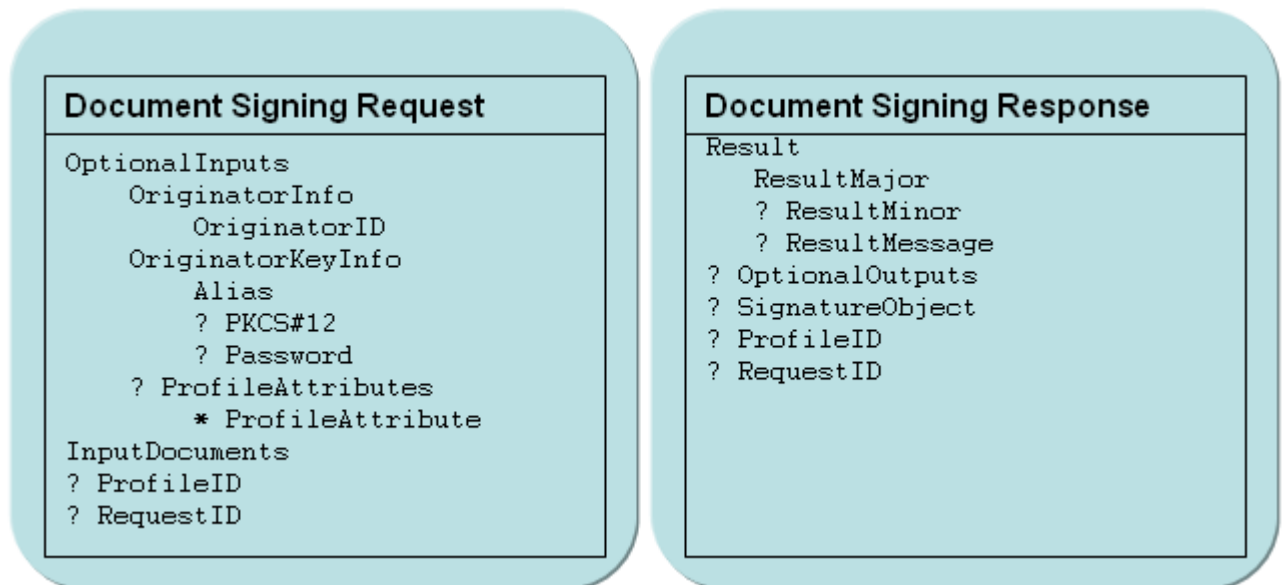
	<p>may be (a) as originally sent in the request in the ContentInfo element, OR (b) as extracted from within an attached signature. The returned value is Base64 encoded.</p> <p>ContentHash (O) - (String) Contains the hash of the Content found in the request Returned value - Base64 encoded.</p> <p>HashAlgorithm (O) - (String) Identifies the hash algorithm used to hash the content, e.g. sha1WithRSAEncryption, sha256WithRSAEncryption, sha512WithRSAEncryption etc. See section 5.11 for further details.</p> <p>X509CertificateChain (O) - (String) Contains the chain of the validated certificate including the trusted CA. The returned value is Base64 encoded.</p> <p>OCSP (O) - (String) Contains the OCSP response used for certificate validation - Base64 encoded.</p> <p>X509CRL (O) - (String) Contains the CRL used for certificate validation - Base64 encoded.</p> <p>SKI (O) - (String) Contains the SubjectPublicKey Identifier of the Signer's certificate - Base64 encoded</p> <p>KeyValue (O) - (String) Contains the Signer's Public Key - Base64 encoded.</p> <p>KeyUsage (O) - (String) Contains all the values inside the KeyUsage of the signer's certificate. The returned data is a formatted string e.g. digitalSignature nonRepudiation keyEncipherment dataEncipherment keyAgreement</p> <p>ExtendedKeyUsage (O) - (String) Contains all the values inside the ExtendedKeyUsage of the signer's certificate. The returned value is a formatted string e.g. sslClientAuthentication sslServerAuthentication</p> <hr/> <p>BasicConstraints (O) - (String) Contains the basic constraints set in the signer's certificate, e.g. Type=End Entity</p> <p>ValidFrom (O) - (String) Contains the Valid From date of the signer's certificate, e.g. Mon Jun 30 16:44:47 GMT+01:00 2005</p> <p>Valid To (O) - (String) Contains the Valid To date of the signer's certificate, e.g. Mon Jun 30 16:44:47 GMT+01:00 2007</p> <p>IssuerName (O) - (String) Contains the Issuer Distinguish Name of the signer's certificate, e.g. CN=Test L2 CA1,OU=SQA,O=Ascertia,C=GB</p> <p>Valid To (O) - (String) If set to true, returns the Valid To of the signer's certificate e.g. Mon Jun 30 16:44:47 GMT+01:00 2007</p> <p>IssuerName (O) - (String) If set to true, returns the Issuer Distinguish Name of the signer's certificate e.g. CN=Test L2 CA1,OU=SQA,O=Ascertia,C=GB</p> <p>CertificateSerialNumber (O) - (String) If set to true, returns the serial number of the signer's certificate e.g. 23495 (as a string integer value).</p> <p>CRLUrl (O) - (Boolean) If set to true then the URL from where the CRL was downloaded is returned.</p> <p>CRLNumber (O) - (Boolean) If set to true then the CRLNumber from the CRL used to check the revocation of the signer's certificate is returned.</p>
--	--

5.2 Signing Web Service

The Signing Web Service interface is based on the W3C OASIS DSS protocol (OASIS DSS, "http://docs.oasis-open.org/dss/oasis-dss-1.0-core-spec-cd-02.pdf"). To provide flexibility of providing other parameters such as **profileID**, **client identifiers** and **certificate aliases** to be sent using the DSS protocol, an extension schema has been developed in the DSS protocol. The extension schema utilizes the **OptionalInputs** element of DSS.

The details of XML interfaces within the DSS protocol are available in the OASIS published specification. The DSS XML Schema file **oasis-dss.xsd** and the corresponding protocol document **oasis-dss-1.0-core-spec-cd-02.pdf** are should be read. The ADSS Signing Service also uses an

extension XML schema along with the OASIS DSS XML Schema feature of DSS. The details of ADSS OptionalInputs elements are described below. The DSS extension schema file **adss_signmodule.xsd** is also available with this document. The high level structure of the schema is as follows:




5.2.1 Document Signing Request

All document signing request elements are under a top level **SignRequest (M) - (Container)** XML element. It has two attributes named RequestID and ProfileID and two child elements OptionalInputs and InputDocuments.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
OptionalInputs (O) - (Container)	Contains details of the entity sending the request and any ProfileAttributes to be used by ADSS during signing.
	OriginatorInfo (O) - (Container) - Contains details of the entity that is sending the request. OriginatorID (M) - (String) Contains a unique ID for the originator. The unique ID must be registered within ADSS Server and if client-authenticated SSL is used then the ID must match the Common Name within the client SSL Certificate. If the OriginatorID is not registered, the request will be rejected and transaction is logged. NOTE: For the signing service this must be set although it is optional within the protocol
	OriginatorKeyInfo (O) - (Container) - Provides details of the certificate alias used to sign the document in the request. The following can be specified for the signing certificate. NOTE: For the signing service this element must be set although it is optional in the protocol <ul style="list-style-type: none"> Alias (M) - (String) The Certificate Alias to be used for signing the document. This Alias must be registered in the ADSS Server CertificationService or inside ADSS Key Manager module Password (O) - (String) The password for the PKCS#12 generated within ADSS Server as a

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values

		<p>result of a Certificate Service CREATE / RENEW request. The password is only required by ADSS Server if the Crypto Source setting of the ADSS Server is set to 'software'. However if the keys of certificate are Server related and not specific to a user / client and generated manually in Key Manager then the password is not required, i.e. this could be a corporate signing certificate.</p>
	<p>ProfileAttributes (O) - (Container)</p>	<p>Used to customize the signing profile used to process DSS signing request. This element provides the flexibility to override different attributes of an ADSS signing profile. Note: ADSS Server will use the values provided within this element instead of the selected profile's default values only if the selected profile allows the calling application to override these attributes. See the ADSS Server Admin Manual for information on how to create signing profiles and also how to lock elements within these.</p> <p>ProfileAttribute (O) - (Container) - (Multiple) Contains Profile attribute(s) and values to be overridden by ADSS Server when processing the request. The following are the Profile Attributes that can be over-ridden</p> <ul style="list-style-type: none"> • SIGNING_REASON (O) - (String) Specifies the signing reason e.g. "I approve this document" • SIGNING_LOCATION (O) - (String) Specifies the location where the document is being signed e.g. "London" • SIGNING_FIELD (O) - (String) Specifies the name of an existing blank signature field to be signed • SIGNING_PAGE (O) - (String) Specifies the page on which the signature should be placed (applicable only if document allows visible signatures) e.g. 10 (Note that multiple pages can be specified only by using an XML preferences file when creating the profile) • SIGNING_AREA (O) - (String) Specifies the page area on which the signature should be placed (applicable only if visible signatures are being used). The following values specify the location: <ul style="list-style-type: none"> ○ 1 for TOP LEFT ○ 2 for TOP RIGHT ○ 3 for CENTER ○ 4 for BOTTOM LEFT ○ 5 for BOTTOM RIGHT • VISIBILITY (O) - (String) This flag indicates whether the signature should be visible or invisible. Use TRUE for a visible signature and FALSE for an invisible signature (applicable only if the document type supports visible signatures) • CONTACT_INFO (O) - (String) Specifies contact information of for the signer e.g. a telephone number, email address, street address. • HAND_SIGNATURE (O) - (Base64) An image to be placed as a hand signature (applicable only if the document type supports visible signatures)

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	<ul style="list-style-type: none"> COMPANY_LOGO (O) - (Base64) An image to be placed as company logo (applicable only if the document type supports visible signatures). The following figure shows a signature appearance with various elements set. Note the contact info is not visible on the document but can be seen when viewing the signature properties: <div data-bbox="837 481 1433 705" style="border: 1px solid black; padding: 5px; margin: 10px 0;">  </div>
	<p>DOCUMENT_SIGNATURE_RELATIONSHIP (O) - (String) This defines what type of signature is to be created by ADSS Server. The supported values are: ENVELOPED, ENVELOPING and DETACHED.</p> <p>Note: If the input document is a PDF then this element is ignored since PDFs only support one format.</p>
InputDocuments(M) - (Container)	<p>Contains the document to be signed by ADSS Server. The supported document types are PDF, XML or any other file based data. The current version only supports a single document per request although the protocol allows for multiple documents to be signed.</p> <p>Document (M) - (Container) - (Multiple) Permitted values are either:</p> <ul style="list-style-type: none"> XMLData (XML) An XML which is to be signed Base64Data (Container) <ul style="list-style-type: none"> Base64 content. MIME type - explains the MIME type of the Base64 e.g. PDF if the base64 content is a PDF document, ANY if the base64 content is a raw data
ProfileID (O) (anyURI)	<p>This attribute is used to identify the signing profile that ADSS Server should use to process this request. See the ADSS Server Admin Manual for more details on how to configure signing profiles on the ADSS server. The value of this attribute is of the form "adss:signing:profile:001". If a profile is not identified then the default signing profile configured within ADSS Server is used.</p>
RequestID (O) (String)	<p>This attribute helps to uniquely identify a certification request. Any arbitrary string can be used. The same value will be provided in the ADSS Server response.</p>

5.2.2 Document Signing Response

All response elements are under a top level **SignResponse (M) - (Container)** XML element.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
Results (M) (Container)	<p>Contains the ADSS Server response status and whether the request was processed successfully</p> <ul style="list-style-type: none"> ResultMajor (M) - (anyURI) Permitted values are: <ul style="list-style-type: none"> urn:oasis:names:tc:dss:1.0:resultmajor.RequesterError shows that request processing has failed due to a problem in the request urn:oasis:names:tc:dss:1.0:resultmajor.ResponderError shows that request processing has failed due to a problem at the server

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	<ul style="list-style-type: none"> ○ urn:oasis:names:tc:dss:1.0:resultmajor.Success shows that request has been processed successfully ● ResultMinor (O) - (anyURI) Specifies a code that identifies why the failure occurred. Permitted values are: <ul style="list-style-type: none"> ○ urn:oasis:names:tc:dss:1.0:resultminor.NotAuthorized if the requester is not authorized to send the requests ○ urn:oasis:names:tc:dss:1.0:resultminor.NotSupported if the request could not be processed because of unsupported information within in the requests ● ResultMessage (O) (String) If an error has occurred, it contains the error code and a brief description of the error. The following error codes are possible: <ul style="list-style-type: none"> ○ 1001: Originator doesn't exist {OriginatorID is unknown / not registered} ○ 1002: OriginatorID is missing in the request ○ 1003: An unsupported document/document type is specified in the request ○ 1004: An invalid Profile ID is specified in the request ○ 1005: The specified profile is not compatible with the document type ○ 1006: No Profile ID is specified in the request (and no default profile has been configured) ○ 1007: The request is not supported ○ 1008: The signer alias cannot be created from the certificate subject DN ○ 1009: The signer certificate is missing for this certificate alias ○ 1010: The issuer certificate is not available to allow a long term signature to be created ○ 1011: The signer certificate has expired ○ 2001: Request processing failed because of an internal error ○ 2002: It was not possible to obtain a timestamp token so long term signature creation failed ○ 2003: It was not possible to get the revocation information of the signer's certificate ○ 2004: Signature computation failed
OptionalOutputs (O) - (Container)	Contains the signed document as Base64 encoded data.
SignatureObject (O) - (Container)	For XML DSig signatures this element is omitted. The permitted values are: <ul style="list-style-type: none"> ● Base64Signature (Container) <ul style="list-style-type: none"> ○ Base64 (M) (base64Binary) ● Type (M) - (anyURI) The CMS is returned in case of PDF signing and PKCS7 is returned in case of PKCS#7 signature creation
RequestID (O) - (String)	- This attribute helps to uniquely identify a signing request. If included in the response, the value copied from the corresponding signing request data.
ProfileID (O) - (anyURI)	- This attribute identifies the signing profile used by ADSS Server. This can either be the ProfileID given in the request or if not provided it contains the default signing ProfileID configured within ADSS Server.



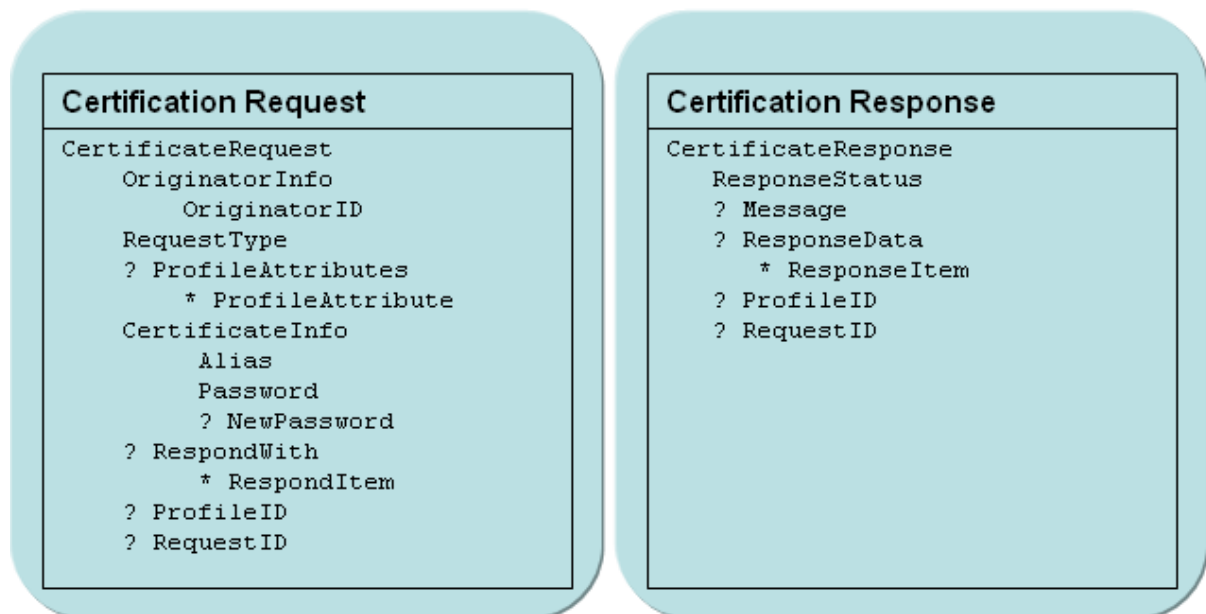
Note that for the SignResponse message the standard DSS protocol is used and all binary data e.g. Documents, Certificates, Hashes, PKCS#7 objects are Base 64 encoded.

5.3 Certification Service

ADSS Server offers an optional certification authority service. This service enables applications to register entities and create certificate(s) on their behalf. Entities can be servers, applications or end-users that wish to be able to sign data using server-side signing processing (not zero-footprint client-side signing). The registration process is:

- Generate RSA signing key pairs according to a pre-defined policy. For enhanced security ADSS Server supports Hardware Security Modules (HSMs)
- Generate a PKCS#10 certificate request for the public key and automatically post this to a configured Certificate Authority (CA) based on a specified certification policy OR from the built-in ADSS Certification Service

The ADSS Certification Service is based on a flexible XML schema. An overview of the XML elements used in the ADSS Certification Service request and response messages are provided below. For a detailed description refer to the XML Schema file **adss_certmodule.xsd** provided separately within the installation software.



5.3.1 Certification request

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
CertificateRequest (M) - (Container)	The top level element of the Certification Request message. It has two attributes named RequestID and ProfileID and five child elements named OriginatorInfo, RequestType, ProfileAttributes, CertificateInfo and RespondWith.
OriginatorInfo (M) - (Container)	Contains details of the entity that is sending the request. OriginatorID (M) - (String) Contains a unique ID for the originator. The unique ID must be registered within ADSS Server and if client-authenticated SSL is used then it must match the Common Name within the Client's SSL

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	<p>Certificate. If the OriginatorID is not registered, the request will be rejected.</p>
	<p>RequestType (M) - (Enumeration) Identifies the purpose of the request. The following set of values can be used for this element:</p> <ul style="list-style-type: none"> • CREATE: To create a new key pair and certificate. • RENEW to renew the certificate (the previous certificate is deleted). • DELETE to delete the certificate • CHANGE_PASSWORD to change the password of the PFX private key file held on the ADSS Server • RECOVER_KEY to recover the PKCS#12 object stored in ADSS server database
	<p>RequestID (O) - (String) This attribute helps to uniquely identify a certification request. Any arbitrary string can be used as the value of this attribute. This is expected in the response back from ADSS Server.</p>
	<p>ProfileID (O) - (anyURI) This attribute is used to identify a certification profile that ADSS Server must use to process this request. See the ADSS Server Admin Manual for further details on how to configure certification profiles on the ADSS Server. The value of this attribute is of the form adss:module:cert:001. If profile is not identified then the ADSS Server default certification ProfileID is used.</p>
	<p>ProfileAttributes (O) - (Container) Used to customize the profile that will be used to process this certification request. This element provides the flexibility to override the attributes of a certification profile. ADSS Server uses the values provided within this element instead of profile default values only if the profile allows these attributes to be over-ridden (see the ADSS Server Admin Guide for details of how to lock the default settings).</p> <p>ProfileAttribute (O) - (Container) - (Multiple) Contains Profile attribute(s) and values to be used as override values by ADSS Server when processing the request. The following are the Profile Attributes that can be over-ridden:</p> <ul style="list-style-type: none"> • SUBJECT_DN: Subject Distinguished Name of the certificate to be generated for the user e.g. CN=Alice, OU=HR Dept, O=ACME, C=GB. Only the following subject DN attributes: CN, OU, O, C, L, S, E can be over-ridden. • KEY_SIZE: Size of the key pair to be generated in bits (1024 and 2048). • KEY_TYPE: Type of the key pair to be generated (Currently only RSA key pairs are supported). • VALIDITY_PERIOD: Validity period of certificate in months e.g. 12. • CA_ALIAS: An alias of the CA that ADSS Server should use to certify the public key of the entity. Refer to the ADSS Server Admin Guide for details on CA aliases. Permitted values are: <ul style="list-style-type: none"> → INTERNAL - specifies that the ADSS Server internal CA service is to be used → Any alias assigned to an external CA

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	<p>CertificateInfo (M) - (Container) This element is used to provide the certificate specific data to ADSS. The list of certificate info elements which can be provided in certificate request is detailed below:</p> <p>ALIAS (M) - (String): The alias of the certificate for this certification request. ADSS Server will generate the certificate using this ALIAS if this alias is not used for this particular OriginatorID.</p> <p>PASSWORD (M) - (String) Password of the PFX or PKCS#12 private key file. If not provided in Certificate CREATE or RENEW request then server assigns a secure password itself which can later be retrieved by specifying PASSWORD in RespondWith element</p> <p>NEW_PASSWORD (O) - (String) New password for the private key file. This is required only if the RequestType is "CHANGE_PASSWORD"</p>
RespondWith (O) - (Container)	<p>Used to specify the items the calling application wants to receive within the certification web service response message.</p> <p>ResponseItem (O) - (Container) - (Multiple). This element is used to specify the items the calling application wants to receive within the certification web service response message. The list of items which can be requested are:</p> <p>CERTIFICATE (Base64) The X509 certificate</p> <p>PKCS_12 (Base64) The PKCS#12 private key file</p> <p>PKCS_7 (Base64): The PKCS#7 certificate chain</p> <p>EXPIRY_DATE (String) - The expiry date / time of the certificate</p> <p>The business application may wish to remember some of these elements. For example a web application may want to keep a local record of the expiry dates of user's certificates so that it can notify users that their certificates need to be renewed.</p> <p>PASSWORD (String): The server generated password of the PKCS#12 object</p>



An external CA must be able to process the certification request from the ADSS Server. For Windows 2003 CAs, a middleware module is provided that handles the dialogue between ADSS Server and Windows CA. Read the ADSS Server installation Manual for details on how to install and configure the Windows 2003 CA middleware module.

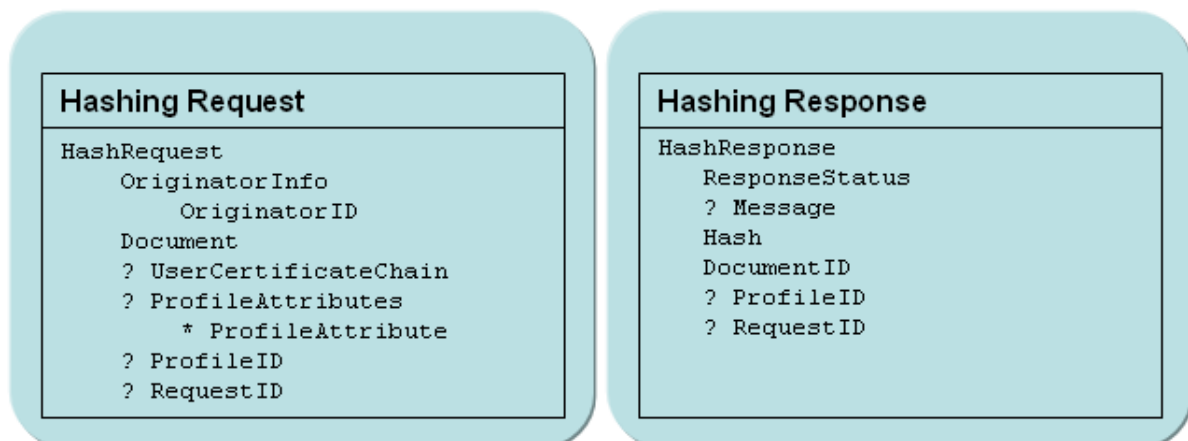
5.3.2 Certification Response

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
CertificateResponse (M) - (Container)	<p>This is the top level element of Certification Response. It has three attributes named ResponseStatus, RequestID and ProfileID and two child elements named Message and ResponseData. The detail of each element is provided below.</p> <p>ResponseStatus (M) - (Enumeration) Provides information regarding whether the request was processed successfully or failed. Possible values are: SUCCESS FAILED {FAILURE}</p> <p>RequestID (O) - (String) This attribute helps to uniquely identify a certification request. If included in the response, the value for this attribute is taken exactly as</p>

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	from the corresponding certification request message.
	ProfileID (O) - (anyURI) This attribute identifies the certification profile ADSS Server used to process this request. This can either be the ProfileID provided in the request or, if not provided, then the default ProfileID.
	Message (O) - (String) If the value of ResponseStatus attribute is FAILED {FAILURE} then this element contains the failure reason. The failure reason is a string description of the error encountered by ADSS Server while processing the request.
ResponseData (O) - (String)	<p>If the value of ResponseStatus attribute is SUCCESS then this contains the RespondWith items requested in corresponding request.</p> <p>ResponseItem (O) - (Container) - (Multiple) This element is used to specify the items the calling application requested and the corresponding values. The list of items that can be requested are:</p> <p>CERTIFICATE (Base64) The X509 certificate</p> <p>PKCS_12 (Base64) The PKCS#12 private key file</p> <p>PKCS_7 (Base64) The PKCS#7 certificate chain</p> <p>EXPIRY_DATE (DateTime) The expiry date of the certificate</p> <p>If a value is not available or ADSS Server cannot produce the value then it is not included in the response</p> <p>PASSWORD (String): The server generated password of the PKCS#12 object</p>


5.4 Hashing Service

An application can use Hashing Service to send a PDF document to ADSS Server and have the Hash value returned. The Hashing web service interface is based on a flexible XML schema. A summary of the XML elements used in the signing service requests and responses is shown in the following diagram. For a detailed description refer to the XML Schema file **adss_signmodule.xsd** provided separately within the ADSS Server installation software. The high level structure of the schema is as follows:



5.4.1 Hashing Request

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
HashRequest (M) - (Container)	This is the top level element of the Hashing Request message. It has four child elements named Document, ProfileAttributes, UserCertificateChain, RequestID, ProfileID and OriginatorInfo
	OriginatorInfo (M) - (Container) Contains details of the entity that is sending the request. OriginatorID (M) - (String) Contains a unique ID for originator. This ID must be registered within the ADSS Server and if client-authenticated SSL is used then it must match the Common Name within the client SSL Certificate. If the OriginatorID is not registered, the request will be rejected.
	Document (M) - (Base64) Contains a Base64 encoded PDF document.
	RequestID (O) - (String) Contains a unique identifier assigned to the HashingRequest. This value is returned in the ADSS Server response.
	ProfileID (O) - (anyURI) Contains the ProfileID to be used by ADSS Server for hash generation. The ProfileID must be registered within the ADSS Server. If it is not the default ProfileID will be used. Refer to the ADSS Server Admin Manual for more details on how to configure profiles.
	UserCertificateChain (O) - (Base64) Contains a Base64 encoded certificate chain for the signer. This certificate chain is embedded in the PDF document before hashing as mandated in the PDF signing specifications. The chain should contain at least the end-entity signing certificate so that the certificate information is set in signature properties. If more than one certificate is specified then the first certificate will be used and rest will be ignored. Note: For the generation of PDF signatures, this element is REQUIRED.
	ProfileAttributes (O) - (Container) The values inside the ProfileAttributes are embedded in the PDF document before the hashing. The list of profile attributes that can be altered from the default values (if allowed in the profile) are: <ul style="list-style-type: none"> • SIGNING_REASON (O) - (String) Specifies the signing reason e.g. "I approve this document" • SIGNING_LOCATION (O) - (String) Specifies the location where the document is being signed e.g. "London" • SIGNING_FIELD (O) - (String) Specifies the name of an existing blank signature field to be signed • SIGNING_PAGE (O) - (String) Specifies the page on which the signature should be placed (applicable only if document allows visible signatures) e.g. 10 • SIGNING_AREA (O) - (String) Specifies the page area on which the signature should be placed (applicable only if visible signatures are being used). The following values specify the location: <ul style="list-style-type: none"> ○ 1 for TOP LEFT ○ 2 for TOP RIGHT ○ 3 for CENTER ○ 4 for BOTTOM LEFT ○ 5 for BOTTOM RIGHT

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
		<ul style="list-style-type: none"> • VISIBILITY (O) - (String) This flag indicates whether the signature should be visible or invisible. Use TRUE for a visible signature and FALSE for an invisible signature (applicable only if the document type supports visible signatures) • CONTACT_INFO (O) - (String) Specifies contact information of for the signer e.g. a telephone number, email address, street address. • HAND_SIGNATURE (O) - (Base64) An image to be placed as a hand signature (applicable only if the document type supports visible signatures) • COMPANY_LOGO (O) - (Base64) An image to be placed as company logo (applicable only if the document type supports visible signatures). The following figure shows a signature appearance with various elements set. Note the contact info is not visible on the document but can be seen when viewing the signature properties: 
		DOCUMENT_SIGNATURE_RELATIONSHIP (O) - (String) This is not applicable

5.4.2 Hashing Response Element

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
HashResponse (M) - (Container)	This is the top level element of the Hashing Response message. It has four child elements named DocumentID, Message, Hash and three attributes named ResponseStatus, RequestID, ProfileID
	Message (O) - (String) Contains the description of the any error that occurred whilst processing the Hashing Request
	Hash (O) - (Base64) Contains the resultant hash of the document.
	DocumentID (O) - (String) Contains a unique identifier assigned to the document received by ADSS Server. This DocumentID must be provided within a Document Assembly request if a PKCS#7 signature object is to be sent back to ADSS Server, e.g. for embedding a signature created by the GoSign Applet. Note: DocumentID is always returned by ADSS Server and the calling application must use this in any subsequent assembly request.
	ResponseStatus (M) - (ResponseStatusEnum) Provides information on whether whether the request was processed successfully or it failed. Possible values are: <ul style="list-style-type: none"> • SUCCESS • FAILED {FAILURE}
	RequestID (O) - (String) Contains the same unique identifier sent earlier in the HashingRequest > RequestID received by ADSS Server.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values

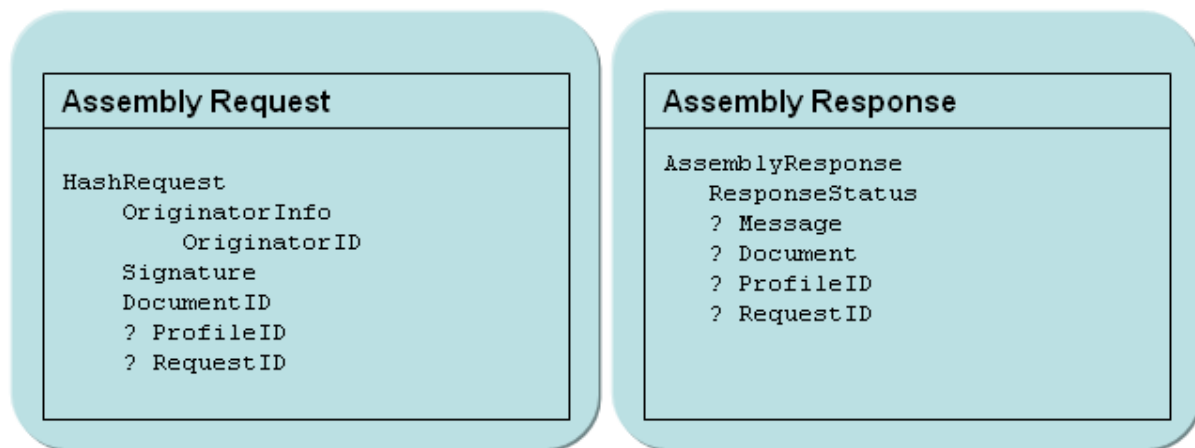
	ProfileID (O) - (anyURI) Contains the ProfileID used by ADSS Server for hash generation. This can either be the ProfileID provided in the request or it is set to the default ProfileID.
--	---

5.5 Assembly Service

The Assembly Service works in conjunction with the Hashing service. The purpose of the Assembly Service is to provide PKCS#7 signatures generated by the GoSign Applet so that final assembly of a document can be completed.

Currently the Assembly Service only supports PKCS#7 signatures and PDF documents. To use the Assembly Service the application should first call the Hashing Service to get the Hash value and DocumentID and then get this Hash signed externally using the GoSign Applet or another signing facility. The generated PKCS#7 and the DocumentID has to be sent to ADSS Server for the PKCS#7 signature to be embedded within the document.

The Assembly Service interface is uses a flexible and XML Schema. An overview of the XML elements used in the Assembly Service request and response messages is shown in the following diagram. For a detailed description refer to the XML Schema file **adss_signmodule.xsd** provided within the ADSS Server software installation:



5.5.1 Assembly Request

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values

AssemblyRequest (M) - (Container)	This is the top level element of the Assembly request message. It has three child elements named DocumentID, Signature and OriginatorInfo and two attributes RequestID, ProfileID. Using this you can send the PKCS7 made on the hash earlier received from ADSS, to ADSS.
OriginatorInfo (O) - (Container)	Contains details of the entity that is sending the request. OriginatorID (M) - (String) Contains a unique ID for the originator. The unique ID must be registered within the ADSS Server and if client-authenticated SSL used then it must match the Common Name within the client SSL Certificate. If the OriginatorID is not registered, the request will be rejected.
Signature (M) - (Base64)	Contains a Base64 encoded PKCS#7 signature that is to be embedded inside the document.
DocumentID (M) - (String)	Contains the unique identifier assigned earlier within the Hashing response. The same DocumentID must be sent to ADSS Server for

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
	successful document assembly.
	RequestID (O) - (String) Contains a unique identifier assigned to the AssemblyRequest by the application. This will be returned in the ADSS response.
	ProfileID (O) - (anyURI) This is not applicable and reserved for future use.

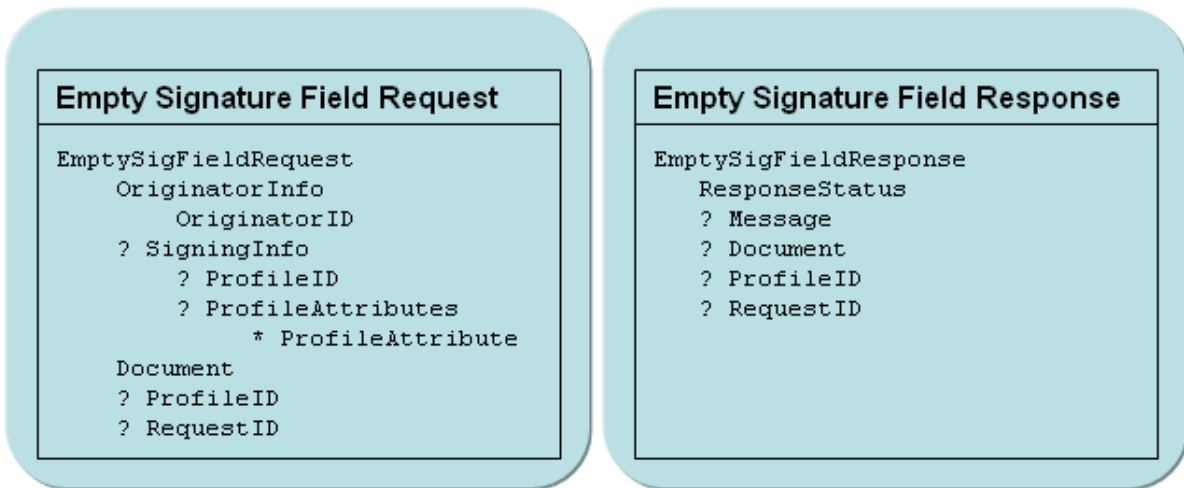
5.5.2 Assembly Response

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
AssemblyResponse (M) - (Container)	This is the top level element of the Assembly response message. It has three child elements named Message, Document, ResponseStatus and two attributes RequestID, ProfileID.
	Message (O) - (String) Contains the description of any error that occurs within ADSS Server during Assembly request processing.
	Document (O) - (Base64) Contains the final signed document formed by embedding the signature within the specified document.
	ResponseStatus (M) - (ResponseStatusEnum) Provides success or failure status information for the request. Possible values are: <ul style="list-style-type: none"> • SUCCESS • FAILED {FAILURE}
	RequestID (O) - (String) Contains the same unique identifier sent earlier in the AssemblyRequest > RequestID received by ADSS.
	ProfileID (O) - (anyURI) Contains the ProfileID used by ADSS Server for assembly. This is either the ProfileID provided in the request or the default ProfileID.

5.6 Empty Signature Field Creation Service

The purpose of the Empty Signature Field Creation service is to generate empty signature field(s) in the target PDF document. As an optionally it is possible to use this service to sign or certify an empty signature field within in the document.

The Empty Signature Field web service interface has a flexible XML schema. A summary of the XML elements used in the Empty Signature Field Creation service request and response messages is shown in the following diagram. For a detailed description refer to the XML Schema file **adss_signmodule.xsd** provided within the ADSS Server software installation:



5.6.1 Empty signature field generation request

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
<p>EmptySigFieldRequest (M) - (Container)</p>	<p>This is the top level element of the Empty Signature Field creation request message. It has three child elements named Document, SigningInfo, OriginatorInfo and two attributes RequestID, ProfileID.</p> <p>OriginatorInfo (O) - (Container) Contains details of the entity that is sending the request. OriginatorID (M) - (String) Contains a unique ID for originator. This must be registered within the ADSS Server and if client-authenticated SSL is used then it must match the Common Name within the client SSL Certificate. If the OriginatorID is not registered, the request will be rejected.</p> <p>SigningInfo (O) - (Container) Contains details of profile attributes and signing certificate details used to sign the PDF. Signing is carried out once the empty fields are created within the PDF document. ProfileID (O) - (anyURI) Contains the ProfileID used by ADSS to sign the empty field(s) in the PDF. The ProfileID must be registered within ADSS Server if specified. ProfileAttribute (O) - (Container) - (Multiple) Contains Profile attribute(s) and values to be overridden during processing by ADSS Server. The list of profile attributes that can be altered from the default values (if allowed) in the ADSS profile is:</p> <ul style="list-style-type: none"> • SIGNING_REASON (O) - (String) Specifies signing reason, e.g. "I approve this document" • SIGNING_LOCATION (O) - (String) Specifies the location data, e.g. "London" • SIGNING_FIELD (O) - (String) Specifies the name of the blank signature field to be signed, e.g. "Sign-1" • SIGNING_AREA (O) - (String) Specifies the page area on which the signature should be placed (applicable only if visible signatures are being used). Use the following values to specify the document location: <ul style="list-style-type: none"> ○ 1 for TOP LEFT ○ 2 for TOP RIGHT ○ 3 for CENTER ○ 4 for BOTTOM LEFT

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values

		<ul style="list-style-type: none"> ○ 5 for BOTTOM RIGHT ● SIGNING_PAGE (O) - (String) Specifies the page on which the signature should be placed (applicable only if document allows visible signatures) e.g. 10 ● VISIBILITY (O) - (String) This flag indicates whether the signature should be visible or invisible. Use TRUE for a visible signature and FALSE for an invisible signature ● CONTACT_INFO (O) - (String) Specifies contact information of document signer e.g. phone number, email address, postal address, etc ● HAND_SIGNATURE (O) - (Base64) Image to be used as a hand signature ● COMPANY_LOGO (O) - (Base64) Image to be placed as company logo <p>The following figure shows a signature appearance with the above elements set. Note the contact info is only shown when reviewing the signature properties:</p> <div data-bbox="783 864 1418 1106" data-label="Image"> </div> <ul style="list-style-type: none"> ● DOCUMENT_SIGNATURE_RELATIONSHIP (O) - (String) This is not applicable ● OriginatorKeyInfo (O) - (Container) Provides details of the certificate alias to be used to sign the target document. The following can be specified for the signing certificate. <ul style="list-style-type: none"> ● Alias (O) - (String) The Certificate Alias to be used for signing the document (must already be registered within ADSS Server) ● Password (O) - (String) The software store password if soft keys are being used for an end-entity. Note that if the keys are server generated keys and are not end-entity keys, i.e. they are generated manually in Key Manager then the password is not required. <p>The Alias element must be set to sign a signature field although it is marked as optional in the protocol.</p>
		<p>Document (M) - (Base64) Contains the Base64 encoded document in which empty signature fields are to be created.</p> <p>RequestID (O) - (String) Contains a unique identifier assigned by the requesting application. This will be returned in the ADSS Server response.</p>
		<p>ProfileID (O) - (anyURI) Contains the signing ProfileID to be used by ADSS Server for empty signature field generation. The ProfileID must be registered within ADSS Server, if it is not present then the default ProfileID will be used. See the ADSS Server Admin Manual for details on how to configure profiles. Note that empty fields are created first before the SigningInfo > ProfileID is used for signing the PDF (if used). Also the empty fields are only generated if XML based preferences are used in the profile.</p>

5.6.2 Empty Signature Field Response

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
EmptySigFieldResponse (M) - (Container)	This is the top level element. It has three child elements named Message, Document, ResponseStatus and two attributes RequestID, ProfileID. This is used to receive a PDF with empty (optionally signed) fields from ADSS Server.
	Message (O) - (String) Contains the description of any error that occurred whilst on the ADSS Server whilst processing the request.
	Document (O) - (Base64) Contains a PDF document formed after generating the empty field signature and optional signing/certifying them.
	ResponseStatus (M) - (ResponseStatusEnum) Provides success or failure status information for the request. Possible values are: <ul style="list-style-type: none"> • SUCCESS • FAILED {FAILURE}
	RequestID (O) - (String) Contains the unique RequestID identifier sent in the EmptySigFieldRequest message to ADSS Server.
	ProfileID (O) - (anyURI) Contains the ProfileID used by ADSS Server for empty signature field creation. This can either be the ProfileID in the request or if not provided the default ProfileID is used.

5.7 XKMS Service

ADSS server implements the XKMS Validate interface to provide an industry standard way to get the certificates validated by a trusted server. Please see below the link referring to the XKMS specification:

<http://www.w3.org/TR/xkms2/>

To know more about ADSS XKMS service please contact Ascertia Support.

5.8 LTAN Service

ADSS server implements the LTAP archive, export and deletes interfaces to provide an industry way to store the documents for a longer period. Please see below the link referring to the draft copy of LTAP specification:

<http://tools.ietf.org/html/draft-ietf-ltans-ltap-07>

To know more about ADSS LTAN service please contact Ascertia Support.

5.9 Decryption Service

ADSS server implements the OASIS DSS Encryption/Decryption profile; encrypted documents can be decrypted using the keys held by ADSS Server. Please see below the link referring to the draft copy of DSS Encryption/Decryption specification:

http://www.oasis-open.org/committees/download.php/25384/oasis-dss_profile-encryption_A-SIT_v0.1.doc

To know more about ADSS LTAN service please contact Ascertia Support.

5.10 Schema elements not currently supported in ADSS Server

The following is the list of the schema elements that are not currently supported in ADSS Server but which will be supported in future versions:

Request Header (For Signature/Certificate validation requests)

GatewayInfo (O) - (Container)	GatewayRespondAddress(O) - (String) Contains an address e.g. email address, or web server address to support asynchronous processing of requests by ADSS Server, i.e. the response is sent some time later.
MessageExtension (O)	For future possible usage to extend the protocol.
Value (O) - (String)	Contains the required transaction value set by a business application to ensure that the usage set by CA does not have a limitation lower than this value.
Request Body (For Signature verification requests)	
ContentInfo (M) - (Container)	<p>The content may also be provided using the following ways</p> <ul style="list-style-type: none"> • URI • XML <p>If the content to be signed is accessible via a URI (pointing to HTTP(s), File on hard disk, FTP, Gopher etc) then the target data is downloaded for signature verification. See http://www.ietf.org/rfc/rfc2396.txt for more details on URI. XML based content could be used where the content is of XML format.</p>
SignatureInfo (M) – (Container) (Multiple)	SignerInfo (O) - (URI or IssuerAndSerialNumber) A reference to the signer certificate, either (a) a URI or (b) An Issuer and Serial Number.
	<p>RespondWith</p> <p>SignHash (O) - (Boolean) If set to true then the Hash on which the signature was formed is returned.</p> <p>Timestamp (O) – (Boolean) If set to true then the RFC 3161 based timestamp present inside the signature is returned.</p>
Request Body (For Certificate Validation requests)	
CertificateInfo (M) - (Container)	CertificateType (M) (String) Contains the type of certificate which was sent i.e. PGP
	CertificateToValidate (M) - (Container) (URI or IssuerAndSerialNumber) A reference to the certificate to be validated, either (a) a URI or (b) An Issuer and Serial Number.
	<p>RespondWith</p> <p>SignHash (O) - (Boolean) If set to true then the Hash on which the signature was formed will be returned.</p> <p>Timestamp (O) – (Boolean) If set to true then the RFC 3161 based timestamp present inside the signature will be returned.</p>
Response Body (For Signature/Certificate Validation requests)	
VerifyInfo (M) - (Container) (Multiple)	<p>RespondWith</p> <p>SignatureQualityLevel (O) - (String) If the assertion status is Invalid or Indeterminate then the actual calculated value for the signature quality level is always returned regardless of the request by the application or Relying Party.</p> <p>CertificateQualityLevel (O) - (String) If the assertion status is Invalid or Indeterminate then the actual calculated value for the certificate quality level is always returned regardless of the request by the application or Relying Party.</p> <p>SignHash (O) - (String) If set to true then the Hash value on which the signature was calculated is returned.</p> <p>Timestamp (O) - (String) If set to true then the RFC 3161 based timestamp present inside the signature is returned.</p>
Document Signing Request	

OptionalInputs (M) - (Container)	Contains details of the entity sending the request and some ProfileAttributes to be used by ADSS Server during signing.	
	OriginatorInfo (M) - (Container)	RespondAddress (O) - (String) Contains an address e.g. email address, web server address to support asynchronous processing.
	OriginatorKeyInfo (M) - (Container)	Provides details of the certificate alias used to sign the target document. The following can be specified for the signing certificate: PKCS#12 (O) - (String) The PKCS#12 to be used for signing the document
Certification request		
CertificateRequest (M) - (Container)	OriginatorInfo (M) - (Container)	RespondAddress (O) - (String) Contains an address e.g. email address, web server address to support asynchronous processing.
	CertificateInfo (M) - (Container) This element is used to provide the certificate specific data to ADSS Server. The list of certificate info elements that can be provided in certificate request are: PKCS_12: Base64 encoded PKCS#12 certificate file that contains the end-entity private key. CERTIFICATE: Base64 encoded X509 certificate file EXPIRY_DATE: Certificate expiry date PKCS_7: Base64 encoded PKCS#7 certificate file	
Hashing Request		
HashRequest (M) - (Container)	This is the top level element of the Hashing Request message. It has four child elements named Document, ProfileAttributes, UserCertificateChain, RequestID, ProfileID and OriginatorInfo	
	OriginatorInfo (M) - (Container)	RespondAddress (O) - (String) Contains an address e.g. email address, web server address to support asynchronous processing of requests i.e. the response is sent after some time without requiring the client to remain connected.
Assembly Request		
AssemblyRequest (M) - (Container)	This is the top level element of the Assembly request message. It has three child elements named DocumentID, Signature and OriginatorInfo and two attributes RequestID, ProfileID.	
	OriginatorInfo (M) - (Container)	RespondAddress (O) - (String) Contains an address e.g. email address, web server address to support asynchronous processing.
Empty Signature Field Request		
EmptySigFieldRequest (M) - (Container)	This is the top level element of the Empty Signature Field creation request message. It has three child elements named Document, SigningInfo, OriginatorInfo and two attributes RequestID, ProfileID.	
	OriginatorInfo (M) - (Container)	RespondAddress (O) - (String) Contains an address e.g. email address, web server address to support asynchronous.
	SigningInfo (O) - (Container)	PKCS#12 (O) - (String) The PKCS#12 file to be used for signing the document. This currently not supported

5.11 Supported Algorithms in ADSS Server

The following is a list of signing/ hashing algorithms and key lengths that ADSS Server supports in different services:

ADSS Service	Signing Algorithms	Hashing Algorithms	Signing Key Lengths
Signing	sha1WithRSAEncryption sha224WithRSAEncryption sha256WithRSAEncryption sha384WithRSAEncryption sha512WithRSAEncryption RipeMD128WithRSAEncryption RipeMD160WithRSAEncryption	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 RipeMD128 RipeMD160	RSA: 1024, 2048, 4096 ECC: (in a future version)
Verification	sha1WithRSAEncryption sha192WithRSAEncryption sha224WithRSAEncryption sha256WithRSAEncryption sha384WithRSAEncryption sha512WithRSAEncryption MD5WithRSAEncryption	SHA-1 SHA-192 SHA-224 SHA-256 SHA-384 SHA-512 MD5	RSA: 512, 1024, 2048, 3072, 4096 ECC: (in a future version)
Hashing, Assembly	-	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 RipeMD128 RipeMD160	-
Certification using LOCAL CA	sha1WithRSAEncryption sha256WithRSAEncryption	SHA-1 SHA-256	RSA: 1024, 2048 ECC: (in a future version)
OCSP	sha1WithRSAEncryption sha256WithRSAEncryption sha384WithRSAEncryption sha512WithRSAEncryption	SHA-1 SHA-256 SHA-384 SHA-512	RSA: 1024, 2048, 4096 ECC: (in a future version)
TSA	sha1WithRSAEncryption sha256WithRSAEncryption sha384WithRSAEncryption sha512WithRSAEncryption	SHA-1 SHA-256 SHA-384 SHA-512	RSA: 1024, 2048, 4096 ECC: (in a future version)

Grey text = in the next release

**** End of Document ****