ADSS Server v8.3.14 –

ADSS Server Developers Guide

ASCERTIA LTD

DECEMBER 2025

Document Version- 1.0.0

© Ascertia Limited. All rights reserved.
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 5

1.1 SCOPE 5
1.2 INTENDED READERSHIP 5
1.3 CONVENTIONS 5
1.4 TECHNICAL SUPPORT 5
1.5 GLOSSARY 6
1.6 REFERENCES TO PKI STANDARDS 7

2 ADSS SERVER OVERVIEW 9

2.1 MODES OF OPERATION 10
2.2 ADSS SERVER ARCHITECTURE 10
2.3 ADSS SERVER WEB SERVICES INTERFACES 11
2.4 INTERFACING TO THE OCSP, SCVP AND TSA SERVICES 13
2.5 INTEGRATION WITH BUSINESS APPLICATIONS 13
2.6 ADSS SERVER ADMIN GUIDE 14
2.7 SERVICE URLS 14
2.8 ADSS SERVER INTERFACE SCHEMA 18
2.9 ADSS CLIENT SDK (JAVA AND .NET VERSIONS) 20

3 MESSAGE, REQUEST AND RESPONSE CLASSES 23

3.1 MESSAGE CLASS 23
3.2 REQUEST CLASS 23
3.3 RESPONSE CLASS 26

4 ADSS SIGNING SERVICE 27

4.1 DIGITAL SIGNATURE STANDARDS 27
4.2 SETTING UP SIGNING SERVICE PROFILES 27
4.3 THE SIGNING SERVICE API 28
4.4 SIGNING REQUEST AND RESPONSE CLASSES 28
4.5 PDF SIGNING REQUEST AND RESPONSE CLASSES 34
4.6 EMPTY SIGNATURE FIELD REQUEST AND RESPONSE CLASSES 36
4.7 DOCUMENT HASHING REQUEST AND RESPONSE CLASSES 38
4.8 SIGNATURE ASSEMBLY REQUEST AND RESPONSE CLASSES 39
4.9 OFFICE SIGNING REQUEST AND RESPONSE CLASSES 40
4.10 SIGNING STATUS REQUEST AND RESPONSE CLASSES 42
4.11 CERTIFICATE DOWNLOAD REQUEST AND RESPONSE CLASSES 43
4.12 SIGNING SERVICE SAMPLE CODE 44
4.13 ADSS SIGNING SERVICE SUPPORTED ALGORITHMS 47
4.14 ERROR CODES 48

5 ADSS VERIFICATION SERVICE 52

5.1 DIGITAL SIGNATURE STANDARDS 52
5.2 SETTING UP VERIFICATION SERVICE PROFILES 52
5.3 THE VERIFICATION SERVICE API 53
5.4 VERIFICATION REQUEST CLASSES 53
5.5 SIGNATURE VERIFICATION REQUESTS 56
5.6 CERTIFICATE VALIDATION REQUESTS 58
5.7 SENDING THE VERIFICATION REQUEST 60
5.8 VERIFICATION RESPONSE METHODS 61
5.9 VERIFICATION SERVICE REPORTS 65
5.10 VERIFICATION SERVICE SAMPLE CODE 69
5.11 ADSS VERIFICATION SERVICE SUPPORTED ALGORITHMS 71
5.12 ERROR CODES 72

6 ADSS CERTIFICATION SERVICE 74

6.1 CERTIFICATION USE CASES AND ASCERTIA PROTOCOL SCHEMA 74
6.2 CERTIFICATION PROFILES 74
6.3 THE CERTIFICATION SERVICE API 75
6.4 CERTIFICATION REQUEST CLASS 75

6.5 Certification Response Class 87
6.6 Certification Service Sample Code 88
6.7 ADSS Certification Service Supported Algorithms 89
6.8 Error Codes 90
7 ADSS OCSP SERVICE 93
7.1 Setting up ADSS OCSP Service Profiles 93
7.2 The ADSS OCSP Service API 93
7.3 OCSP Request Class 93
7.4 OCSP Response Class 95
7.5 OCSP Service Sample Code 96
7.6 ADSS OCSP Service Supported Algorithms 97
7.7 Error Codes 97
8 ADSS TSA SERVICE 99
8.1 Setting up ADSS TSA Profiles 99
8.2 The ADSS TSA Service API 99
8.3 TSP Request Class 99
8.4 Timestamp Response Class 101
8.5 Timestamp Verify Request Class 103
8.6 Timestamp Verify Response Class 103
8.7 TSA Service Sample Code 104
8.8 ADSS TSA Service Supported Algorithms 105
8.9 Error Codes 105
9 ADSS XKMS SERVICE 107
9.1 Support for the PEPPOL standard 107
9.2 Setting up XKMS Validation Profiles 107
9.3 The XKMS Validation Service API 107
9.4 Validate Request Class 108
9.5 Validate Result Class 109
9.6 Compound Request Class 112
9.7 Compound Result Class 112
9.8 XKMS Service Sample Code 113
9.9 ADSS XKMS Service Supported Algorithms 115
9.10 Error Codes 115
10 ADSS SCVP SERVICE 117
10.1 Simplified Use of SCVP 117
10.2 The SCVP Client API 119
10.3 SCVP Request Class 119
10.4 SCVP Response Class 122
10.5 SCVP Validation Policy Request Class 126
10.6 SCVP Validation Policy Response Class 126
10.7 SCVP Get Policy Info Request Class 128
10.8 SCVP Get Policy Info Response Class 128
10.9 SCVP Sample Code 129
10.10 ADSS SCVP Service Supported Algorithms 129
10.11 Error Codes 130
11 ADSS LTANS SERVICE 132
11.1 LTANS Service 132
11.2 LTANS Service Profiles 132
11.3 The LTANS Service API 133
11.4 Archiving Request Class 133
11.5 Archiving Response Class 134
11.6 LTANS Service Sample Code 136
11.7 ADSS LTANS Service Supported Algorithms 137
11.8 Error Codes 137
12 ADSS DECRYPTION SERVICE 140
12.1 ADSS Decryption Service Profiles 140

12.2 The ADSS Decryption Service API 140
12.3 Decryption Request Class 140
12.4 Decryption Response Class 142
12.5 Error Codes 142
13 ADSS GO>SIGN SERVICE 144
13.1 ADSS Go>Sign Service Overview 144
13.2 Error Codes 144
14 ADSS RA SERVICE 147
14.1 RA Use Cases and Ascertia Protocol Schema 147
14.2 RA Profiles 147
14.3 The RA Service API 147
14.4 RA Service Sample Code 156
14.5 Error Codes 157
15 ADSS RAS SERVICE 159
15.1 RAS Profiles 159
16 ADSS SAM SERVICE 160
16.1 SAM Profiles 160
17 ADSS CSP SERVICE 161
17.1 CSP Profiles 161
18 UTILITY CLASSES 162
18.1 AuthorisationData Class 162
18.2 Util Class 164
19 ADSS SIGNING SERVICE - USE CASES AND SCHEMA 167
19.1 Server-side Document Signing 167
19.2 Client-Side Document Signing 167
19.3 Creating an Empty Signature Field in PDF Documents 168
19.4 Document Hashing and Assembly 171
19.5 Document Hashing 172
19.6 Signature Assembly 174
20 ADSS CERTIFICATION SERVICE – USE CASE OVERVIEW 177
20.1 Generating / Registering a Key Pair and Certificate 177
20.2 Renewing a Key pair and Certificate 177
20.3 Retrieving Private Key (PKCS#12 object) and Certificate 178
20.4 Deleting a Key pair and Certificate 178
20.5 Changing an end-user key Authorisation Code 178
20.6 Operation of the Certification Service 179
21 ADSS RA SERVICE – USE CASE OVERVIEW 184
21.1 Generating a Key Pair and Certificate 184
21.2 Status of a Certificate 184
21.3 Revoking a Certificate 185
21.4 Operations of the RA Service 185

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:

Website	https://www.ascertia.com
Email	support@ascertia.com
Knowledge Base	Ascertia Community Portal

In addition to the free support service described 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	ADSS Server is Ascertia's strategic product for a wide range of Infrastructure and Enterprise trust services including digital signature creation ,verification, timestamping, certificate validation, certificate issuance and long-term archiving
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
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
OCSP	Online Certificate Status Protocol (an IETF protocol for verifying the revocation status of a digital certificate)
PKCS	Public Key Cryptographic Standards
PKI	Public Key Infrastructure
RFC	Request For Comments (an IETF Internet Standards Track Protocol)
RSA	Rivest, Shamir, Adleman (public key algorithm)
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 / TLS	Secure Sockets Layer / Transport Layer Security (a later version of SSL)
TA	Trust Authority (authority trusted for issuing certificates, CRLs, OCSP responses and/or time stamps)
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
PAdES	PDF Advanced Electronic Signatures
XKMS	XML Key Management Specifications
XML DigSig	XML Digital Signature standard

1.6 References to PKI Standards

CMS	http://tools.ietf.org/html/rfc3852
CAdES	http://pda.etsi.org/exchangefolder/ts_101733v010801p.pdf
PDF Signatures	PDF Public Key Digital Signature and Encryption Specification v3.2 http://www.adobe.com/devnet/pdf/pdf_reference.html
PAdES	http://pda.etsi.org/exchangefolder/ts_10277801v010101p.pdf
	http://pda.etsi.org/exchangefolder/ts_10277802v010201p.pdf
	http://pda.etsi.org/exchangefolder/ts_10277803v010101p.pdf
	http://pda.etsi.org/exchangefolder/ts_10277804v010101p.pdf
	http://pda.etsi.org/exchangefolder/ts_10277805v010101p.pdf
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
XAdES	http://uri.etsi.org/01903/v1.3.2/ts_101903v010302p.pdf
OCSP	http://www.ietf.org/rfc/rfc6960.txt
SCVP	http://www.ietf.org/rfc/rfc5055.txt
XKMS	http://www.w3.org/TR/xkms2/
LTANS	http://tools.ietf.org/html/draft-ietf-ltans-ltap-07
OASIS DSS Core	http://docs.oasis-open.org/dss/v1.0/oasis-dss-core-spec-v1.0-os.html
OASIS DSS AdES Profile	http://docs.oasis-open.org/dss/v1.0/oasis-dss-profiles-AdES-spec-v1.0-os.html
OASIS DSS-X Visible Signature Profile	http://docs.oasis-open.org/dss-x/profiles/visualsig/v1.0/cs01/oasis-dssx-1.0-profiles-visualsig-cs1.html
OASIS DSS-X Multi-Signature Verification Reports	http://docs.oasis-open.org/dss-x/profiles/verificationreport/oasis-dssx-1.0-profiles-vr-cs01.html
OASIS DSS Decryption Profile	http://www.oasis-open.org/committees/download.php/25384/oasis-dss_profile-encryption_A-SIT_v0.1.doc
PEPPOL XKMS	http://www.peppol.eu/work_in_progress/wp-1-esignature/results/deliverable-1.1/d1-1-part-1-background-and-scope
	http://www.peppol.eu/work_in_progress/wp-1-esignature/results/deliverable-1.1/d1-1-part-2-etendering-pilots-specification
	http://www.peppol.eu/work_in_progress/wp-1-esignature/results/deliverable-1.1/d1-1-part-3-signature-policies
	http://www.peppol.eu/work_in_progress/wp-1-esignature/results/deliverable-1.1/d1-1-part-4-architecture-and-trust-models
	http://www.peppol.eu/work_in_progress/wp-1-esignature/results/deliverable-1.1/d1-1-part-5-xkms-interface-specification
	http://www.peppol.eu/work_in_progress/wp-1-esignature/results/deliverable-1.1/d1-1-part-6-oasis-dss-interface-specification
	http://www.peppol.eu/work_in_progress/wp-1-esignature/results/deliverable-1.1/d1-1-part-7-eid-and-esignature-quality-classification

CMC	https://www.ietf.org/rfc/rfc2797.txt
EST	https://datatracker.ietf.org/doc/rfc7030/

2 ADSS Server Overview

ADSS Server provides the following trust services:

- An optional Signing Service that supports these features:
  - 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 Go>Sign Desktop and it needs to be embedded
  - Creating blank signature fields in PDF documents – useful when using Certify Signatures
  - Working with ADSS Go>Sign Desktop to offer local sign functionality
- An optional Verification Service which supports these features:
  - 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 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 and RFC 5816 specifications.
- An optional CRL Monitor service that supports these features:
  - Download CRLs of registered CAs
  - High availability of CRLs in fail over mode
- An optional XKMS Validate service which supports these features:
  - Validating a full certificate chain.
- 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 6960 specifications.
- An optional Server-based Certificate Validation Protocol (SCVP) service that supports these features:
  - Determining the path between X.509 digital certificate and a trusted root and the validation of that path according to a particular validation policy.
- 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 previously generated certificates
- 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

ADSS Server is a JEE 6 compliant application. It is a secure and scalable security services product that delivers trust for e-business applications and meets the CWA 14167-1 security requirements for trustworthy systems. 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 Guide**.

Typically only those trust services required by the business are licensed and deployed. Evaluation versions have a short-term license with most options enabled.

Integration with ADSS Server is easy when the ADSS Client SDK is used. This provides high level .Net and Java APIs that make it easy to call otherwise complex web-services.

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 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.

Ascertia also provides a small number of utility business applications that call ADSS Server. These applications provide immediate, out-of-the-box processing features to simplify certain task such as workflow approval, bulk file processing and bulk email processing. The list of front-end applications for ADSS Server includes:

- **SigningHub (SH)** is a document collaboration and approval / sign-off application that is ideal for replacing paper based signing with an on-line web-based document tracking and digital signature service. Users can be registered, documents can be uploaded, prepared for signature and then shared with other users for review and sign-off. Each collaborating user is automatically notified of new documents that require their review and approval. Documents are viewed and digitally signed using ADSS Go>Sign Desktop under the control of the SigningHub web-application.
- **Auto File Processor (AFP)** 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, verified or archived). AFP retrieves these documents and passes them to ADSS Server for processing. The returned documents are then placed into a designated output (or error) folder. This is an ideal solution for bulk signing, verification or archiving of documents in an unattended automated environment.
- **Secure Email Gateway (SES)** is a Mail Transfer Agent (MTA) server application that monitors emails as they are routed through the MTA. For outgoing emails it can filter emails that require digital signature or archiving and perform this automatically by making calls to ADSS Server. Both the signing of emails and/or signing of the attachment are supported. For incoming emails, SES can identify emails which are signed (or contained signed attachments) and pass these to ADSS Server for signature verification.

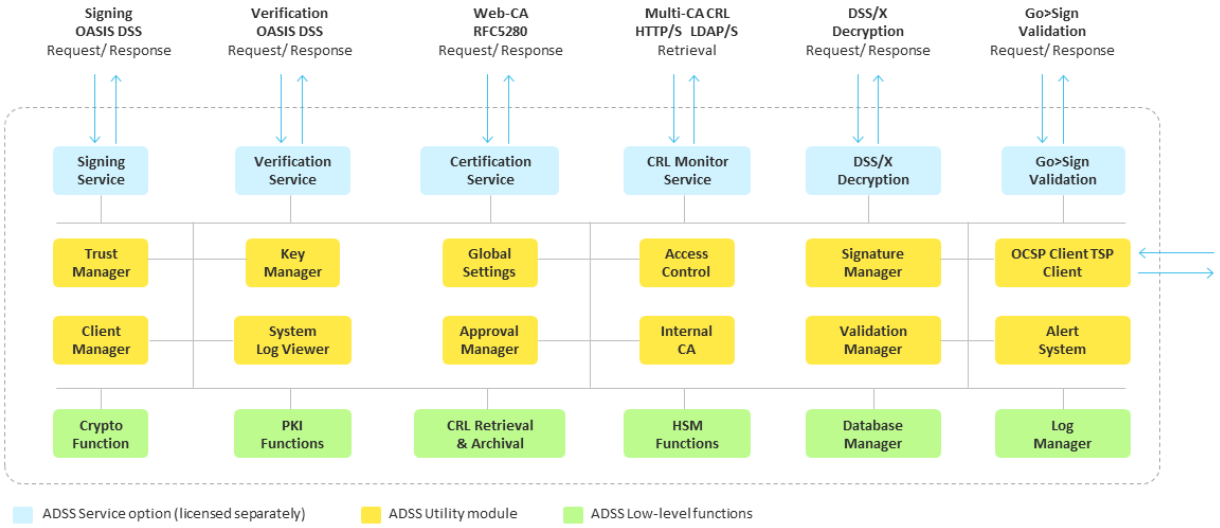
The above applications are described separately in their own manuals and not covered further in this document.

2.2 ADSS Server Architecture

ADSS Server is a multi-function application that provides a range of trust services. Only the relevant trust services are licensed and deployed to meet a business requirement. This approach enables cost-effective deployments spanning small, large and nation organisations, multi-nationals, national PKIs and global service providers. ADSS Server is normally discussed in terms of the following packages:

2.2.1 ADSS Enterprise Server Architecture

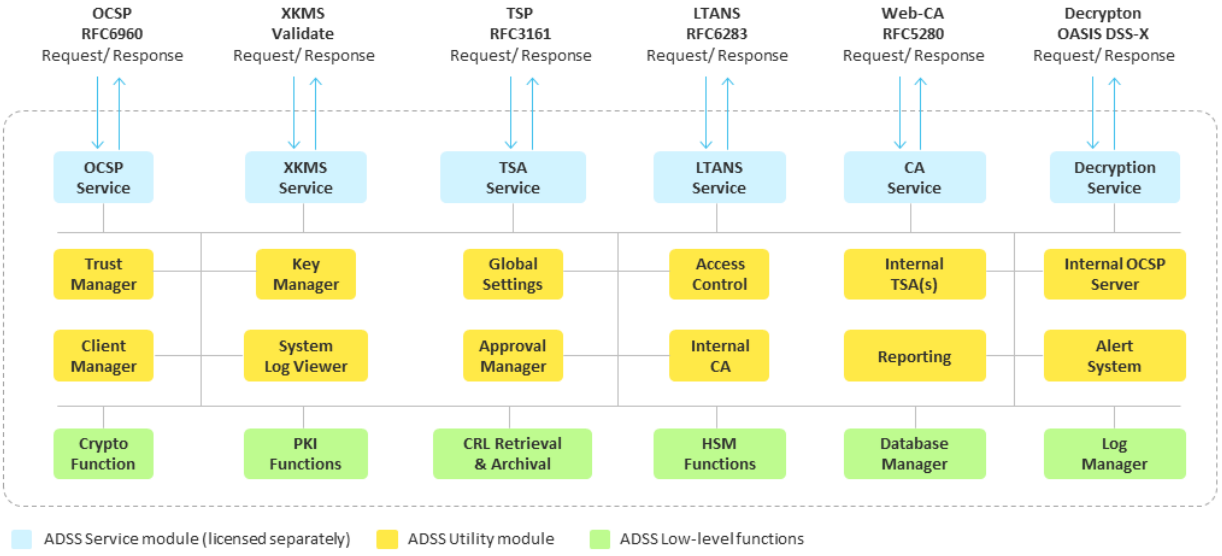
ADSS Enterprise Server is aimed at organisations wishing to primarily deploy digital signature creation and verification services. It typically uses the following services and modules:



The ADSS Certification Service can be included in this package if it is required to create and certify user-based signing keys for signing and verification services.

2.2.2 ADSS Infrastructure Server Architecture

ADSS Infrastructure Server is aimed at organisations that need to deploy infrastructure services such as a TSA, an OCSP, SCVP or XKMS VA, a CA or Archive Authority. The ADSS Infrastructure Server can be provided with the following services and modules:



As can be seen the architecture is so flexible that a bespoke solution can be easily created.

2.3 ADSS Server Web Services Interfaces

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

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

Service Name	Function
Signing Service ADSS digital signature service also includes:	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 PAdES, 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. Typically used with ADSS Go>Sign Desktop1 when local user signatures are needed and in this case the hashing is processes within ADSS Server, whilst signing is performed locally using the user's soft or token based key.
• Signature assembly	Embeds a digital signature within a PDF document – typically used with ADSS Go>Sign Desktop when it returns a signed hash value from the user's system and ADSS Server then embeds this 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, PAdES, CAdES and XAdES are supported. It also supports validation of digital certificates using simple or complex validation methods.
XKMS Service	Validates a certificate chain. Certificates can be validated against an existing defined CA chain or via advanced validation services using DPD & DPV. (Note: OCSP & SCVP services are not web-services)
Certification Service	Allows keys and certificates to be generated for users or other entities which can be used later for server-side signing of documents or passed to the user or used in roamed credential solutions for a specific application. (Note: This service does not interact with keys and certificates created in ADSS Server Key Manager using the ADSS Admin Console)
LTANS Service	Securely archives data or documents for long-term preservation. Each archive object is time stamped 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 the calling applications.
Decryption Service	Decrypts XML data using ADSS Server private keys. It is typically used in conjunction with ADSS Go>Sign Desktop where end-users sign and encrypt their document submissions. These encrypted objects can then be decrypted and the event audited via this web service. It provides good control for tender, health and similar solutions.

[1] The ADSS Go>Sign Desktop can be used to sign server created hashes OR hash and sign data locally.

Go>Sign Service	ADSS Go>Sign Service empowers business applications to perform document signing on user's machines using the credentials held either locally by the user or server-side keys. ADSS Go>Sign Service also enables business applications to show PDF documents to users using a server-side HTML-based Go>Sign Document Viewer. For more information on how to integrate the Go>Sign Service in a business application see ADSS-Go-Sign-Developers-Guide.pdf available within ADSS Client SDK package.
RA Service	Allows keys and certificates to be generated for network devices, end users and servers. The service is accessible using XML web services and SCEP protocol

2.4 Interfacing to the OCSP, SCVP and TSA Services

The ADSS Server OCSP, SCVP and TSA services are different to the other services and interface using traditional HTTP or HTTPS based requests/ responses from/ to client systems using the appropriate protocols (RFC 6960, RFC 5055, RFC 3161 and RFC 5816 respectively).

The table below summarises the function of these service interfaces. To make it easier to rapidly deploy a solution the ADSS Client SDK provides easy to use APIs for request creation and response parsing for the OCSP, SCVP and TSA services if a business application needs to programmatically access these services. Other standards compliant commercial or free APIs can be used to access these services.

Service Name	Function
TSA Service	Receives timestamp requests from one or more TSA clients, optionally authenticates the request and then creates and returns a timestamp token. The ADSS Server TSA Profiles define the way the TSA service operates.
OCSP Service	Receives OCSP certificate validation requests from one or more OCSP clients, optionally authenticates the request and then creates and returns the validation result, i.e. a "GOOD", "REVOKED" or "UNKNOWN" status for each CertID in the request. The ADSS Server validation policy defines the way the OCSP service operates.
SCVP Service	Receives SCVP certificate validation requests from one or more SCVP clients and optionally authenticates the request. As requested (a) the certificate path is determined using Delegated Path Discovery (DPD techniques) and then (b) the certificate chain is validated using basic or advanced validation processes. The validation result is created using the particular validation policy and information within the SCVP request.

2.5 Integration with Business applications

Business applications communicate with ADSS Server using standard web services calls or HTTP/S as explained above. The request is processed and ADSS Server returns an XML response wrapped in a SOAP message in a standard HTTP/S message or just as HTTP/S

For the web services interfaces there are defined schemas for the different request/response messages between the business application and ADSS Server. For the HTTP/S only protocols messages these are described in the relevant RFC. If ADSS Server cannot successfully parse or process the request it responds with an appropriate error message.

Various use cases are defined and these are described in the following sections of this manual:

- Signing Service (Sections 4 and 19)
    - Create an Empty Signature Field on a PDF Document

        - Sign a PDF document
        - Hash a PDF Document
        - Assemble a signature within a PDF document
    - Verification Service (Section 5)
        - Verify a signed document and validate an X509 certificate
    - TSA Service (Section 8)
        - Generate a timestamp token
    - XKMS Service (Section 9)
        - Validate an X509 digital certificate
    - OCSP Service (Section 7)
        - Validate an X509 digital certificate
    - SCVP Service (Section 10)
        - Validate an X509 digital certificate
    - Certification Service (Sections 6 and 20)
        - Generate a key pair and an associated digital certificate
        - Renew a key pair and an associated digital certificate
        - Retrieve a Private Key (PKCS#12 object) and Certificate
        - Delete a key pair and its corresponding certificate
        - Change the Authorisation Code associated with private key usage
    - LTANS Service (Section 11)
        - Securely archive, export or delete data from the ADSS Server long-term archive
    - Decryption Service (Section 12)
        - Decrypt encrypted XML data using private decryption keys held by ADSS Server
    - Go>Sign Service (Section 13)
        - Empowers business applications to perform document signing on user's machines using the credentials held either locally by the user or server-side keys
    - RA Service (Section 21)
        - Register and Revoke Certificates via XML web service interface and SCEP protocol

2.6 ADSS Server Admin Guide

Most commonly used administration tasks are performed using ADSS Server Admin Console. The ADSS Server Admin Guide describes how to configure the ADSS Server with the keys, certificates, trust anchors, profiles etc. that are required to process business application requests. Also, to make use of the sample programs and demos, an option in the ADSS Server installation wizard enables sample data to be inserted to simplify and automate the server configuration for testing purposes. The sample data is required for the sample programs and ADSS Server Test Tool to work.

2.7 Service URLs

To make use of the ADSS Server services, business applications send HTTP(S)/SOAP requests to the URLs shown below. This task is simplified using the ADSS Client SDK .Net and Java APIs:

Service Name	Default ADSS Address
Signing Service (Digital signature creation request over OASIS DSS and HTTP modes)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/signing/dss (For DSS Mode) https://<ADSS machinename>:8779/adss/signing/hdsi (For HTTP Mode) **TLS with Server authentication only:** https://<ADSS machinename>:8778/adss/signing/dss (For DSS Mode) https://<ADSS machinename>:8778/adss/signing/hdsi (For HTTP Mode) **Plain HTTP:** http://<ADSS machinename>:8777/adss/signing/dss (For DSS Mode) http://<ADSS machinename>:8777/adss/signing/hdsi (For HTTP Mode)
Signing Service (Empty Signature Field creation request over Ascertia XML and HTTP modes)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/signing/esi (For Ascertia XML Mode) https://<ADSS machinename>:8779/adss/signing/hesi (For HTTP Mode) **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/signing/esi (For Ascertia XML Mode) https://<ADSS machinename>:8778/adss/signing/hesi (For HTTP Mode) **Plain HTTP:** http://<ADSS machinename>:8777/adss/signing/esi (For Ascertia XML Mode) http://<ADSS machinename>:8777/adss/signing/hesi (For HTTP Mode)
Signing Service (Document hashing request over Ascertia XML and HTTP modes)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/signing/dhi (For Ascertia XML Mode) https://<ADSS machinename>:8779/adss/signing/hdhi (For HTTP Mode) **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/signing/dhi (For Ascertia XML Mode) https://<ADSS machinename>:8778/adss/signing/hdhi (For HTTP Mode) **Plain HTTP:** http://<ADSS machinename>:8777/adss/signing/dhi (For Ascertia XML Mode) http://<ADSS machinename>:8777/adss/signing/hdhi (For HTTP Mode)

Signing Service (Signature assembly request over XML and HTTP modes)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/signing/sai (For Ascertia XML Mode) https://<ADSS machinename>:8779/adss/signing/hsai (For HTTP Mode) **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/signing/sai (For Ascertia XML Mode) https://<ADSS machinename>:8778/adss/signing/hsai (For HTTP Mode) **Non TLS:** http://<ADSS machinename>:8777/adss/signing/sai (For Ascertia XML Mode) http://<ADSS machinename>:8777/adss/signing/hsai (For HTTP Mode)
Verification Service (Signature verification request over OASIS DSS mode)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/verification/dss (For DSS Mode) https://<ADSS machinename>:8779/adss/verification/hsvi (For HTTP Mode) **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/verification/dss (For DSS Mode) https://<ADSS machinename>:8778/adss/verification/hsvi (For HTTP Mode) **Non TLS:** http://<ADSS machinename>:8777/adss/verification/dss (For DSS Mode) http://<ADSS machinename>:8777/adss/verification/hsvi (For HTTP Mode)
TSA Service (TimeStampReq according to RFC 3161/5816)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/tsa **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/tsa **Non TLS:** http://<ADSS machinename>:8777/adss/tsa
XKMS Service (ValidateRequest according to XKMS 2.0)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/xkms **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/xkms **Non TLS:** http://<ADSS machinename>:8777/adss/xkms
OCSP Service (OCSPRequest according to RFC 6960)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/ocsp **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/ocsp **Non TLS:** http://<ADSS machinename>:8777/adss/ocsp

SCVP Service (CVRequest according to RFC 5055)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/scvp **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/scvp **Non TLS:** http://<ADSS machinename>:8777/adss/scvp
Certification Service (Certification request over Ascertia XML, EST and CMC modes). CMC mode is only supported over HTTPS with client server authentication	**TLS with client and server authentication:** https://<ADSS machinename>:8779/adss/certification/csi (Ascertia XML Mode) https://<ADSS machinename>:8779/adss/certification/cmc (CMC Mode) https://<ADSS machinename>:8779/.well-known/est/simpleenroll (EST Mode) https://<ADSS machinename>:8779/.well-known/est/simplereenroll (EST Mode) https://<ADSS machinename>:8779/.well-known/est/fullcmc (EST Mode) https://<ADSS machinename>:8779/.well-known/est/serverkeygen (EST Mode) https://<ADSS machinename>:8779/.well-known/est/cacerts (EST Mode) **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/certification/csi (Ascertia XML Mode) https://<ADSS machinename>:8778/.well-known/est/simpleenroll (EST Mode) https://<ADSS machinename>:8778/.well-known/est/simplereenroll (EST Mode) https://<ADSS machinename>:8778/.well-known/est/fullcmc (EST Mode) https://<ADSS machinename>:8778/.well-known/est/serverkeygen (EST Mode) https://<ADSS machinename>:8778/.well-known/est/cacerts (EST Mode) **No TLS:** http://<ADSS machinename>:8777/adss/certification/csi (Ascertia XML Mode) http://<ADSS machinename>:8777/.well-known/est/simpleenroll (EST Mode) http://<ADSS machinename>:8777/.well-known/est/simplereenroll (EST Mode) http://<ADSS machinename>:8777/.well-known/est/fullcmc (EST Mode) http://<ADSS machinename>:8777/.well-known/est/serverkeygen (EST Mode) http://<ADSS machinename>:8777/.well-known/est/cacerts (EST Mode)
LTANS Service (LTAPRequest according to draft-ietf-ltans-ltap-07)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/ltap (For XML mode) https://<ADSS machinename>:8779/adss/hltans (For HTTP mode) **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/ltap (For XML mode) https://<ADSS machinename>:8778/adss/hltans (For HTTP mode) **Non TLS:** http://<ADSS machinename>:8777/adss/ltap (For XML mode) http://<ADSS machinename>:8777/adss/hltans (For HTTP mode)

Decryption Service (EncryptRequest according OASIS DSS Encryption Profile)	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/decryption **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/decryption **Non TLS:** http://<ADSS machinename>:8777/adss/decryption
Go>Sign Service	**TLS with client-server authentication:** https://<ADSS machinename>:8779/adss/gosign/service **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/gosign/service **Non TLS:** http://<ADSS machinename>:8777/adss/gosign/service
RA Service (RA request over Ascertia XML and SCEP modes).	**TLS with client and server authentication:** https://<ADSS machinename>:8779/adss/ra/cri (Ascertia XML Mode) https://<ADSS machinename>:8779/adss/ra/scep (SCEP Mode) **TLS with server authentication only:** https://<ADSS machinename>:8778/adss/ra/cri (Ascertia XML Mode) https://<ADSS machinename>:8778/adss/ra/scep (SCEP Mode) **No TLS:** http://<ADSS machinename>:8777/adss/ra/cri (Ascertia XML Mode) http://<ADSS machinename>:8777/adss/ra/scep (SCEP Mode)

The ADSS Client SDK provides easy to use libraries written in the JAVA and .Net programming languages to minimise application integration development effort. Using these libraries business applications can be developed easily and quickly, whilst minimizing errors.

The use of ADSS Client SDK is strongly recommended to substantially reduce development time. There is no need to create XML writers and parsers for the web-service protocol. If it is necessary to write bespoke requests or to use a development language incompatible with Java or .Net APIs then it is important to ensure that the requests comply with the corresponding schema file and are compliant with the schema details detailed later in this manual

Sample source code is provided and the readme.html file in the ADSS Client SDK folder gives further information on the sample directory structure and how to run them.

2.8 ADSS Server Interface Schema

The following table 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	• **xmldsig-core-schema.xsd** (W3C specification for XML Digital Signature core schema) • **oasis-dss-core-schema-v1.0-os.xsd** (OASIS DSS core schema) • **oasis-dss-profiles-AdES-schema-v1.0-os.xsd** (OASIS DSS profile for advanced electronic signatures i.e. PAdES, CAdES, XAdES) • **oasis-dss-vissig-schema-v1.0-cd1.xsd** (OASIS DSS profile for visible signatures i.e. PDF/PAdES)

	• **adss-dss-extensions.xsd** (OASIS DSS extension schema – Ascertia specific enhancements in DSS protocol e.g. supporting authorized signatures)
Verification	• **xmldsig-core-schema.xsd** (W3C specification for XML Digital Signature core schema) • **oasis-dss-core-schema-v1.0-os.xsd** (OASIS DSS core schema) • **oasis-dssx-1.0-profiles-vr-cd1.xsd** (OASIS DSS-X profile for comprehensive signature verification report) • **oasis-dss-profiles-AdES-schema-v1.0-os.xsd** (OASIS DSS profile for advanced electronic signatures i.e. PAdES, CAdES, XAdES) • **dss_peppol_extensions.xsd** (OASIS DSS profile for PEPPOL compliance)
TSA	The protocol's ASN.1 specification is available from: http://www.ietf.org/rfc/rfc3161.txt
XKMS	• **xkms.xsd** (W3C XKMS schema) • **xmldsig-core-schema.xsd** (W3C specification for XML Digital Signature core schema) • **xenc-schema.xsd** (W3C XML Encryption schema – internally used by XKMS Schema) • **xkms_custom_extensions.xsd** (Ascertia extension in W3C XKMS schema e.g. sending request authentication parameters like Originator ID) • **xkms_peppol_extensions.xsd** (XKMS extensions for PEPPOL compliance) The W3C specification is available at from: http://www.w3.org/TR/xkms2/
OCSP	The protocol's ASN.1 specification is available from: http://www.ietf.org/rfc/rfc6960.txt
SCVP	The protocol's ASN.1 specification is available from: http://www.ietf.org/rfc/rfc5055.txt
Certification	• **adss-dss-extensions.xsd** (ADSS Certification Service schema – Ascertia proprietary protocol) For CMC the protocol's ASN.1 specification is available from: http://www.ietf.org/rfc/rfc5272.txt For EST the protocol's ASN.1 specification is available from: https://www.ietf.org/rfc/rfc7030.txt
LTANS	• **ltan.xsd** (IETF LTANS schema) • **ers.xsd** (ERS schema - Evidence Record Structure) The XML specification is available from: http://tools.ietf.org/html/draft-ietf-ltans-ltap-07
Decryption	• encryption_profile_0.4.xsd (OASIS DSS profile for data encryption and 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
RA	• **adss-ra.xsd** (ADSS RA Service schema – Ascertia proprietary protocol)

2.8.1 WSDL Files

Service	Schema File
Signing	• **dss.wsdl** (WSDL for OASIS DSS protocol)
Verification	• **dss.wsdl** (WSDL for OASIS DSS protocol)
XKMS	• **xkms.wsdl** (WSDL for W3C XKMS specification)
Certification	• **certification.wsdl** (WSDL for ADSS Certification Service – based on Ascertia proprietary protocol)
LTANS	• **ltap.wsdl** (WSDL for IETF LTAP protocol)
Decryption	• **encryption.wsdl** (WSDL for OASIS DSS profile for data encryption and decryption)
Attribute Authority	• **aa.wsdl** (WSDL for attribute authority based on Ascertia proprietary protocol)
Registration Authority	• **ra.wsdl** (WSDL for registration authority based on Ascertia proprietary protocol)

2.8.2 Protocol Dependencies

There are a number of schema files which are referenced by the protocols implemented by ADSS Server. These schemas and their elements are not directly used by any of service in ADSS Server but these may be needed by clients to be able to compile and generate code at their end.

- oasis-sstc-saml-schema-protocol-1.1.xsd
- saml-schema-assertion-2.0.xsd
- schema.xsd
- soap-envelope.xsd
- XAdES.xsd

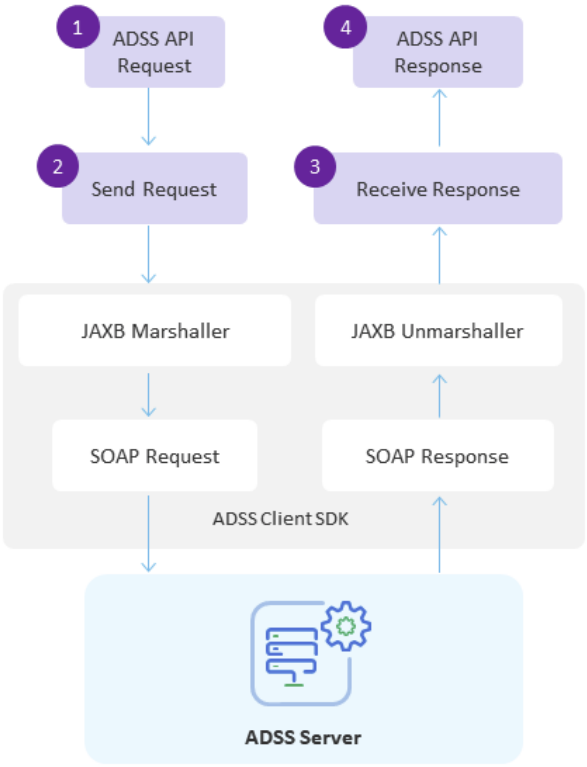
2.9 ADSS Client SDK (Java and .Net versions)

The ADSS Java and .Net libraries contain all the necessary components to create HTTP and web services requests that are understood by ADSS Server. The requests may be based upon open standards (e.g. OASIS) or they may use the faster Ascertia defined protocols.

In the main body of the document the various API methods are briefly described with examples using a .Net (C#) syntax. The equivalent Java calls are almost identical so this should not present a problem for Java developers. In addition to this, full documentation of the various classes and methods is available in the JavaDoc (Java) and SandCastle (C#) class documentation which comes as part of the ADSS Client SDK download.

2.9.1 Using the Java API

The Java API components performs behind the scenes 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 build a working application.



[Using non-default XML Parser and XML Transformer implementations with ADSS Client SDK (Java)]

A system property flag is used to instruct the ADSS Client SDK to use abstract classes for the XML Parser and XML Transformer implementation instead of using proprietary classes. The system property can be set as:

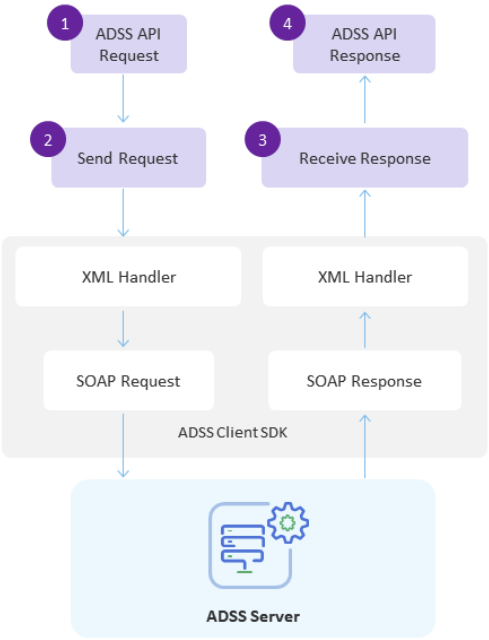
```
System.setProperty("com.ascertia.adss.client.api.xml.impl","SYSTEM");
```

By setting this property, business applications can specify the available XML Parser and XML Transformer implementation. The ADSS Client SDK checks this property, and then loads the system default implementation OR the implementation requested via the following system properties:

```
System.setProperty("javax.xml.parsers.DocumentBuilderFactory",
  "<DocumentBuilderFactory implementation class>");
```

2.9.2 Using the .Net API

Similarly the .Net API components perform the XML marshalling and SOAP message handling:



2.9.3 HTTP/S Protocol API Code

The Java and .Net APIs also provide classes for sending requests and receiving responses for the HTTP/S only services.

In the following sections of this manual, the API classes relevant to each service are discussed.

3 Message, Request and Response Classes

This section provides descriptions of the Message, Request and Response classes with their respective methods which are inherited by other classes described in subsequent sections of this guide.

The following sub-sections briefly describe these classes but full documentation is also available in the JavaDoc (Java) and SandCastle (C#) class documentation which comes as part of the ADSS Client SDK download.

The Java API provides the required classes under the package:

```
com.ascertia.adss.client.api
```

The .Net /C# API provides the required classes under the namespace:

```
Com.Ascertia.ADSS.Client.API
```

3.1 Message Class

The Message class is an `abstract` class that acts as a base class to the Request and Response classes. It has the following three methods:

Message Class Method	Purpose
`ToString()returns string`	Returns the textual representation of an XML output. It overrides the Object.ToString() method.
`WriteTo(Stream outStream)`	Writes the XML output to the given output stream.
`WriteTo(string filePath)`	Writes the XML output to the given file path.

3.2 Request Class

The Request class provide a number of common methods that are inherited by the more specific request classes.

The following is a list of methods of the Request Class (in addition to the `ToString` and `WriteTo` methods described above):

Request Method	Purpose
`Send(string url) returns Response`	Sends the request to the ADSS service on the specified URL. This is a `virtual` method that is sometimes overridden by the specific service request class. The resultant `Response` object is that of the corresponding service response class and contains parsed response or error information.
`SetProxy(string host, int port) or` `SetProxy(string host, int port, string userName, string password, bool digest)`	Specifies proxy information if the client application is behind a proxy. Two variants are available for this method. • Com.Ascertia.ADSS.Client.API.Request.PROXY_AUTHENTICATION • Com.Ascertia.ADSS.Client.API.Request.SERVER_AUTHENTICATION
`SetRequestID(string requestID)`	Unique ID assigned to the request. **Note:** Request ID can contain only alphanumeric chars, a hyphen, and dash other chars are not allowed.
`SetRequestRetries(int requestRetries)`	Specifies the number of client retries when making the request to the service.
`SetRespondAddress(string respondAddress)`	Specifies the address where the response will be sent. This is currently not implemented.

`SetSigningCredentials (X509Certificate2 requestSigningKey)` or `SetSigningCredentials(string pfxFilePath, string password)`	Specifies the signing credentials that will be used to sign the request. This is provided either as a: • private key and certificate chain, or a • PFX or PKCS#12 file. The latter method is defined as `virtual` and may be overridden.
`SetSigningMode(string signingMode)`	Specifies the XML request signing mode. The possible values are: - `SIGNING_MODE_DETACHED` - `SIGNING_MODE_ENVELOPED` - `SIGNING_MODE_ENVELOPING` `SIGNING_MODE_ENVELOPED` is the default mode if not set by the client.
`SetSoapVersion(string soapVersion)`	Specifies the version of the SOAP message. The possible values are: - `SOAP_VERSION_1_1` - `SOAP_VERSION_1_2` If the client does not set the soap version then `SOAP_VERSION_1_1` used as a default. (Note that OASIS DSS requires the use of `SOAP_VERSION_1_2` to be compliant with the standard).
`SetSslClientCredentials( X509Certificate2 sslKey)` or `SetSslClientCredentials( string pfxFilePath, string password)`	Specifies the TLS client credentials that will be used for TLS client authentication, either as a: • Private key and certificate chain, or a • PFX or PKCS#12 file. • Password to decrypt the PKCS#12 file **Note:** The TLS Server authentication certificate must include machine Name/Domain Name/IP Address of the relevant ADSS Server in certificate's Common Name (and also as SAN extension if there are multiple domain names).
`SetSslTrustStore(String truststorePath, String truststorePassword)`	Specifies the trust store for the TLS server certificate.
`SetTimeout(int timeout)`	Specifies the communication timeout for the request in seconds. Default value is 60 secs.
`SetApplicationName (String)`	Set client's application name.
`SetVerifyResponse(bool verifyResponse)`	Specifies whether the client application needs to verify the service response signature or not. By default, it does not verify the response.

3.2.1 Using SSL /TLS

To use SSL/TLS a secure SSL/TLS channel must be established and this requires that the following are properly set up:

- Make sure the ADSS Server, TLS Server authentication certificate, configured in Global Settings > System Certificates, has the ADSS Server machine name/IP as its common name (CN).
- Make sure the client application can trust the ADSS Server TLS Server authentication certificate.

- Make sure the issuer of the client authentication certificate is registered within the ADSS Server Trust Manager with the purpose "CA for verifying TLS client certificates".
- Make sure the ADSS Server Windows/Daemon services have been restarted after making the above configurations.

For ADSS Client SDK .Net, before calling the SDK classes this requires the following to be properly set up:

```
ServicePointManager.SecurityProtocol = SecurityProtocolType.Ssl3 |
SecurityProtocolType.Tls12 | SecurityProtocolType.Tls11 |
SecurityProtocolType.Tls;
```

3.3 Response Class

The Response class provides a number of common methods that are inherited by the more specific response classes.

The following is a list of methods of the Response Class (in addition to the ToString and WriteTo methods described above):

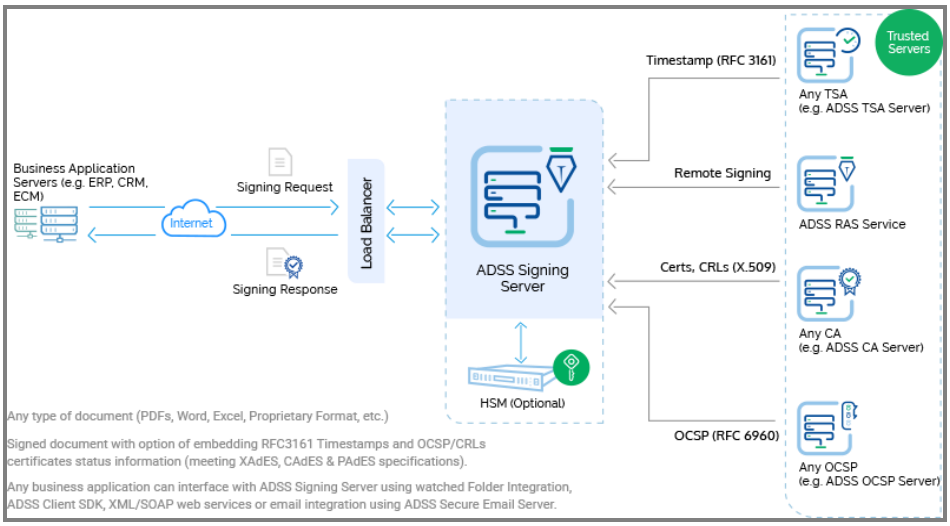
Response Method	Purpose
`ContainsException() returns bool`	Returns true if there was an error while processing the request. If so, the Exception object is also set.
`GetErrorCode() returns int`	Returns the service specific error code in the case of request process failure.
`GetErrorMessage() returns string`	Returns the user friendly error message in the case of request process failure.
`GetException() returns Exception`	Returns the Exception object for the case of request process failure.
`GetRequestId() returns string`	Returns the request Id of the corresponding request.
`GetSigningCertificates() returns X509certificate[]`	Returns the XML response signing certificate chain.
`GetStatus() returns string`	Returns the status of the request. The actual text will vary depending upon the service and mode of operation (e.g. Ascertia mode, OASIS DSS mode etc.) For Ascertia mode, the status message is either `Success` or `Failed`. For OASIS DSS mode, the returned status message is the `Result Major` status, which is one of the following: - `Success` - `Requester Error` - `Responder Error` - `Insufficient Information` For the LTAP (long-term archiving protocol) the returned message is either `granted` or `rejected`.
`IsSuccessful() returns bool`	Returns `true` if request processing has completed successfully and does not indicate for example whether the request has a positive or negative result (e.g. whether a certificate validated or a signature verified). In OASIS DSS mode, for example, as a minimum, it is necessary to look at the `Result Major` and `Result Minor` status messages to determine if all the signatures were fully valid.

4 ADSS Signing Service

The ADSS Server Signing Service provides the following types of signature services:

- OASIS DSS Compliant Document Signing (PDF, XML and PKCS7 signatures)
- OASIS DSS AdES Profile for Advanced Signature Formats (XAdES, CAdES and PAdES)
- Digital signatures with authorised remote signing through RAS/SAM
- Ascertia Specific Signing Utility Classes for:
    - Document Hashing and Assembly
    - (PDF) Visible Signature Field creation & optional signing

Business Client Applications send requests to ADSS Server and receive responses back. Normally most signing parameters do not need to be sent in the request as they are already set up in a Signing Profiles at ADSS Server.



All the Trust Services shown above can be provided either by ADSS Server or can be external.

The protocol used for the OASIS compliant services is based on the OASIS Digital Signature Service Core Protocols, Elements and Bindings specification (oasis-dss-core-spec-v1.0-os). Messages are wrapped in a SOAP message and sent using HTTP; or sent directly using HTTP POST without SOAP (in enhanced performance mode).

The protocol used for the Ascertia specific (signing) utility classes is Ascertia proprietary and uses either SOAP or HTTP POST. The protocol schemas are described in section 19. Various signing use cases are possible using the ADSS Signing Service and these are also described in section 19.

4.1 Digital Signature Standards

Digital signatures created by ADSS Server are open standards compliant and can include timestamps and revocation information. The following signature types are supported:

Supported Signature Types (ascertia.com)

4.2 Setting up Signing Service Profiles

The ADSS Signing Service requires that Signing Profiles are defined at ADSS Server. These profiles identify the type of document (e.g. PDF) and type of signature required (e.g. PDF certifying signature with embedded timestamp and revocation information) and any other settings that may be required.

Refer to the following online admin guide for an explanation of Signing Profile settings:

Step 4 - Configuring Signing Profile (ascertia.com)

4.3 The Signing Service API

In order to simplify the use of the OASIS DSS and Ascertia proprietary HTTP protocols, a Signing Service API is provided as part of the ADSS Client SDK.

This API consists of the following Request and Response classes:

- Signing: Used for XML and PKCS7/CMS signatures.
- PDF Signing: Used for creating PDF signatures. For quick introduction, refer to **"A Quick Guide for using PDF Signatures within ADSS Server"** guide.
- Empty Signature Field Creation: Used to create a blank signature field within a PDF document and optionally to sign the PDF document.
- Document Hashing: Used where ADSS Server performs the hash operation but the signing operation is performed on the client side.
- Signature Assembly: Used after client side signing to assemble the completed signature

4.4 Signing Request and Response Classes

The Signing Request Class is used with any file type to add a PKCS7/CMS signature and with XML documents to add an XML digital signature.

The following constructor is used to build the initial Signing Request message. There are different variants depending upon the source of the document to be signed i.e. whether the `document` is referenced as a `file path`, or provided directly as a `Stream or byte[] or Base64 format document hash`.

```
var signingRequest = new SigningRequest(clientID, document,
documentMimeType);
```

4.4.1 Signing Request methods

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send, SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

In addition, the following methods are specific to the Signing Request Class:

Signing Request Method	Purpose
`AddDocument(string filepath or byte[])`	Specifies additional documents or data to be signed. This method may be called multiple times if multiple documents are covered by the same request.
`AddSignedAuthorisation( string, byte[] or XMLDocument)`	Adds a single Signed Authorisation file to the signing request. This method can be called multiple times when there is more than one authoriser. Authorisation files (ready for signing by the authorisers) can be created with the Authorisation Data utility class - see section 17.1. *Authorised signing requests are only supported using web-services.* *Authorisation profiles are described in the online admin guide:* Authorisation Profiles (ascertia.com)

`RequestContentTimeStamp(bool)`	If set to true, this requests the addition of a content time stamp into the signature.
`requestCounterSignature(bool)`	If set to true, this requests the addition of counter signature as an unsigned attribute in the signature.
`RequestSigningTime(bool)`	If set to true, this request the addition of the signing time into the signature.
`SetCertificateAlias (string)`	Specifies a certificate alias to identify the ADSS Server held key that will be used to sign the document. It is optional in case of remote signing.
`SetCertificatePassword (string)`	Supplies the password to use (unlock) the private key associated with the above managed certificate. Moreover, the password would be used to authenticate user at the time of remote authorised signing.
`SetProfileId (string)`	Specifies the Signing Profile identifier.
`SetRequestMode (Int32)`	Specifies the request mode (one of): - `Request.HTTP (Default mode)` - `Request.DSS` In high performance HTTP mode, the document is placed in the HTTP body while other information is placed in the HTTP header. In OASIS DSS mode, a message is created by following the OASIS DSS specification. This is then wrapped in a SOAP message and sent using the HTTP protocol. **Note:** The mode parameter is important as it can affect how the response status is handled.
`SetSignatureForm (string)`	This method is used when upgrading an existing signature to an extended form. These signature types are described here: Supported Signature Types (ascertia.com) For example, to extend a basic signature to include a timestamp the parameter required is: `SigningRequest.SIGNATURE_FORM_ES_T`
`SetCity (string)`	Specifies the city used as Signed attribute in XAdES signature generation.
`SetStateOrProvince (string)`	Specifies the state/province used as a signed attribute in CAdES/XAdES signature generation.
`SetPostalCode (string)`	Specifies the postal code used as a signed attribute in CAdES/XAdES signature generation.
`SetCountryName (string)`	Specifies the country name used as a signed attribute in CAdES/XAdES signature generation.
`SetSignerRole (string)`	Specifies the signer role used as a signed attribute in PAdES/CAdES/XAdES signature generation.
`SetCommitmentTypeIdentifier (string)`	Specifies the commitment type identifier used as a signed attribute in PAdES/CAdES/XAdES signatures. If the signature type is PAdES then it is tightly associated with signature policy settings. It will be included when explicit signature policy is available. The following commitment types are supported: - `PROOF_OF_APPROVAL`

	- PROOF_OF_CREATION - PROOF_OF_DELIVERY PROOF_OF_ORIGIN PROOF_OF_RECEIPT - PROOF_OF_SENDER
SetCommitmentTypeQualifier(string)	Specify the commitment type qualifier to include additional qualifying information on the commitment made by the signer. Used in conjunction with SetCommitmentTypeIdentifier(String) method. **Note:** If null is set than the value set for setCommitmentTypeIdentifier(String) is used for this parameter.
SetSignaturePolicyOID(string)	Specifies the signature policy OID used as Signed attribute in PAdES/CAdES/XAdES signature generation. Used in conjunction with SetLocalHash method.
SetSignaturePolicyURI(string)	Specifies the signature policy URI used as Signed attribute in PAdES/CAdES/XAdES signature generation. Used in conjunction with SetLocalHash method.
SetSignaturePolicyUserNotice(string)	Specifies the signature policy user notice used as Signed attribute in PAdES/CAdES/XAdES signature generation. Used in conjunction with SetLocalHash method.
SetPolicyDocument(byte[])	Specifies the signature policy document used as Signed attribute in PAdES/CAdES/XAdES signature generation. Used in conjunction with SetLocalHash method.
SetDocumentFormat (string)	Specifies the document format of the data object used as a signed attribute in the CAdES/XAdES signature generation. In case of CAdES/MS Office signatures document format must be an OID e.g. **"1.2.3.4.5"** **Note:** MS Office signatures document format supported only on the HTTP interface. In case of XAdES signatures document format can be any string.
RequestContentTimeStamp(bool)	Flag to add the content timestamp in the PAdES/CAdES/XAdES signature.
RequestSigningTime (bool)	Flag to add the signing time in the PAdES/CAdES/XAdES signature.
RequestCounterSignature(bool)	Flag to create counter signature in PAdES/CAdES/XAdES signature.
SetLocalHash(bool)	If set to true this causes the client SDK to locally hash the document. Just the hash value is then sent to ADSS Server for signing. Local hashing can reduce network data transfer and maintain document confidentiality if this is an issue.
SetDocumentLock(bool)	If set to true the document will be locked after signing operation, only works when local hash is set to true.
SetListOfFieldsToLock(ArrayList a_listOfFieldsToLock)	Specifies the list of fields to be locked after signing operation, only works when local hash is set to true.
AddFieldToLock(String a_strFieldName)	Specifies the field name to be locked after signing operation, only works when local hash is set to true.

`SetLocalDigestAlgorithm(string)`	Specifies the hash algorithm to be used in the case of local document hashing.
`SetSignerCertificate(string or X509Certificate)`	Specifies the signing certificate in case of: • Local document hashing. • Common name to be used for “Signed By” attribute in signature. • When routing request to RAS/SAM Gateway as user key is not available at the relevant gateway to generate the signature structure. • It is optional in case of remote signing.
`SetSignerCertificateChain(X509Certificate[])`	Set the signing certificate chain. It is optional in case of remote signing.
`SetSignatureMode(string)`	Specifies the signature mode e.g. Enveloped/Enveloping
`SetSignatureHash(bool)`	If set to true the final hash would be sent to ADSS server. If false, only structure of the document would be sent and final hash would be computed by ADSS Server.
`SetRevocationInformationOCSP(List)`	Set revocation information in form of OCSP Responses.
`SetRevocationInformationCRL(List)`	Set revocation information in form of CRLs.
`SetSigningElementName (string)`	XML part signing based on XML element names or XPath expressions. Multiple values can be comma separated e.g. ‘ContractName,ContractDate’ or ‘//ContractName,//ContractDate’
`IsDocumentSigned() returns bool`	Checks if the provided document already signed or not. This method is helpful in deciding whether to send an already signed document for signing to ADSS Server or skip it.
`setUserID (string)`	User ID used to maintain unique user identification at the RAS/SAM server. It is a mandatory parameter in the remote authorised signing request. **Note:** It is only supporting on HTTP interface.
`setDataToBeDisplayed (string)`	Display message to help the remote authorised signer to know what he is going to sign. This message will be shown on the user mobile device. Message should be in base64 format to maintain multilingual characters. It is an optional parameter in the remote authorised signing request. **Note:** It is only supporting on HTTP interface.
`setDocumentID (string)`	Set document ID for identification of a document.. **Note:** It is only supporting only on HTTP interface.
`setDocumentName (string)`	A friendly name of the document. Its an optional parameter and used only in case of authorised remote signing using RAS/SAM services. Note: It is only supporting only on HTTP interface.
`SetSamlAssertion (string)`	SAML assertion would be used to authenticate user at the time of remote authorised signing. It is an optional parameter in the remote authorised signing request. **Note:** It is only supporting on HTTP interface.

`EmbedSignatures (List<byte[]>)`	This method takes signatures in a list for assembly in the signing documents. Assembly operation will perform in the client SDK. **Note:** This method will be applicable on the time of local hash with remote authorised signing. ADSS Signing Server supports remote authorised signing on HTTP interface only.
`setTransactionID (string)`	Set transaction ID to check status of the pending signing request.
`setXmlObjectID (string)`	Set Object ID for identification of xml document content. **Note:** It is only supports on HTTP interface.
`SetDocumentHash (bool)`	If set to `true` this causes the client SDK to get hash of the document. Just the hash value is then sent to ADSS Server for signing. Document hashing can reduce network data transfer and maintain document confidentiality if this is an issue.

4.4.2 Sending the Signing Request

Once the signing request message has been constructed using the above methods, it can be sent to ADSS Server using the following call:

```
var signingResponse = (SigningResponse)signingRequest.Send(string URL);
```

The `URL` is that of the Signing Service e.g. http://machine-name:8777/adss/signing/dss or http://machine-name:8777/adss/signing/hdsi when using HTTP mode.

4.4.3 Example of creating and sending a Signing Request

```
// Build Signing Request
var signRequest = new SigningRequest(clientID, document, requestType);
signRequest.SetProfileId(profileID);
signRequest.SetCertificateAlias(certAlias);
signRequest.SetRequestMode(requestMode);

// Send request to ADSS server
var signResponse = (SigningResponse)signRequest.Send(serviceAddress);
```

4.4.4 Signing Response Methods

The following methods are inherited from the generic Response and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetRequestID, GetSigningCertificates, GetStatus, IsSuccessful.`

In addition, the following methods are specific to the Signing Response Class:

Signing Response Method	Purpose
`GetDocument() returns byte[]`	Returns the signed document.
`GetDocuments() returns ArrayList`	Returns all the signed documents.
`GetProfileId() returns string`	Returns the Signing Profile ID used by the ADSS Signing Service to process the request.

`GetResultMajor() returns string`	(Used in OASIS DSS protocol mode). The Result Major status message provides the main result from processing the signing request. The status can be one of the following: - `Success` - `Requester Error` - `Responder Error` - `Insufficient Information`
`GetResultMinor() returns string`	(Used in OASIS DSS protocol mode). The possibilities for the Result Minor status messages depend upon the Result Major message. For Result Major = Success, Result Minor values are: - `valid:signature:OnAllDocuments` - `valid:signature:NotAllDocumentsReferenced` - `invalid:IncorrectSignature` - `valid:signature:HasManifestResults` - `valid:signature:InvalidSignatureTimestamp` For Result Major = Requester Error, Result Minor values are: - `ReferencedDocumentNotPresent` - `MoreThanOneRefUriOmitted` - `InvalidRefUri` - `NotParseableXMLDocument` - `NotSupported` - `InappropriateSignature` `(Other values are possible)` For Result Major = ResponderError, Result Minor values are: - `GeneralError` - `KeyLookupFailed` `(Other values are possible)` For Result Major = Insufficient Information, Result Minor values are: - `CrlNotAvailable` - `OcspNotAvailable` - `CertificateChainNotComplete`
`GetXmlDocument() returns XmlDocument`	Returns the signed XML document.
`PublishDocument(string /Stream)`	Publishes the signed document to the specified path or stream.
`GetTransactionID() returns string`	Return pending signing request transaction ID to check its status later on.
`GetHtmlPage() returns String`	Returns a RAS internal authorisation HTML page. This HTML is returned by Signing Service when configured for remote authorisation signing with RAS and an Go>Sign mobile is configured at RAS for user authentication/ authorisation. The business application then needs to load this page in browser window.

`GetRedirectURI returns String`	Returns the redirect URI of the external IdP or RAS internal authorisation page. This URI is returned by Signing Service when configured for remote authorisation signing with RAS and an IdP is configured at RAS for user authentication/authorisation. The business application then needs to navigate the user to this URI.

4.5 PDF Signing Request and Response Classes

The PDF Signing Request Class is used to add signatures to PDF files.

The following constructor is used to build the initial PDF Signing Request message. There are different variants depending upon the source of the document to be signed i.e. whether the `document` is specified as a `file path, Stream or byte[].`

```
var pdfSigningRequest = new PdfSigningRequest(string clientID, document);
```

4.5.1 PDF Signing Request methods

The following methods are inherited from the Sign Request class and described in section 4.4.1:

`AddDocument, AddSignedAuthorisation, RequestContentTimeStamp, requestCounterSignature, RequestSigningTime, SetCertificateAlias, SetCertificatePassword, SetCommitmentTypeIdentifier, SetProfileId, SetRequestMode, SetSignatureForm/`

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send (overridden), SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

The following additional methods are specific to the PDF Signing Request:

PDF Signing Request Method	Purpose
`AddSignaturePosition(<visible signature parameters>)`	Supplies a list of parameters to fully specify a signature position and appearance.
`AddEmptySignatureFieldPosition(<visible signature field parameters>)`	Supplies a list of parameters to fully specify an empty signature field position. It can be used only for local hashing.
`SetCertifyPermission(int)`	Sets the certifying signature permission value for only local hashing. The value can be one of: • `PdfSigningRequest.CERTIFIED_FORM_FILLING` • `PdfSigningRequest.CERTIFIED_FORM_FILLING_AND_ANNOTATIONS` • `PdfSigningRequest.CERTIFIED_NO_CHANGES_ALLOWED` **Note:** When generating PAdES_LT / PAdES_LTV signatures in case of local hashing then certify permission CERTIFIED_NO_CHANGES_ALLOWED will be ignored.
`SetHashAlgorithm(string)`	Specifies the hash algorithm OID to be used in the case of local document hashing.
`SetLocalHash(bool)`	If set to `true` this causes the client SDK to locally hash the document after possibly applying various signature fields

	(e.g. signature appearance, signed by etc.). Just the hash value is then sent to ADSS Server for signing. Local hashing can reduce network data and maintain PDF document confidentiality if this is an issue. **Note:** Supported hashing algorithms for client side hashing are SHA1, SHA256, SHA384, SHA512
`SetSignatureAppearance(string or byte[])`	Specifies a signature appearance XML file.
`SetSignerCertificate(string or X509Certificate)`	Specifies the signing certificate in case of: • Local document hashing. • Common name to be used for "Signed By" attribute in signature. • When routing request to RAS/SAM Gateway as user key is not available at the relevant gateway to generate the signature structure. • It is optional in case of remote signing.
`SetWaterMarkInfo(PdfWaterMarkInfo)`	Specifies the water mark info used to create water mark in the PDF document before sending the signing request to ADSS Server.
`SetFormFieldValue(String formFieldName,String formFieldValue)`	Specifies the PDF form field value before sending the signing request to ADSS Server.
`SetPadesSignatureType(String PadesSignatureType)`	Specifies the PAdES signature type computed. Possible values are PAdES-BES, PAdES-T and PAdES-LTV
`SetVerificationServiceAddress(String VerificationServiceAddress)`	Specifies the Verification Service URL to verify and enhance the PAdES signatures to PAdES-LTV. It would be used when the document hash locally computed.
`SetVerificationProfile(String VerificationProfile)`	Specifies Verification Profile ID to process the verification request accordingly. It would be used when the document hash locally computed.
`SetTimeStampServiceAddress(String TimeStampServiceAddress)`	Specifies the Time-Stamp Service URL to time-stamp PAdES-LTV signatures. It would be used when the document hash locally computed.
`SetTimeStampPolicyId(String TimeStampPolicyId)`	Specifies Time-Stamp Policy ID to timestamp the signatures accordingly. It would be used when the document hash locally computed.
`IsDocumentSigned() returns bool`	Checks if the provided PDF document is already signed or not. This method is helpful in deciding whether to send an already signed document for signing to ADSS Server or skip it.
`GetCertifyPermission() returns int`	Returns the certify permission applied on the PDF document. The return value can be one of: - `CERTIFIED_FORM_FILLING` - `CERTIFIED_FORM_FILLING_AND_ANNOTATIONS` - `CERTIFIED_NO_CHANGES_ALLOWED` NO_RESTRICTIONS (No Certify Permission applied)

4.5.2 Other PDF Signing Request Methods

In addition to the above, the following methods are available for specifying various aspects of the PDF signatures. Again these are documented in the JavaDoc and Sandcastle documentation:

```
SetCompanyLogo, SetContactInfo, SetFontrepository, SetHandSignature,
SetSignedBy, SetSigningArea, SetSigningField, SetSigningLocation,
SetSigningPage, SetSigningReason, SetWatermarkInfo,
SetSignatureAppearanceId, SetLocalHash, SetSignatureDictionarySize
```

4.5.3 Sending the PDF Signing Request

Once the signing request message has been built using the above methods, it can then be sent to ADSS Server using the following method call:

```
var pdfSigningResponse = (PdfSigningResponse)pdfSigningRequest.Send(string URL);
```

The `URL` is that of the Signing Service e.g. http://machine-name:8777/adss/signing/dss or http://machine-name:8777/adss/signing/hdsi when using HTTP mode.

4.5.4 Example of building and sending a PDF Signing Request

```
// Build PDF Signing Request
var signRequest = new PdfSigningRequest(clientID, document);
signRequest.SetProfileId(profileID);
signRequest.SetCertificateAlias(certAlias);
signRequest.SetRequestMode(reqMode);

// Set these parameters only if local hashing, otherwise hash at server
if (localHash == true)
{
    signRequest.SetSignatureAppearance(sigAppearance);
    signRequest.SetSignedBy(signedBy);
    signRequest.SetSigningField(signingField);
    signRequest.SetLocalHash(true);
}

// Send request to ADSS server
var signResponse = (PdfSigningResponse)signRequest.Send(serviceAddress);
```

4.5.5 PDF Signing Response Methods

All PDF Signing Response methods are inherited from the Message, Response and Signing Response classes described in sections 3 and 4.4.3, and in the JavaDoc/Sandcastle class documentation.

4.6 Empty Signature Field Request and Response Classes

The Empty Signature Field Request Class is used to request creation of an empty signature field in a PDF document and optionally to sign the document if that is required.

The following constructor is used to build the initial Empty Signature Field Request. There are two variants depending upon whether the `pdfDocument` is specified as a `file path or byte[]`.

```
var sigFieldRequest = new EmptySignatureFieldRequest(string clientID,
pdfDocument);
```

4.6.1 Empty Signature Field Request Method

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

```
ToString, WriteTo, Send, SetProxy, SetRequestID, SetRequestRetries,
SetSigningCredentials, SetSigningMode, SetSoapVersion,
SetSSLClientCredentials, SetTimeout, SetVerifyResponse.
```

In addition, the following methods are specific to the Empty Signature Field Request class:

Empty Signature Field Request Method	Purpose
`AddSignaturePosition(<visible signature parameters>)`	Supplies a list of parameters to fully specify a signature position and appearance.
`OverrideProfileAttribute (string attributeID, string/byte[])`	Allows specific signing attributes to be overridden if signing is required/allowed as part of empty signature field creation. For example 'Signing Reason' could be overridden.
`SetProfileId(string)`	Specifies the Signing Service Profile ID to be used for servicing the Signature Field Creation.
`SetSigningCertificatePassword (string)`	Specifies the P12/PFX file password for the PDF signing operation if this is required.
`SetSigningInfo(string profileId, string certAlias)`	Identifies the Signing Profile and references the certificate/server key to be used for signing the PDF (if this is required).

4.6.2 Sending the Empty Signature Field Request

Once the Empty Signature Field request message has been built using the above methods, it can then be sent to ADSS Server using the following method call:

```
var sigFieldResponse =
(EmptySignatureFieldResponse)sigFieldRequest.Send(string URL);
```

The `URL` is that of the Empty Signature Field Request Service e.g. http://machine-name:8777/adss/signing/esi

4.6.3 Example of an Empty Signature Field Request

```
// Constructing Empty Signature Field Request
var sigFieldRequest = new EmptySignatureFieldRequest (clientID, pdfFile);
sigFieldRequest.SetProfileId (profileID);
sigFieldRequest.SetSigningInfo (signingProfile, certAlias);
sigFieldRequest.OverrideProfileAttribute (EmptySignatureFieldRequest.SIGNING_REASON, signingReason);
sigFieldRequest.SetRequestMode (requestMode);

// Send the constructed request to ADSS server
var sigFieldResponse = (EmptySignatureFieldResponse) sigFieldRequest.Send (serviceAddress);
```

4.6.4 Empty Signature Field Response Methods

The Empty Signature Field Response methods are mainly inherited from the Message and Response classes and are described in section 3 and in the JavaDoc/Sandcastle class documentation.

In addition, the following methods are specific to the Empty Signature Field Response Class:

Empty Signature Field Response Method	Purpose
`GetDocument() returns string`	If signing information is also provided as part of the empty signature field request, then the service first creates the empty signature field and then signs the same field. If signing information is not provided, then the service only returns the PDF document with the empty signature field.
`GetProfileId() returns string`	Returns the Signing Profile ID used by the ADSS Signing Service to process the request.
`PublishDocument(string /Stream)`	Publishes the signed document to the specified path or stream.

4.7 Document Hashing Request and Response classes

The Document Hashing classes are used together with Signature Assembly where ADSS Server is first requested to create the hash for a PDF document, and then the client signs the hash and finally ADSS Server assembles the document signature.

The following constructor is used to build the initial Document Hashing Request. There are four variants depending upon whether the `pdfDocument` is specified as a `file path` or `byte[]` and whether the `userCertificate` is passed as an `X509Certificate` or as a `byte[]`.

```
var hashRequest = new DocumentHashingRequest(string clientID, pdfDocument,
userCertificate);
```

4.7.1 Document Hashing Request Methods

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send, SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

In addition, the following methods are specific to the Document Hashing Request class:

Document Hashing Request Method	Purpose
`OverrideProfileAttribute (string attributeID, string/byte[])`	Allows specific attributes to be passed to ADSS Server as part of the hashing request - see example below where various signing attributes are supplied.
`SetProfileId(string)`	Specifies the Signing Service Profile ID to be used for servicing the Document Hashing request.

4.7.2 Sending the Document Hashing Request

Once the Document Hashing Request message has been built, it can be sent to ADSS Server using the following method call:

```
var hashResponse = (DocumentHashingResponse)hashRequest.Send(string URL);
```

The `URL` is that of the Document Hashing Request Service e.g. http://machine-name:8777/adss/signing/dhi

4.7.3 Example of a Document Hashing Request

```
// Construct document hashing request
var hashRequest = new DocumentHashingRequest(clientID, pdfFile, userCert);
hashRequest.SetProfileId(profileID);
hashRequest.OverrideProfileAttribute(DocumentHashingRequest.SIGNING_REASON, signingReason);
hashRequest.OverrideProfileAttribute(DocumentHashingRequest.CONTACT_INFO, contactInfo);
hashRequest.OverrideProfileAttribute(DocumentHashingRequest.SIGNING_FIELD, signingField);
hashRequest.SetRequestMode(requestMode);

// Send request to ADSS server
var hashResponse = (DocumentHashingResponse)hashRequest.Send(serviceAddress);
```

4.7.4 Document Hashing Response Methods

Document Hashing Response methods are mainly inherited from the Message and Response classes and are described in section 3 and in the JavaDoc/Sandcastle class documentation.

In addition the following methods are specific to the Document Hashing Response class:

Document Hashing Response Method	Purpose
`GetDocumentHash() returns byte[]`	Returns the hash of the PDF document calculated by the Signing Service. Required for the subsequent signature assembly request.
`GetDocumentId() returns ArrayList`	Returns a document Id. Required for the subsequent signature assembly request.
`GetProfileId() returns string`	Returns the Signing Profile ID used by the ADSS Signing Service to process the request.

4.8 Signature Assembly Request and Response classes

The Signature Assembly classes are used together with the Document Hashing classes described in section 0. ADSS Server is first requested to create the hash for a PDF document, then the client signs the hash and finally ADSS Server assembles the document signature.

The following constructor is used to build the initial Signature Assembly Request.

`Signature` is the signature generated at the client after ADSS Server had returned the document hash. `documentId` is the ID of the hashed document.

```
var assemblyRequest = new SignatureAssemblyRequest(clientID, signature,
documentId);
```

4.8.1 Signature Assembly Request Methods

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send, SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

In addition, the following method is specific to the Signature Assembly Request class:

Signature Assembly Request Method	Purpose
`SetProfileId(string)`	Specifies the Signing Service Profile ID to be used for servicing the request.

4.8.2 Sending the Signature Assembly Request

Once the Signature Assembly Request message has been built, it can be sent to ADSS Server using the following method call:

```
var assemblyResponse =
(SignatureAssemblyResponse)assemblyRequest.Send(string URL);
```

The `URL` is that of the Signing Service e.g. http://machine-name:8777/adss/signing/sai

4.8.3 Example of a Signature Assembly Request

```
// Construct signature assembly request
var assemblyRequest = new SignatureAssemblyRequest(clientID, signature, hashResponse.GetDocumentId());
assemblyRequest.SetProfileId(profileID);
assemblyRequest.SetRequestMode(requestMode);

// Send request to the ADSS server
var assemblyResponse = (SignatureAssemblyResponse)assemblyRequest.Send(serviceAddress2);
```

4.8.4 Signature Assembly Response Status Processing

The Signature Assembly Response methods are mainly inherited from the Message and Response classes and are described in section 3 and in the JavaDoc/Sandcastle class documentation.

In addition, the following methods are specific to the Signature Assembly Response Class:

Signature Assembly Response Method	Purpose
`GetDocument() returns string`	Returns the signed PDF document.
`GetProfileId() returns string`	Returns the Signing Profile ID used by the ADSS Signing Service to process the request.
`PublishDocument(string /Stream)`	Publishes the signed document to the specified path or stream.

4.9 Office Signing Request and Response Classes

The Office Signing Request Class is used to sign signature lines in office document.

The following constructor is used to build the initial Office Signing Request message. There are different variants depending upon the source of the document to be signed i.e. whether the `document` is specified as a `file path, Stream or byte[].`

```
OfficeSigningRequest obj_officeSigningRequest = new
OfficeSigningRequest(String clientID,byte[] document, String mimeType);
```

4.9.1 Office Signing Request methods

The following methods are inherited from the Sign Request class and described in section 0:

`setCertificateAlias,setCertificatePassword,setProfileId,setSignerCertificate, setSignerCertificateChain,setSignatureForm`

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc class documentation:

`reset, setProxy, setProxy, setRequestId, setRequestRetries, setRespondAddress, setSigningCredentials, setSigningCredentials, setSigningMode, setSoapVersion, setSslClientCredentials,`

```
setSslClientCredentials, setSslTrustStore, setTimeout,
setVerifyResponse,toString, writeTo, writeTo.
```

The following additional methods are specific to the Office Signing Request:

Office Signing Request Method	Purpose
`SetSignHash(boolean a_bComputeHash)`	If this flag is set to false in ADSS Client SDK then ADSS Signing Service will do the hashing and encryption. If this flag is set to true then ADSS Client SDK will do hashing and ADSS Signing Service will do encryption. By default it is set to false.
`SetDigestAlgorithm(String a_strDigestAlgorithm)`	Specifies the hash algorithm to be used in the case of local document hashing.
`SetLocalHash(boolean a_bLocalHash)`	It sets the flag for local hashing. This method is used if client need to do the local hashing on the Office document instead of sending the whole document to Signing Service. This method reduces network band because only the hash travels and also Office document confidentiality will be achieved if client does not want to send whole document to the server.
`SetHandSignature(byte[] a_byteHandSignature)`	Specifies a hand signature image file.
`SetSetupId(String a_strSetupId)`	Specifies the setup Id. It is used to identify the signature line in Office document. Possible values are UUID i.e {C777B235-1C61-4304-A7EC-0581612CD745} or email address. If same email address is associated with multiple signature lines then all signature line will be signed in case of server side signing.
`IsDocumentSigned() returns bool`	Checks if the provided MS Office document is already signed or not. This method is helpful in deciding whether to send an already signed document for signing to ADSS Server or skip it.

4.9.2 Sending the Office Signing Request

Once the office signing request message has been built using the above methods, it can then be sent to ADSS Server using the following method call:

```
obj_officeSigningResponse = (OfficeSigningResponse) officeSigningRequest.send(String URL);
```

The `URL` is that of the Signing Service e.g. `http://machine-name:8777/adss/signing/dss` or `http://machine-name:8777/adss/signing/hdsi` when using HTTP mode.

4.9.3 Example of building and sending an Office Signing Request

```
// Constructing request for word document signing
OfficeSigningRequest obj_officeSigningRequest = new
OfficeSigningRequest(str_ClientId, arr_bdocument, str_mimeType);
obj_officeSigningRequest.setProfileId(str_ProfileId);
obj_officeSigningRequest.setSetupId(str_setupId);
obj_officeSigningRequest.setRequestMode(str_requestMode);


// Sending the above constructed request to the ADSS server
```

```
OfficeSigningResponse obj_officeSigningResponse = (OfficeSigningResponse)
officeSigningRequest.send(str_serviceAddress);
```

4.9.4 Office Signing Response Methods

All Office Signing Response methods are inherited from the Message, Response classes described in sections 3 and 4.4.3, and in the JavaDoc class documentation.

4.10 Signing Status Request and Response Classes

The Signing Status Request Class is used to get status of signing request.

The following constructor is used to build the initial Signing Status Request message.

```
StatusRequest signingStatusRequest = new StatusRequest (String clientID,
String transactionID);
```

4.10.1 Sending the Signing Status Request

Once the signing status request message has been built using the above methods, it can then be sent to ADSS Server using the following method call:

```
obj_signingStatusResponse = (StatusResponse)
signingStatusRequest.send(String URL);
```

The `URL` is that of the Signing Service e.g. http://machine-name:8777/adss/signing/hdsi

4.10.2 Example of building and sending an Signing Status Request

```
// Constructing request for signing status request
StatusRequest signingStatusRequest = new StatusRequest(str_ClientId,
str_transactionID);

// Sending the above constructed request to the ADSS server
StatusResponse obj_signingStatusResponse = (StatusResponse)
signingStatusRequest.send(str_serviceAddress);
```

4.10.3 Signing Status Response Methods

All Signing Status Response methods are inherited from the Message, Response classes described in sections 3 and 4.4.3, and in the JavaDoc class documentation.

Also they are described in section 3 and in the JavaDoc/Sandcastle class documentation.

In addition the following methods are specific to the Status Response class:

Document Hashing Response Method	Purpose
`getStatus()` `returns string`	Possible statuses are: • SUCCESS • FAILED • PENDING • CANCELLED
`getTransactionID ()` `returns string`	Returns the transaction id that used to get status of the request.
`getComputedSignature ()` `returns byte[]`	Return the computed signature.

4.11 Certificate Download Request and Response Classes

The Certificate Download Request Class is used to get the certificate and its chain of the certificate alias which will be mentioned in the request.

The following constructor is used to build the initial Certificate Request message.

```
CertificateRequest obj_certRequest = new CertificateRequest(String
CertificateAlias, String MimeType);
```

4.11.1 Certificate Request methods

The following methods are inherited from the Sign Request class and described in section 0:

`setCertificateAlias,setCertificatePassword,setProfileId,setSignerCertificate, setSignerCertificateChain,setSignatureForm`

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc class documentation:

`reset, setProxy, setProxy, setRequestId, setRequestRetries, setRespondAddress, setSigningCredentials, setSigningCredentials, setSigningMode, setSoapVersion, setSslClientCredentials, setSslClientCredentials, setSslTrustStore, setTimeout, setVerifyResponse,toString, writeTo, writeTo.`

The following additional methods are specific to the Certificate Request:

Certificate Request Method	Purpose
`setMimeType()` `returns string`	Mime type attribute is to identify the type of Certificate that client wants (user certificate or service certificate) to export. It is a mandatory parameter in the certificate download request.
`setClientId()` `returns string`	Client ID to identify that which client is processing the request.
`setAlias()` `returns string`	Specifies the certificate alias which will be used to export that particular certificate and its chain.
`setProfileId()` `return string`	Specifies the Signing Service Profile ID to be used for downloading the specific certificate and its chain.

4.11.2 Sending the Certificate Download Request

Once the certificate download request message has been built using the above methods, it can then be sent to ADSS Server using the following method call:

```
CertificateResponse obj_certificationResponse = (CertificateResponse)
obj_certRequest.Send(string str_serviceAddress);
```

The str_serviceAddress is that of the Signing Service e.g. `http://machine-name:8777/adss/signing/hcert`

4.11.3 Example of building and sending a Certificate Download Request

```
// Constructing request for certificate download request
CertificateRequest obj_certRequest = new
CertificateRequest(CertificateAlias,MimeType);

// Sending the above constructed request to the ADSS server
CertificateResponse obj_certificationResponse = (CertificateResponse)
obj_certRequest.Send(str_serviceAddress);
```

4.11.4 Certificate Response Methods

All Certificate Response methods are inherited from the Message, Response classes described in sections 3 and 4.4.3, and in the JavaDoc class documentation.
They are also described in section 3 and in the JavaDoc/Sandcastle class documentation.

In addition, the following methods are specific to the Certificate Response class:

Certificate Response Method	Purpose
`getCertificate ()` `returns byte[]`	Return the certificate and its chain.

4.12 Signing Service Sample Code

Java and .Net sample code is provided as part of the ADSS Client SDK and can be used to make Signing Service requests and to process the Signing Service responses.

The Java API provides the required classes under the package:

`com.ascertia.adss.client.api.signing`

The .Net API provides the required classes under the namespace:

`Com.Ascertia.ADSS.Client.API.Signing`

4.12.1 Java API Sample Code

The following sample programs demonstrate how the Java API can be used to send a Signing Service request and to process the response:

```
samples/src/com/ascertia/adss/client/samples/signing/SignPDF.java
samples/src/com/ascertia/adss/client/samples/signing/SignPdfWithTextWater
mark.java
samples/src/com/ascertia/adss/client/samples/signing/SignFile.java
samples/src/com/ascertia/adss/client/samples/signing/SignXML.java
samples/src/com/ascertia/adss/client/samples/signing/SignXmlWithLocalHash
.java
samples/src/com/ascertia/adss/client/samples/signing/SignPdfHTTP.java
samples/src/com/ascertia/adss/client/samples/signing/SignPdfUsingPreferen
ces.java
samples/src/com/ascertia/adss/client/samples/signing/SignPdfWithLocalHash
.java
samples/src/com/ascertia/adss/client/samples/signing/AuthoriseSignPDF.jav
a
samples/src/com/ascertia/adss/client/samples/signing/CreateEmptySigFields
.java
samples/src/com/ascertia/adss/client/samples/signing/HashAndAssemblyManag
er.java
samples/src/com/ascertia/adss/client/samples/signing/SignMsOfficeDocument
.java
samples/src/com/ascertia/adss/client/samples/signing/SignMsOfficeDocument
HTTP.java
```

```
samples/src/com/ascertia/adss/client/samples/signing/SignMsOfficeDocument
WithLocalHash.java
samples/src/com/ascertia/adss/client/samples/signing/DownloadCertificate.
java
```

Precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/SignPDF.bat
samples/bin/SignPdfWithTextWatermark.bat
samples/bin/SignFile.bat
samples/bin/SignXML.bat
samples/bin/SignXmlWithLocalHash.bat
samples/bin/SignPdfHTTP.bat
samples/bin/SignPdfUsingPreferences.bat
samples/bin/SignPdfWithLocalHash.bat
samples/bin/AuthoriseSignPDF.bat
samples/bin/CreateEmptySigFields.bat
samples/bin/HashAndAssemblyManager.bat
samples/bin/SignMsOfficeDocument.bat
samples/bin/SignMsOfficeDocumentHTTP.bat
samples/bin/SignMsOfficeDocumentWithLocalHash.bat
samples/bin/ DownloadCertificate.bat
```

4.12.2 .Net API Sample Code

The following sample programs demonstrate how the .Net API can be used to send a Signing Service request and to process the response:

```
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignPDF.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignPdfWithTextWater
mark.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignFile.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignXML.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignXmlWithLocalHash
.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignPdfHTTP.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignPdfUsingPreferen
ces.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignPdfWithLocalHash
.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/AuthoriseSignPDF.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/CreateEmptySigFields
.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/HashAndAssemblyManag
er.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignMsOfficeDocument
.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignMsOfficeDocument
HTTP.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/SignMsOfficeDocument
WithLocalHash.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Signing/
DownloadCertificate.cs
```

Precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/SignPDF.bat
samples/bin/SignPdfWithTextWatermark.bat
samples/bin/SignFile.bat
samples/bin/SignXML.bat
samples/bin/SignXmlWithLocalHash.bat
```

```
samples/bin/SignPdfHTTP.bat
samples/bin/SignPdfUsingPreferences.bat
samples/bin/SignPdfWithLocalHash.bat
samples/bin/AuthoriseSignPDF.bat
samples/bin/CreateEmptySigFields.bat
samples/bin/HashAndAssemblyManager.bat
samples/bin/SignMsOfficeDocument.bat
samples/bin/SignMsOfficeDocumentHTTP.bat
samples/bin/SignMsOfficeDocumentWithLocalHash.bat
samples/bin/DownloadCertificate.bat
```

4.12.3 Document Hashing and Assembly API sample

An API sample HashAndAssemblyManager is present in the Client-SDK to help in understanding the usage of Document Hashing and Assembly. The sample is located at the following path in the Client-SDK **"**{Client-SDK}/API/samples/bin/HashAndAssemblyManager.bat**"**. A configuration file located at **"**{Client-SDK}/API/samples/conf/HashAndAssemblyManager.properties**"** is used by this API sample which can be configured accordingly to allow support for both ISO Standard PDF and PAdES Baseline/Extended signatures.

The following properties are present in the HashAndAssemblyManager.properties file.

Property (M/O)	Description
ADSS_SERVER_DOCUMENT_HASHING_URL (M)	For XML interface use **http://localhost:8777/adss/signing/dhi** and for HTTP use **http://localhost:8777/adss/signing/hdhi**
ADSS_SERVER_DOCUMENT_ASSEMBLY_URL (M)	For XML interface use **http://localhost:8777/adss/signing/sai** and for HTTP use **http://localhost:8777/adss/signing/hsai**
REQUEST_MODE (M)	The request mode using which the request is sent. Possible value of Request Mode are **XML**, **DSS** and **HTTP**; however the **DSS** mode is not supported.
CLIENT_ID (M)	The clientID of a client registered in the ADSS server.
PROFILE_ID (O)	The ProfileID to be used by ADSS Server. The ProfileID must be registered within ADSS Server. If it is not, the default ProfileID will be used.
ADSS_SERVER_VERIFICATION_SERVER_URL (O)	The URL of the verification server to fetch revocation info of the signing certificate to be embedded in the signed attributes while signing. Required when the **SIGNATURE_TYPE** is selected as **PDF_RI** e.g **http://localhost:8777**
UNSIGNED_INPUT_FILE_PATH (M)	Input file on which hash is to be computed.
SIGNED_OUTPUT_FILE_PATH (M)	Output file with the embedded signature.
PFX_FILE_PATH (M)	Pfx file path to sign the hash.
PFX_FILE_PASSWORD (M)	Password of the Pfx file.
SIGNATURE_TYPE (M)	The type of signature to be computed. Possible values of **SIGNATURE_TYPE** are **PDF** (for ISO Standard PDF), **PDF_RI** (for ISO Standard Pdf with embedded revocation information), **PADES** (for PAdES ETSI Extended) and **PADES_B** (for PAdES ETSI Baseline).
HASH_ALGORITHM (M)	Hash algorithm to be used while computing the signature on the hash, it must match with the hash algorithm configured in the profile. It is recommended to keep the value of the Hash algorithm as **SHA1** when the **SIGNATURE_TYPE** is selected as **PDF** and the **SIGNATURE_MODE** is selected as **ENVELOPING**.
SIGNATURE_MODE (M)	The relationship between the signature and the document. Possible values of Signature mode are **ENVELOPING** (Only allowed when the **SIGNATURE_TYPE** is **PDF**) and

	DETACHED, it must be same as the one configured in the signing profile
SIGNATURE_PADDING_SCHEME (M)	Padding scheme to compute the signature. Possible values are **PKCS1** and **PSS**. it must match with the Padding settings of signing profile.
PDF_PROTECTED_PASSWORD (O)	Permission passphrase that will be set on the PDF document to change the permissions of the document.
SIGNING_FIELD (O)	Name of an existing blank signature field to be signed. Only one value is supported.
COMPANY_LOGO (O)	A file path of an image to be placed as a company logo (applicable only if the document type supports visible signatures).
HAND_SIGNATURE (O)	A file path of an image to be placed as a hand signature (applicable only if the document type supports visible signatures).
CONTACT_INFO (O)	Contact information for the signer e.g. a telephone number, email address, street address.
SIGNING_AREA (O)	Page area on which the signature should be placed (applicable only if visible signatures are being used). Possible values are **top_left**, **top_right**, **center**, **bottom_left** and **bottom_right**.
SIGNED_BY (O)	Signer name who signed the document e.g. “John Doe”
SIGNING_LOCATION (O)	Location where the document is being signed e.g. “London, UK”.
SIGNING_REASON (O)	Signing reason e.g. “I approve this document”.
SIGNING_PAGE (O)	Page number on which the signature should be placed (applicable only if the document allows visible signatures) e.g. 10. Only one value is supported.
SIGNATURE_APPEARANCE (O)	Signature appearance to be used while computing the hash e.g. default_sig_appearance. The signature appearance must already exist in the ADSS server. Only one value is supported.
VISIBLE_SIGNATURE_X1_COORDINATE (O)	X1 coordinate for visible signature position where the signature will be placed. Only one value is supported.
VISIBLE_SIGNATURE_Y1_COORDINATE (O)	Y1 coordinate for visible signature position where the signature will be placed. Only one value is supported.
VISIBLE_SIGNATURE_X2_COORDINATE (O)	X2 coordinate for visible signature position where the signature will be placed. Only one value is supported. **Note:** For iText version 7 or later, the X2 coordinate is interpreted as the width value.
VISIBLE_SIGNATURE_Y2_COORDINATE (O)	Y2 coordinate for visible signature position where the signature will be placed. Only one value is supported. **Note:** For iText version 7 or later, the Y2 coordinate is interpreted as the height value.

4.13 ADSS Signing Service Supported Algorithms

The following is a list of signing/hashing algorithms and key lengths that ADSS Signing Service supports:

ADSS Service	Signing Algorithms	Hashing Algorithms	Signing Key Algorithm Lengths
Signing	SHA1WithRSAEncryption SHA224WithRSAEncryption SHA256WithRSAEncryption	SHA-1 SHA-224 SHA-256	RSA: 1024, 2048, 3072, 4096

	SHA384WithRSAEncryption SHA512WithRSAEncryption SHA3-224WithRSAEncryption SHA3-256WithRSAEncryption SHA3-384WithRSAEncryption SHA3-512WithRSAEncryption SHA3-224WithRSAandMGF1 SHA3-256WithRSAandMGF1 SHA3-384WithRSAandMGF1 SHA3-512WithRSAandMGF1 RipeMD128WithRSAEncryption RipeMD160WithRSAEncryption SHA1withECDSA SHA224withECDSA SHA256withECDSA SHA384withECDSA SHA512withECDSA SHA3-224withECDSA SHA3-256withECDSA SHA3-384withECDSA SHA3-512withECDSA	SHA-384 SHA-512 SHA3-224 SHA3-256 SHA3-384 SHA3-512 RipeMD128 RipeMD160 GOST-3411-94	ECDSA: 192,224,256,384,521 GOST-3410-2001 512
Hashing, Assembly	-	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 SHA3-224 SHA3-256 SHA3-384 SHA3-512 RipeMD128 RipeMD160	-

4.14 Error Codes

ADSS Signing Server returns the following statuses in case of any failure:

Error Code	Error Message
41001	ADSS Signing service not enabled in license.
41002	ADSS Signing service license has expired.
41003	ADSS Signing service is stopped.
41004	An internal error occurred while processing the request.

41005	Failed to create signature.
41006	Failed to create Visible signature.
41007	Failed to create invisible signature.
41008	Failed to create PKCS#7 signature.
41009	Failed to embed timestamp token.
41010	Failed to embed revocation information.
41011	Failed to embed archive timestamp token.
41012	PDF signing not enabled.
41013	Time stamped signature creation not enabled in license.
41014	Long term signature creation is not enabled in license.
41015	SigG compliant signature creation not enabled in license.
41016	Failed to embed revocation information.
41017	Failed to create XAdES-X type 2 signatures.
41018	User certificate format is invalid.
41019	No document found in the request.
41020	No Originator ID found in the request.
41021	User certificate chain not present in the request.
41022	No signature found on the request.
41023	Request structure is invalid.
41024	Signature verification failed.
41025	Referenced private key does not belong to the client.
41026	Referenced certificate chain not found.
41027	Signer certificate has expired.
41028	Signer certificate is a CA certificate.
41029	Request is not signed.
41030	Document Id is not found in the request.
41031	Signing profile is not appropriate for this file type.
41032	Signed PDF cannot subsequently be certify signed.
41033	PDF is already certify signed.
41034	Failed to authenticate signing request.
41035	Authorised signing is not enabled in license.
41036	Failed to validate authoriser’s signature.
41037	Authorisers signature not found in request.
41038	Signer certificate status is revoked or unknown.
41039	Default profile not configured and neither found in request.
41040	Signing certificate alias is not one of the allowed set of certificate aliases.

41041	Default signing certificate alias not found in request or profile.
41042	An internal error occurred while processing the request.
41043	Failed to match data hash with signed hash.
41044	Failed to assemble signature within the document.
41045	Failed to create empty signature field.
41046	Failed to create file signature.
41047	Failed to parse PDF document.
41048	Failed to create signing response.
41049	Failed to compute hash.
41050	Failed to create CAdES-T signature.
41051	Private key referenced is not found.
41052	Document hash was not provided in the request.
41053	Failed to create XAdES-T signature.
41054	Failed to create CMS signature.
41055	Failed to create CAdES-BES signature.
41056	Failed to create S/MIME signature.
41057	XAdES detached signatures are not supported.
41058	Failed to create XADES-BES signature.
41059	Failed to create CAdES-X-L signature.
41060	Failed to create XAdES-X-L signature.
41061	Failed to create CAdES-A signature.
41062	Failed to create XAdES-A signature.
41063	Certify signed PDF with no changes allowed cannot be signed.
41064	Signing field information is not available.
41065	Problem in creating/signing empty signature field.
41066	SigG signature configurations are not available.
41067	Failed to create SigG signature.
41068	Signer certificate does not contain required extensions.
41069	Processing of requested signature type is not enabled in license.
41070	Signature grace period is not yet elapsed.
41071	Failed to create PAdES-BES signature.
41072	Failed to create XAdES-X type 1 signature.
41073	Input file is either unsigned or contains an archive time stamped signature.
41074	PDF document already contains PAdES-LTV Document Timestamp signature.
41075	Signing service not allowed.

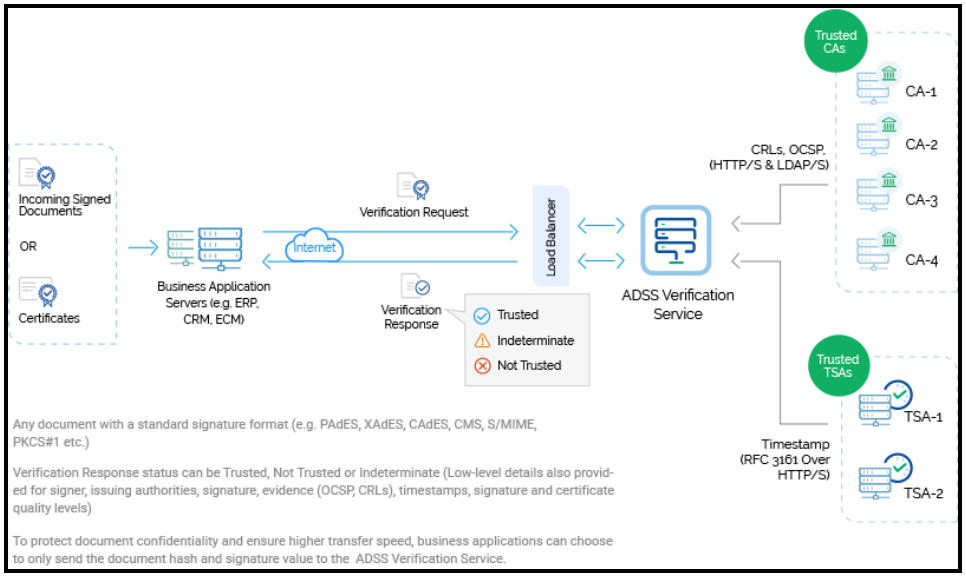
41076	Signature dictionary size is smaller than required.
41077	No key alias available from request and profile.
41078	Invalid input document format.
41079	Embedded data in signature time stamp is invalid.
41080	Embedded data in time stamp is invalid.
41081	Embedded data in signature is invalid.
41082	Signing time not present in signature.
41083	Signing service not enabled in system.
41084	Signing field not found in the PDF document.
41085	Signing certificate cannot be used for document or notary or email signing.
41086	Signing profile is inactive.
41087	Signing profile does not exist or marked inactive.
41088	Input document is encrypted.
41089	Signed document cannot be encrypted.
41090	Authorised signature not supported on http interface.
41091	Invalid password.
41092	Password not provided in the request.
41093	Key cannot be used for document signing.
41094	Only PDF documents can be time stamped.
41095	Revocation information unavailable for existing document time stamp.
41096	Unable to create document time stamp.
41097	User authentication failed against the directory.
41098	PDF document cannot be time stamped, document has user signature.
41155	Certificate or certificate chain is not available.
41158	Unable to get RAS profile information
41161	Invalid credential authorisation settings or request parameters are missing
41162	eSeal is not enabled in SAM profile
41163	Unable to authenticate client over RAS OAuth2/Token interface
41164	Explicit with OTP credential authorisation method is not supported
41165	Request ID is not found in signing request required for OAuth2 credential authorisation
41166	Failed to sign hash
41167	Response URI is not configured in signing profile
41168	Eseal is not enabled in signing profile

5 ADSS Verification Service

The ADSS Server Verification Service provides the following types of verification services:

- OASIS DSS Compliant signature verification (PDF, XML and PKCS7 signatures)
- OASIS DSS Compliant verification of advanced signature formats (XAdES, CAdES and PAdES)
- OASIS DSS certificate validation

Business Client Applications send requests to ADSS Server and receive responses back. The majority of the verification parameters are already set up in profiles at the server but some may be overridden if the profile permits this (e.g. certificate or key quality). The Business Application just needs to provide a list of the items it requires back in the response.



All the Trust Services shown above are provided either by ADSS Server or they can be external.

The protocol used for the OASIS compliant services is based on the OASIS Digital Signature Service (DSS) Core Protocols, Elements and Bindings specification (oasis-dss-core-spec-v1.0-os). Messages are wrapped in a SOAP message and sent using HTTP.

In addition to this core protocol, the Verification Service also supports the OASIS DSS-X Verification Report protocol (oasis-dsss-1.0-profiles-vr-cd01).

5.1 Digital Signature Standards

Digital signatures verified by ADSS Server are open standards compliant and can include timestamps and revocation information. The following signature types are supported:

Supported Signature Types (ascertia.com)

5.2 Setting up Verification Service Profiles

The ADSS Verification Service requires that Verification Profiles are defined at ADSS Server. These profiles identify the type of signature that can be verified using that profile (e.g. PDF certifying signature with embedded timestamp and revocation information) and any other settings that may be required (e.g. specific key usages or extended key usages, certificate quality levels, certificate path settings etc.).

Refer to the following online admin guide for an explanation of Verification Profile settings:

Step 4 - Configuring Verification Profile (ascertia.com)

5.3 The Verification Service API

In order to simplify the use of the OASIS DSS and Ascertia proprietary HTTP protocols, a Verification Service API is provided as part of the ADSS Client SDK.

5.3.1 Verification Request

The API consists of the following classes for building signature verification requests:

- Signature Verification Request class – the main signature verification request class
- Signature Info class – a class to hold a signature to be verified by the service.

The API consists of these classes for building certificate validation requests:

- Certificate Validation Request class – the main certificate validation request class
- Certificate Info class – a class to hold a certificate to be validated by the service.

Both the Signature Verification and Certification Validation Request classes make use of a common base class which provides some of the required methods:

- Verification Request class – see section 5.4.

5.3.2 Verification Response

The API has these classes to process the verification response:

- Verification Response class – provides access to the verification response object
- Verify Info class – provides response details for each signature verified
- Various Verification Report classes.

5.4 Verification Request Classes

5.4.1 Verification Request Methods

The Signature Verification Request and Certificate Validation Request classes both inherit a number of methods provided by the Verification Request class. These common methods are described below.

In addition, the following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

```
ToString, WriteTo, Send (overridden), SetProxy, SetRequestID,
SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion,
SetSSLClientCredentials, SetTimeout, SetVerifyResponse.
```

5.4.2 Specifying Required Key Usages

Verification Request method	Purpose
`AddKeyUsage(string keyUsage)`	Adds key usage(s) into the request. The Verification Service checks that these are present in the signing certificate. The method can be called multiple times to add multiple key usages into the request. The following key usages constants are defined: - `KEY_USAGE_DIGITAL_SIGNATURE` - `KEY_USAGE_NON_REPUDIATION` - `KEY_USAGE_KEY_ENCIPHERMENT` - `KEY_USAGE_DATA_ENCIPHERMENT` - `KEY_USAGE_KEY_AGREEMENT`

	- KEY_USAGE_KEY_CERT_SIGN - KEY_USAGE_CRL_SIGN - KEY_USAGE_ENCIPHER_ONLY - KEY_USAGE_DECIPHER_ONLY

5.4.3 Adding 'Respond With' Items

Verification Request method	Purpose
`AddRespondWithItem(string respondWith)`	Adds 'respond with' item(s) into the request. The respond with items will then be returned in the response. The method can be called multiple times to add multiple 'respond with' items. The following are the defined 'respond with' constants: - RESPOND_WITH_KEY_VALUE - RESPOND_WITH_HASH_ALGORITHM - RESPOND_WITH_SIGNATURE_ALGORITHM - RESPOND_WITH_X509_CERTIFICATE_CHAIN - RESPOND_WITH_X509_CRL - RESPOND_WITH_SKI - RESPOND_WITH_OCSP - RESPOND_WITH_TIMESTAMP - RESPOND_WITH_SIGN_HASH - RESPOND_WITH_CONTENT_HASH - RESPOND_WITH_CONTENT - RESPOND_WITH_KEY_USAGE - RESPOND_WITH_EXTENDED_KEY_USAGE - RESPOND_WITH_BASIC_CONSTRAINTS - RESPOND_WITH_VALID_FROM - RESPOND_WITH_VALID_TO - RESPOND_WITH_CERTIFICATE_SERIAL_NUMBER - RESPOND_WITH_ISSUER_NAME - RESPOND_WITH_SUBJECT - RESPOND_WITH_CRL_URL - RESPOND_WITH_CRL_NUMBER - RESPOND_WITH_TSA_TIME **(supported in HTTP interface only)** - RESPOND_WITH_TSA_NAME **(supported in HTTP interface only)** - RESPOND_WITH_TSA_HASH_ALGORITHM **(supported in HTTP interface only)** - RESPOND_WITH_SIGNATURE_TYPE **(supported in HTTP interface only)** - RESPOND_WITH_LEI_INFO **(supported in HTTP interface only)**

5.4.4 Overriding Minimum PEPPOL Quality Trust Settings

The ADSS Verification Service supports the PEPPOL standard certificate and algorithm quality trust levels.

Verification Request method	**Purpose**
`SetCertificateQualityLevel (string certQualityLevel)`	Sets the minimum acceptable PEPPOL certificate quality level.
`SetHashAlgoQualityLevel (string hashAlgoQualityLevel)`	Sets the minimum acceptable PEPPOL hash algorithm quality level.
`SetIndependentAssuranceLevel (string independentAssuranceLevel)`	Sets the minimum acceptable PEPPOL independent assurance level.
`SetPublicKeyAlgoQualityLevel (string publicKeyAlgoQualityLevel)`	Sets the minimum acceptable PEPPOL public key algorithm quality level.

5.4.5 OASIS DSS-X Verification Reports

The ADSS Verification Service supports the generation of OASIS DSS-X reports. These are requested using the following method:

Verification Request method	**Purpose**
`SetReturnVerificationReport (bool includeVerifier, bool includeCertificateValues, bool includeRevocationValues, bool expandBinaryValues, string reportDetailLevel)`	Specifies the OASIS DSS-X verification report level required in the response. The meaning of these values is: includeVerifier: If true then the identity of the verifier will be included in the report. This option defaults to true. includeCertificateValues: If true then the certificate values, used to verify the signature (in binary form or as equivalent XML structure) are included in the report. This option defaults to false. includeRevocationValues: If true then the revocation values used (e.g. OCSP responses, CRLs and timestamps) will be included (in binary form or as equivalent XML structure) in the report. This option defaults to false. expandBinaryValues: If true then the Verification Service (as it fulfils the OASIS conformance level "Convenient") will include the expanded content of certificates and revocation information into the verification report. The binary ASN.1-coded binary values will thereby be included as equivalent XML structures. This option defaults to false. reportDetailLevel: This specifies the OASIS Verification Report detail level and can be one of the following values: - `REPORT_LEVEL_ALL_DETAILS` - `REPORT_LEVEL_NO_DETAILS` - `REPORET_LEVEL_NO_PATH_DETAILS`

5.4.6 Other Verification Request Methods

The remaining methods specific to the Verification Service class are as follows:

Verification Request method	**Purpose**
`SetGatewayCompliance(bool gatewayCompliance)`	If set to true this causes the verification request to consist of document hashes and signatures rather than the complete documents. This ensures document confidentiality for a business client.
`SetRequestMode (Int32)`	Specifies the request mode (one of): - `VerificationRequest.HTTP` (default mode)

	- VerificationRequest.DSS In high performance HTTP mode, the document is placed in the HTTP body while other information is placed in the HTTP header. In OASIS DSS mode, a message is created by following the OASIS DSS specification. It is then wrapped in a SOAP message and sent using the HTTP protocol. **Note**: The mode parameter is important as affects how the response status is handled. **Note:** Detached signature verification and respond with items are not supported over the HTTP interface.
SetProfileId (string)	Specifies the Verification Profile identifier.
SetHistoricalValidation (DateTime historicalValidation, int timeZone)	Specifies a date and time that should be used as the basis for performing historical signature verification. (The time zone value is optional).
SetPeppolCompliance(bool peppolCompliance)	If set to true then the verification request will be constructed in compliance with the PEPPOL standard.
SetValidationAtCurrentTime (bool currentTime)	If set to true then current time should be used for performing signature validation.

5.5 Signature Verification Requests

5.5.1 Signature Verification Request Constructor

The Signature Verification Request Class is used when requesting ADSS Server to verify signatures. It can be used with any document and signature type e.g. (PKCS7/CMS, XML or PDF).

The following constructor is used to build the initial Signature Verification Request message and specifies the clientID of the calling business application plus a transaction id reference.

```
var sigverifyRequest = new SignatureVerificationRequest(clientID, transID);
```

5.5.2 Signature Verification Request Methods

The following methods are inherited from the Verification Request class and described in section 5.4:

AddKeyUsage, AddRespondWithItem, Send, SetCertificateQualityLevel, SetGatewayCompiance, SetHashAlgoQualityLevel, SetHistoricalValidation, SetIndependentAssuranceLevel, SetPeppolCompliance, SetProfileID, SetPublicKeyAlgoQualityLevel, SetRequestMode, SetReturnverificationReport.

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

ToString, WriteTo, SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.

The following is a list of the remaining methods that are specific to the Signature Verification Request Class:

Signature Verification Request Method	Purpose
AddSignatureInfo(signatureInfo)	Adds a signature into the request. This method may be called multiple times if more than one signature is to be verified.

`SetContent(byte[] content)`	If the signature is a detached PKCS#7/CMS signature then the signed content must be provided using this method.
`SetReturnSignerIdentity()`	Asks the Verification Service to return the signer's identity.
`SetReturnSigningTimeInfo()`	Asks the Verification Service to return the signature creation time. If there are multiple signatures then it returns the signing time of the first signature validated.
`SetReturnTimeStampSignature()`	Asks the Verification Service to upgrade the signature to a time stamped signature.
`SetReturnUpdatedSignature(string returnUpdatedSignature)`	Asks the verification Service to return the updated signature type (e.g. CAdES-T).
`SetReturnVerificationTimeInfo()`	Asks the Verification Service to return the time at which the signature(s) were validated
`sendOfficeSignaturesOnly()`	If this method is called, only the signatures in a Microsoft Office document would be sent to ADSS Verification Service instead of full document.
`SetSignatureTagPosition(String a_strSignatureTagPosition)`	If there are multiple signatures and client wants to verify only particular signature/signatures then this method is used to set the position of signatures to be verified in XAdES signatures.It is comma separated string and client can define more then one signature's position (e.g 1,2)
`SetSignatureXpathToVerify(String a_strSignatureXpathToVerify)`	If there are multiple signatures and client wants to verify only particular signature/signatures then this method is used to set the signatures to be verified in XAdES signatures.It is "|" separated string and client can define more then one signature's xpaths (e.g //ds:Signature[@Id='Signature_11299']|//ds:Signature[@Id='Signature_11549'])

5.5.3 Signature Info Class Constructor

The Signature Info class is used to indicate a specific signature to be verified and once prepared it is added into the Signature Verification request.

The following constructor is used to build the Signature Info class instance and specifies a `signatureID`, the `signatureToValidate` (provided as a document file path or as a `byte[]`) and the `signedDocumentType`.

```
var signatureInfo = new SignatureInfo(signatureID, signatureToValidate,
signedDocumentType);
```

The `signedDocumentType` is one of the following:

```
SignatureInfo.SIGNED_DOCUMENT_TYPE_CMS
SignatureInfo.SIGNED_DOCUMENT_TYPE_MIME
SignatureInfo.SIGNED_DOCUMENT_TYPE_OTHER
SignatureInfo.SIGNED_DOCUMENT_TYPE_PDF
SignatureInfo.SIGNED_DOCUMENT_TYPE_PKCS7
SignatureInfo.SIGNED_DOCUMENT_TYPE_XML
SignatureInfo.SIGNED_DOCUMENT_TYPE_TIMESTAMP
```

Note: `SIGNED_DOCUMENT_TYPE_TIMESTAMP` is used for creating / verifying a timestamp token. When timestamp token creation is required make sure that **Signature Enhancement** is checked on **Advance Setting** tab of **Verification Profile**

5.5.4 Signature Info Methods

The following is a list of the main methods of the Signature Info class:

Signature Info Method	Purpose
`AddRespondWithItem(string respondWith)`	Adds 'respond with' item(s) for the signature request. The respond with items will then be returned in the response. The method can be called multiple times to add multiple 'respond with' items into the request. The values are as for the Verification Request class.
`SetCertificate(byte[])`	Specifies the signing certificate used when creating the signature.
`SetContentHash(byte[] contentHash, string contentHashAlgorithm)`	Specifies the hash algorithm and content hash. If this method is called then it means the content hash will be sent in the request instead of the full content.
`SetHistoricalValidation (DateTime)`	It specifies the date and time at which the signature needs to be validated.
`SetSignatureFormat(string signatureFormat)`	Specifies the signature format, which can be one of: - `SIGNATURE_FORMAT_A` - `SIGNATURE_FORMAT_B` - `SIGNATURE_FORMAT_C` - `SIGNATURE_FORMAT_OTHER.` (Note, currently format A, C and OTHER are supported. In format A, where one of more signers produces an individual signature on the document. The signatures are detached in this case. Each of these signatures contains a single signer info type). In format C, where one signature wraps another signature. Each signature has only one signer info type In format OTHER, where ADSS Verification Service intelligently detects the signature format.

5.6 Certificate Validation Requests

5.6.1 Certificate Validation Request Constructor

The Certificate Validation Request Class is used to ask the Verification Service to validate one or more certificates.

The following constructor is used to build the initial Certificate Validation Request message and specifies the clientID of the calling business application plus a transaction id reference.

```
var certValidateRequest = new CertificateValidationRequest(clientID,
transID);
```

5.6.2 Certificate Validation Request Methods

The following methods are inherited from the Verification Request class and described in section 5.4:

`AddKeyUsage, AddRespondWithItem, Send, SetCertificateQualityLevel, SetGatewayCompiance, SetHashAlgoQualityLevel, SetHistoricalValidation,`

`SetIndependentAssuranceLevel, SetPeppolCompliance, SetProfileID, SetPublicKeyAlgoQualityLevel, SetRequestMode, SetReturnverificationReport.`

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

The following is a list of the methods that are specific to the Certificate Validation Request Class:

Certificate Validation Request Method	Purpose
`AddCertificateInfo (certficateInfo)`	Adds a certificate into the request. (Currently only one certificate is supported by a single request).
`SetReturnVerificationTimeInfo( )`	Asks the Verification Service to return the time at which the certificate was validated.

5.6.3 Certificate Info Class Constructor

The Certificate Info class is used to specify a certificate to be verified and, once populated, is added into Certificate Validation request.

The following constructor is used to build the Certificate Info class instance and specifies a `certificateID`, a `certificateType` and the `certificateToValidate` (provided as a document file path or as a `byte[]`).

```
var certificateInfo = new CertificateInfo(certificateID, certificateType, certificateToValidate);
```

Currently the only valid value for `certificateType` is `CertificateInfo.CERTIFICATE_TYPE_X509.`

5.6.4 Certificate Info Methods

The following is the main method of the Certificate Info class:

Certificate Validation Request Method	Purpose
`SetHistoricalValidation (DateTime)`	Specifies the date and time at which the certificate needs to be validated.

5.7 Sending the Verification Request

Once the signature verification or certificate validation request message has been built using the above methods, it can then be sent to ADSS Server using the following call:

```
var verifyResponse = (VerificationResponse)verifyRequest.Send(string URL);
```

The `URL` is that of the Verification Service e.g. http://machine-name:8777/adss/verification/dss when using DSS mode `or` http://machine-name:8777/adss/verification/hsvi when using HTTP mode.

5.7.1 Example of creating and sending a Signature Verification Request

In the following example, a signature verification request is created. Specific quality levels are set for the signing certificate and various 'respond with' items are requested from the signing certificate. In addition, a fully detailed verification report and validation of the signature at an historic time is requested.

```
// Construct Signature Verification Request
var sigVerifyRequest = new SignatureVerificationRequest(clientID, transID);
sigVerifyRequest.SetCertificateQualityLevel("5");
sigVerifyRequest.SetPublicKeyAlgoQualityLevel("5");
sigVerifyRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_BASIC_CONSTRAINTS);
sigVerifyRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_KEY_USAGE);
sigVerifyRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_FROM);
sigVerifyRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_TO);
sigVerifyRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_ISSUER_NAME);
sigVerifyRequest.SetRequestMode(VerificationRequest.DSS);

// Add verification report elements
sigVerifyRequest.SetReturnVerificationReport(true, true, true, true, VerificationRequest.REPORT_LEVEL_ALL_DETAILS);
sigVerifyRequest.SetReturnVerificationTimeInfo();

// Construct and add signature element into the request
var signatureInfo = new SignatureInfo(requestID, inFile, SignatureInfo.SIGNED_DOCUMENT_TYPE_XML);
signatureInfo.SetSignatureFormat(SignatureInfo.SIGNATURE_FORMAT_OTHER);
sigVerifyRequest.AddSignatureInfo(signatureInfo);

// Ask for historic signature validation
sigVerifyRequest.SetHistoricalValidation(dateTime);

// Send Signature Verification Request
var verifyResponse = (VerificationResponse)sigVerifyRequest.Send(serviceAddress);
```

5.7.2 Example of creating and sending a Certificate Validation Request

In the following example, a certificate validation request is created. Specific quality levels are set for the certificate and various 'respond with' items are requested from the signing certificate. In addition, a fully detailed verification report and validation of the certificate at an historic time is required.

```
// Construct Certificate Validation Request
var certValidateRequest = new CertificateValidationRequest(clientID, transID);
certValidateRequest.SetCertificateQualityLevel("5");
certValidateRequest.SetPublicKeyAlgoQualityLevel("5");
certValidateRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_BASIC_CONSTRAINTS);
certValidateRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_KEY_USAGE);
certValidateRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_FROM);
certValidateRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_VALID_TO);
certValidateRequest.AddRespondWithItem(VerificationRequest.RESPOND_WITH_ISSUER_NAME);
certValidateRequest.SetRequestMode(VerificationRequest.DSS);

// Add verification report elements
certValidateRequest.SetReturnVerificationReport(true, true, true, true, VerificationRequest.REPORT_LEVEL_ALL_DETAILS);
certValidateRequest.SetReturnVerificationTimeInfo();

// Construct and add signature element into the request

var certificateInfo = new CertificateInfo(certificateID, CertificateInfo.CERTIFICATE_TYPE_X509, inFile);
certValidateRequest.AddCertificateInfo(certificateInfo);

// Ask for historic signature validation
certValidateRequest.SetHistoricalValidation(dateTime);

// Send Certificate Validation Request
var verifyResponse = (VerificationResponse)certValidateRequest.Send(serviceAddress);
```

5.8 Verification Response Methods

5.8.1 Verification Response Methods

The following methods are inherited from the generic Response and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetRequestID, GetSigningCertificates, GetStatus, IsSuccessful.`

In addition, the following methods are specific to the Verification Response Class:

Verification Response Method	Purpose
`GetCertificateQualityLevel () returns string`	Returns the certificate quality level used during the verification process.
`GetHashAlgoQualityLevel() returns string`	Returns the hash algorithm quality level used during the verification process.
`GetIndependentAssuranceLevel () returns string`	Returns the independent assurance level used during the verification process.
`GetOriginatorId() returns string`	Returns the Originator ID sent in the request.
`GetOverallAssertionStatus( ) returns string`	Returns the overall assertion status, one of the following: trusted, not trusted or indeterminate.
`GetProfileId() returns string`	Returns the Verification Profile ID used to process the request.
`GetPublicKeyAlgoQualityLevel () returns string`	Returns the public key quality level used during the verification process.
`GetReason() returns string`	Returns a comma separated list of all Signature Ids which were not valid. For a PDF signature the Signature ID is the signature field name.
`GetResponderName() returns string`	Returns the subject name of the verification response signing certificate.

`GetResponseId() returns string`	Returns the transaction Id received in the request. This is used to correlate the request and response.
`GetResponseType() returns string`	Returns the type of response, either signature or certificate validation.
`GetResultMajor() returns string`	(Used in OASIS DSS protocol mode). The Result Major status message provides the main result from processing the verification request. The status can be one of the following: - `Success` - `Requester Error` - `Responder Error` - `Insufficient Information`

`GetResultMinor() returns string`	(Used in OASIS DSS protocol mode).
	The possibilities for the Result Minor status messages depend upon the Result Major message.
	For Result Major = Success, Result Minor values are:
	`valid:signature:OnAllDocuments`
	`valid:signature:NotAllDocumentsReferenced`
	`invalid:IncorrectSignature`
	`valid:signature:HasManifestResults`
	`valid:signature:InvalidSignatureTimestamp`
	For Result Major = Requester Error, Result Minor values are:
	`ReferencedDocumentNotPresent`
	`MoreThanOneRefUriOmitted`
	`InvalidRefUri`
	`NotParseableXMLDocument`
	`NotSupported`
	`InappropriateSignature`
	`(Other values are possible)`
	For Result Major = ResponderError, Result Minor values are:
	`GeneralError`
	`KeyLookupFailed`
	`(Other values are possible)`
	For Result Major = Insufficient Information, Result Minor values are:
	`CrlNotAvailable`
	`OcspNotAvailable`
	`CertificateChainNotComplete`

`GetSignerIdentity() returns string`	Returns the first signer identity name.
`GetSigningTimeInfo() returns DateTime`	Returns the signing time of the first signature.
`GetTimeInstant() returns DateTime`	Returns the date and time when the response message was formed.
`GetTimeStampedSignatue() returns byte[]`	Returns a first TimeStampedSignature.
`GetUpdatedSignature() returns byte[]`	Returns the first updated signature.

`PublishUpdatedSignature(Stream)`	Publishes the updated signatures to the specified stream.
`GetUpdatedSignaturesList() returns List<byte[]>`	Returns the list of updated signatures when multiple signature are send
`GetUpdatedSignature(String a_signatureId) returns byte[]`	This method returns the updated signature bytes for a specified signature ID. Same signature Id which is sent in request. Return null if no updated signature is found against specified signature ID.
`GetUpdatedSignatures() return HashTable<String,byte[]>`	Returns the `HashTable<String,byte[]>` which contains updated signatures with its respective signature IDs. Never return null. In .Net this returns `Dictionary<String, byte[]>`
`GetUpdatedSignatureType() returns string`	Returns the updated signature type.
`GetVerificationReport() returns VerificationReport`	Returns the OASIS DSS-X signature verification report if this was set in the request.
`GetVerificationTimeInfo()`	Returns the date and time for which the signature is validated.
`GetVerifyInfo() returns ArrayList`	Returns a list of VerifyInfo elements, each item corresponding to one of the SignatureInfos in the request.
`GetVersion() returns string`	Returns the version of the Verification Service interface.

5.8.2 Verify Info Class

As explained above `GetVerifyInfo()` returns a list of `VerifyInfo` elements, each one corresponding to one of the `SignatureInfos` in the request. The following are the methods available to access this information:

Verify Info Method	Purpose
`GetCertificate() returns string`	Returns the signer certificate as base64 encoded string.
`GetCertificateQualityLevel() returns string`	Returns the certificate quality level used during the verification process.
`GetFailureReason() returns string`	Returns 'null' if the signature verified correctly otherwise it contains the failure reason.
`GetHashAlgoQualityLevel() returns string`	Returns the hash algorithm quality level used during the verification process.
`GetHistoricalValidation() returns DateTime`	Returns the historical validation date and time that was sent in the request.
`GetIndependentAssuranceLevel () returns string`	Returns the independent assurance level used during the verification process.
`GetNextUpdate() returns string`	Returns the content of the 'nextUpdate' field of the CRL from which the signer certificate revocation status is checked
`GetPublicKeyAlgoQualityLevel () returns string`	Returns the public key quality level used during the verification process.
`GetResponseItems() returns ArrayList`	Returns the list of response items as specified in the 'respond with' calls.

`GetSignatureId() returns string`	Returns the Signature ID. For a PDF signature the Signature ID is the signature field name.
`GetSignatureStatus() returns string`	Returns status information for a particular signature, either valid or invalid.
`GetSignerDN() returns string`	Returns the subject name of the signer certificate.
`GetThisUpdate() returns string`	Returns the content of the 'thisUpdate' field of the CRL from which the signer certificate revocation status is checked
`GetSigningTimeInfo() returns DateTime`	(Used in HTTP mode). It returns the signing time of the signature.
`GetVerificationTimeInfo() returns DataTime`	(Used in HTTP mode). It returns the date time on which signature is validated.
`GetContentType() returns string`	(Used in HTTP mode). It returns the Content-Type.
`GetIssuerName() returns string`	(Used in HTTP mode). It returns the IssuerName of the signer certificate.
`GetCrlNumber() returns string`	(Used in HTTP mode). It returns the CRL number.
`GetCertificateSerialNumber() returns string`	(Used in HTTP mode). It returns the Serial Number of the signer certificate.
`GetValidFrom() returns string`	(Used in HTTP mode). It returns the validFrom of the signer certificate.
`GetValidTo() returns string`	(Used in HTTP mode). It returns the validTo of the signer certificate.
`IsQcCompliance() returns Boolean`	(Used in HTTP mode). It returns whether or not the signer certificate has Qualified Certificate Statements extension.
`IsQcSSCD() returns Boolean`	(Used in HTTP mode). It returns whether or not the signer certificate has Secure Signature Creation Device QC statement.
`GetQcCaRetentionPeriod() returns int`	(Used in HTTP mode). It returns the CA retention period QC statement.
`GetQcCurrency()returns String`	(Used in HTTP mode). It returns the Currency Code in Transaction Limit QC statement.
`GetQcAmount() returns String`	(Used in HTTP mode). It returns the Amount in Transaction Limit QC statement.
`GetQcExponent() returns String`	(Used in HTTP mode). It returns the Exponent in Transaction Limit QC statement.
`GetEpesSignaturePolicyId() returns String`	(Used in HTTP mode). It returns EPES Signature Policy OID.
`GetEpesSignaturePolicyUri() returns String`	(Used in HTTP mode). It returns EPES Signature Policy URI.

`GetEpesSignaturePolicyUserNotice() returns String`	(Used in HTTP mode). It returns EPES Signature Policy User Notice.
`GetLeiNumber() returns String`	(Used in HTTP mode). It returns LEI number.
`GetLeiRole() returns String`	(Used in HTTP mode). It returns LEI role.
`GetEtsiValidationReport() returns ValidationReportType`	(Used in HTTP mode). It return the ETSI Validation Report when Trust Anchor from TSL is configured in Verification Profile. The report is compliant to ETSI TS 119 102-2 v1.4.1 Schema
`GetSignatureValidationIndication() returns String`	(Used in HTTP mode). It return the main indication of signature validation via TSL
`GetSignatureValidationSubIndication() returns String`	(Used in HTTP mode). It return the sub indication of signature validation via TSL
`IsQcCCLegislated() returns Boolean`	(Used in HTTP mode). It returns whether or not the signer certificate has Qualified Certificate Statements extension for showing the Legislated Countries.
`GetLegislatedCountries() returns String`	(Used in HTTP mode). It returns the Legilated Countries include in the QC statement.

5.9 Verification Service Reports

The following classes are provided for accessing information returned in OASIS-DSSX Verification Reports:

- Verification Report Class
- Individual Report Class
- Result Class
- Certificate Validity Class
- Certificate Path Validity Class
- CertID Class

5.9.1 Verification Report Class

The Verification Report class is the main container for holding the set of individual Verification Reports.

As stated previously, if Verification Service Reports have been requested these can be retrieved using the `GetVerificationReport()` method. The individual reports can then be accessed with the following methods:

Verification Report Method	Purpose
`GetIndividualReports() returns ArrayList`	Returns the set of Verification Reports.
`GetVerificationTime() returns DateTime`	Returns the signature validation time.
`GetVerifierIdentity() returns X509Certificate[]`	Returns the chain of certificates that identify the verifier. This can be useful for timestamped reports.

5.9.2 Individual Report Class

The Individual Report class contains an individual Verification Report and has the following methods:

Individual Report Method	Purpose
`GetIndividualReportType() returns string`	Returns the individual report type i.e. CERTIFICATE or SIGNATURE.
`GetCertValidity() returns List`	Returns a list of certificate validity information for each certificate in the path.
`GetCertificatePathValidity() returns CertificatePathValidity`	Returns the certificate validation path – see the Certificate Path Validity class below.
`GetResult() returns Result`	Returns the overall Signature Verification result – see the Result class below.
`GetSigMathOK() returns Result`	Returns the result of the signature check itself.
`GetSignatureAlgorithm() returns string`	Returns the signature algorithm as an OID in the case of Detailed Signature Report .
`GetSignatureAlgorithmSuitability() returns Result`	Returns the signature algorithm suitability result.
`GetSignatureObjectId() returns string`	Returns the Signature Identity of the signed data or validation data.
`GetSignerCertChainReferences() returns List<CertID>`	Returns a list of the signer certificate chain references – see the CertID class below for methods to access these objects.
`IsFormatOK() returns Result`	Returns the result of format checking of a signature – see the Result class below.

In addition to the above methods, the following are also available and documented in the JavaDoc/Sandcastle class documentation:

`GetSignatureProductionPlaceCity, GetSignatureProductionPlaceCountryName, GetSignatureProductionPlacePostalCode, GetSignatureProductionPlaceStateOrProvince, GetSignerLocation, GetSignerRole, IsSignatureHasVisibleContent.`

The first four are specific to XAdES/CAdES signatures and the last three to PDF signatures.

5.9.3 CertID Class

The CertID class provides information about the certificates in the certificate chain.

CertID Method	Purpose
`GetDigestMethod() returns string`	Returns the hash/digest method OID.
`GetDigestValue() returns string`	Returns the digest value as base64 encoded string.
`GetIssuerName() returns string`	Returns the issuer name.
`GetSerialNumber() returns string`	Returns the issuer serial number.

5.9.4 Certificate Validity Class

The Certificate Validity class contains validity information for an individual certificate in the path. The class provides many methods but these can be divided into two types, those that provide validity status information about the certificate and those that provide information related to the certificate (e.g. certificate related fields or OCSP/CRL information).

5.9.5 Certificate Validity: Status Information Methods

The following certificate status methods all return a Result class object (see result class for details):

Certificate Validity Method	Purpose
`IsCertStatusOK() returns Result`	Returns the result of the certificate revocation status check.
`AreExtensionsOK() returns Result`	Returns the status of the certificate extensions check.
`GetCrlSignature() returns Result`	Returns the status of the CRL signature check.
`GetCrlSignatureAlgorithmSuitability() returns Result`	Returns the status of the CRL Signature Algorithm Suitability check.
`GetOcspSignatureAlgorithmSuitability() returns Result`	Returns the status of the OCSP Signature Algorithm Suitability check.
`GetOcspSingleResponseStatus() returns Result`	Returns the result of checking OCSP responder revocation
`GetRevocationReason() returns Result`	Returns the Revocation Reason. If the certificate is Revoked and the revocation reason is present in the verification Report then this information will also be included in the Result Minor element using a URI to indicate the revocation reason (e.g. keyCompomise).
`GetSignatureAlgorithmSuitability() returns Result`	Returns the status of the Signature Algorithm Suitability check.
`IsChainingOK() returns Result`	Returns the result of the certificate chaining check.
`IsCrlSigMathOK() returns Result`	Returns the result of the CRL signature verification.
`IsOcspSigMathOK() returns Result`	Returns the result of the OCSP response signature verification.
`IsSigMathOK() returns Result`	Returns the result of the signature verification.

`IsValidityPeriodOK() returns Result`	Returns the certificate expiry result.

5.9.6 Certificate Validity: Certificate, CRL and OCSP Information

Various other methods are provided by the Certificate Validity class to retrieve information about the certificate. These are some of the most useful:

Certificate Status Method	Purpose
`GetCertValue() returns string`	Returns the certificate as base64 encoded string.
`GetCrlCertificatePathValidity() returns CertificatePathValidity`	Returns the result of CRL issuer certificate path validity test – see also the Certificate Path Validity class below.
`GetCrlValue() returns string`	Returns the CRL as base64 encoded string.
`GetOcspCertificatePathValidity() returns CertificatePathValidity`	Returns the result of OCSP responder chain validation – see also the Certificate Path Validity class below.
`GetOcspValue() returns string`	Returns the OCSP Response as base64 encoded string.

The remaining methods below provide various fields from the certificate or information regarding OCSP results, the CRL or CRL issuer. These are described in the JavaDoc/Sandcastle class documentation:

```
GetCertExtension, GetCertIssuerName, GetCertIssuerSerialNo,
GetCertSubject, GetCertValidityPeriodNotAfter,
GetCertValidityPeriodNotBefore, GetCertVersion, GetCrlExtensions,
GetCrlIssuer, GetCrlIssueTime, GetCrlNextUpdate, GetCrlNumber,
GetCrlSignatureAlgorithm, GetCrlThisUpdate, GetCrlURI, GetCrlVersion,
GetOcspProducedAt, GetOcspResponderId, GetOcspResponseExtensions,
GetOcspSignatureAlgorithm, GetOcspSingleResponseHashAlgorithm,
GetOcspSingleResponseIssuerKeyHash, GetOcspSingleResponseIssuerNameHash,
GetOcspSingleResponseNextUpdate, GetOcspSingleResponseSerialNumber,
GetOcspSingleResponseThisUpdate, GetOcspURI, GetOcspVersion,
GetRevocationDate, GetSignatureAlgorithm,
GetSingleOcspResponseExtensions, GetSubject,
```

5.9.7 Certificate Path Validity Class

The Certificate Path Validity class provides information about each certificate in the validation path:

Certificate Path Validity Method	Purpose
`GetPathValiditySummary() returns Result`	Returns a summary of the Certificate Path Validation.
`GetCertValidity() returns ArrayList`	Returns a list of certificate validity information for each certificate in the path.
`GetSignerCertIssureName() returns string`	Returns the signer certificate issuer name.
`GetSignerCertSerialNo() returns string`	Returns signer certificate serial number.

5.9.8 Result Class

The Result class provides access to the DSS-X Result Major, Result Minor and Result Message results. These are returned to provide the overall status of various parts of the Verification Report. The actual values vary by context, see below:

Result Method	Purpose

`GetResultMajor() returns string`	Returns the Result Major status from the Verification Report which must be present in the Report.
	Result Major can take different values depending upon context.
	For the overall status of the report the following values are possible:
	`urn:oasis:names:tc:dss:1.0:detail:valid`
	`urn:oasis:names:tc:dss:1.0:detail:invalid`
	`urn:oasis:names:tc:dss:1.0:detail:indetermined`
	For the status of a certificate the following values are possible:
	`urn:oasis:names:tc:dss-x:1.0:profiles:verificationreport:certstatus:good`
	`urn:oasis:names:tc:dss-x:1.0:profiles:verificationreport:certstatus:revoked`
	`urn:oasis:names:tc:dss-x:1.0:profiles:verificationreport:certstatus:unknown`

`GetResultMinor() returns string`	Returns the Result Minor status from the Verification Report. Where Result Major has the status 'invalid' 'indetermined', 'revoked' or 'unknown', then Result Minor will be present to provide further details.
`GetResultMessage() returns string`	Returns the Result Message from the Verification Report. This is an optional field so may return null but could also provide other information as is the case with a revoked certificate (when it holds the revocation time).

5.10 Verification Service Sample Code

Java and .Net sample code is provided as part of the ADSS Client SDK and can be used to make Verification Service requests and to process the Verification Service responses.

The Java API provides the required classes under the packages:

```
com.ascertia.adss.client.api.verification
com.ascertia.adss.client.api.verification.report
```

The .Net API provides the required classes under the namespaces:

```
Com.Ascertia.ADSS.Client.API.Verification
Com.Ascertia.ADSS.Client.API.Verification.Report
```

5.10.1 How to verify Document using document Hash and Signature

When necessitating document confidentiality and desiring to avoid transmitting the entire document for verification, it is recommended to transmit solely the signature bytes and document hash to the verification service for the purpose of signature validation.

For details see the following JAVA/DotNet Sample API:

- `samples/src/com/ascertia/adss/client/samples/verification/VerifySignatureUsingDocumentHash.java`

5.10.2 Java API Sample Code

The following sample programs demonstrate how the Java API can be used to send a Verification Service request and to process the response:

- samples/src/com/ascertia/adss/client/samples/verification/VerifyPDF.java
- samples/src/com/ascertia/adss/client/samples/verification/VerifyPdfReport.java
- samples/src/com/ascertia/adss/client/samples/verification/VerifyPKCS7.java
- samples/src/com/ascertia/adss/client/samples/verification/VerifyXmlEnveloped.java
- samples/src/com/ascertia/adss/client/samples/verification/VerifyCertificate.java
- samples/src/com/ascertia/adss/client/samples/verification/historical/VerifyPDF.java
- samples/src/com/ascertia/adss/client/samples/verification/historical/VerifyPKCS7.java
- samples/src/com/ascertia/adss/client/samples/verification/historical/VerifyXmlEnveloped.java
- samples/src/com/ascertia/adss/client/samples/verification/historical/VerifyCertificate.java
- samples/src/com/ascertia/adss/client/samples/verification/VerifyMsOfficeDocument.java
- samples/src/com/ascertia/adss/client/samples/verification/VerifyMsOfficeSignatures.java
- samples/src/com/ascertia/adss/client/samples/verification/VerificationGenerateTimeStampToken.java
- samples/src/com/ascertia/adss/client/samples/verification/VerifySignatureUsingDocumentHash.java

Precompiled and ready to run version of the above sample programs can be found at:

- samples/bin/VerifyPDF.bat
- samples/bin/VerifyPdfReport.bat
- samples/bin/VerifyPKCS7.bat
- samples/bin/VerifyXmlEnveloped.bat
- samples/bin/VerifyCertificate.bat
- samples/bin/VerifyPDF_historical.bat
- samples/bin/VerifyPKCS7_historical.bat
- samples/bin/VerifyXmlEnveloped_historical.bat
- samples/bin/VerifyCertificate_historical.bat
- samples/bin/VerifyMsOfficeDocument.bat
- samples/bin/VerifyMsOfficeSignatures.bat
- samples/bin/VerificationGenerateTimeStampToken.bat
- samples/bin/VerifyTimeStampToken.bat
- samples/bin/VerifySignatureUsingDocumentHash.bat

5.10.3 .Net API Sample Code

The following sample programs demonstrate how the .Net API can be used to send a Verification Service request and to process the response:

- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/VerifyPDF.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/VerifyPdfReport.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/VerifyPKCS7.cs

- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/VerifyXmlEnveloped.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/VerifyCertificate.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/VerifyMsOfficeDocument.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/VerifyMsOfficeSignatures.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/Historical/VerifyPDF.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/Historical/VerifyPKCS7.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/Historical/VerifyXmlEnveloped.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/Historical/VerifyCertificate.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/VerificationGenerateTimeStampToken.cs
- samples/src/Com/Ascertia/ADSS/Client/Samples/Verification/VerifySignatureUsingDocumentHash.cs

Precompiled and ready to run version of the above sample programs can be found at:

- samples/bin/VerifyPDF.bat
- samples/bin/VerifyPdfReport.bat
- samples/bin/VerifyPKCS7.bat
- samples/bin/VerifyXmlEnveloped.bat
- samples/bin/VerifyCertificate.bat
- samples/bin/VerifyMsOfficeDocument.bat
- samples/bin/VerifyMsOfficeSignatures.bat
- samples/bin/VerifyPDF_historical.bat
- samples/bin/VerifyPKCS7_historical.bat
- samples/bin/VerifyXmlEnveloped_historical.bat
- samples/bin/VerifyCertificate_historical.bat
- samples/bin/VerificationGenerateTimeStampToken.bat
- samples/bin/VerifyTimeStampToken.bat
- samples/bin/ VerifySignatureUsingDocumentHash.bat

5.11 ADSS Verification Service Supported Algorithms

The following is a list of signing/hashing algorithms and key lengths that ADSS Verification Service supports:

ADSS Service	Signature Algorithm	Hashing Algorithm	Algorithm / Key Sizes
Verification	• SHA1WithRSAEncryption • SHA192WithRSAEncryption • SHA224WithRSAEncryption • SHA256WithRSAEncryption • SHA384WithRSAEncryption • SHA512WithRSAEncryption • SHA3-224WithRSAEncryption • SHA3-256WithRSAEncryption • SHA3-384WithRSAEncryption • SHA3-512WithRSAEncryption • MD5WithRSAEncryption • RipeMD128WithRSAEncryption	SHA-1 SHA-192 SHA-224 SHA-256 SHA-384 SHA-512 SHA3-224 SHA3-256 SHA3-384 SHA3-512 MD5 RipeMD128	Below are the list of Key Algorithms and Sizes • RSA: 1024, 2048, 3072, 4096, 6144, 8192 • ECDSA Curves: ○ NIST P-160, P-192, P-224, P-256, P-384,P-521 ○ TeleTrust (Brainpool) brainpoolp160r1, brainpoolp160t1, brainpoolp192r1,

	• RipeMD160WithRSAEncryption • SHA1withECDSA • SHA224withECDSA • SHA256withECDSA • SHA384withECDSA • SHA512withECDSA • SHA3-224withECDSA • SHA3-256withECDSA • SHA3-384withECDSA • SHA3-512withECDSA • RSASSA-PSS	RipeMD160 GOST-3411-94	brainpoolp192t1, brainpoolp224r1, brainpoolp224t1, brainpoolp256r1, brainpoolp256t1, brainpoolp320r1, brainpoolp320t1, brainpoolp384r1, brainpoolp384t1, brainpoolp512r1, brainpoolp512t1 • GOST-3410-2001 512

5.12 Error Codes

ADSS Verification Server returns the following statuses in case of any failure:

Error Code	Error Message
42001	Historic time is later than the signing time.
42002	Invalid signature.
42003	CAdES-BES/XAdES-BES signature verification failed.
42004	CAdES-EPES/XAdES-EPES signature verification failed.
42005	CAdES-T/XAdES-T signature timestamp verification failed.
42006	CAdES-C/XAdES-C signature verification failed.
42007	CAdES-X/XAdES-X signature verification failed.
42008	CAdES-A/XAdES-A signature verification failed.
42009	Signature verification failed - either failed to decrypt the signature or hash does not match.
42010	Quality level is too low.
42011	Signed request is required.
42012	Signature verification failed.
42013	Request structure is invalid.
42014	Failed to sign verification response.
42015	Failed to authenticate verification request.
42016	ADSS Verification service license is expired.
42017	ADSS Verification service is stopped.
42018	ADSS Verification service not enabled in license.
42019	Failed to verify embedded revocation information.
42020	Signature type not configured.
42021	Signature type not enabled in license.
42022	Validation at historical time not enabled in license.
42023	Invalid signature type.
42024	Invalid PDF document.

42025	No signature found.
42026	Validation at historical time is not configured.
42027	Invalid signature.
42028	Verification profile is not allowed to the client.
42029	An internal error occurred during processing the request.
42030	Certificate validation not enabled in license.
42031	Failed to create CMS signature.
42032	Signature grace period not yet elapsed.
42033	Signing time not present in signature.
42034	Signature enhancement failed.
42036	Verification service not allowed.
42037	Failed to verify revocation information embedded in signature timestamp token.
42038	Failed to verify revocation information embedded in reference timestamp token.
42039	Failed to verify revocation information embedded in archive timestamp token.
42040	Signature verification failed.
42041	Long term signature does not contain a secure timestamp token.
42042	Signature does not contain signing time.
42043	ADSS Verification service not enabled in system.
42044	Content timestamp verification failed.
42045	Failed to verify embedded revocation information for content timestamp token.
42046	Signature enhancement is not supported in PDF collection.
42047	Verification profile is inactive.
42048	Verification profile does not exist or marked inactive.
42049	Document is not available.
42050	Originator info not available.
42051	No Time stamping Authorities configured for this verification profile.
42052	Signature enhancement is not allowed to this verification profile.
42053	Failed to generate time stamped verification response.
42054	Default profile not configured and neither found in request.
42055	Hash algorithm is not in the allowed list.
42056	Key algorithm is not in the allowed list.
42057	Key length is not in the allowed list.
42058	XmlDsig signature cannot be enhanced.
42059	CMS signature cannot be enhanced.

6 ADSS Certification Service

The ADSS Server Certification Service provides support for following protocols:

- Ascertia proprietary XML protocol
- The standard Certificate Management over CMC protocol (CMC)
- The standard Certificate Enrollment over EST protocol (EST)
- via a Java or .Net API.

Following operations are accessible using XML protocol:

- Request and Revoke Certificates
- Renew or delete certificates
- Change the password for a server held key
- Recover a server held key
- Manage the user keys on RAS/SAM
- Get Certificates

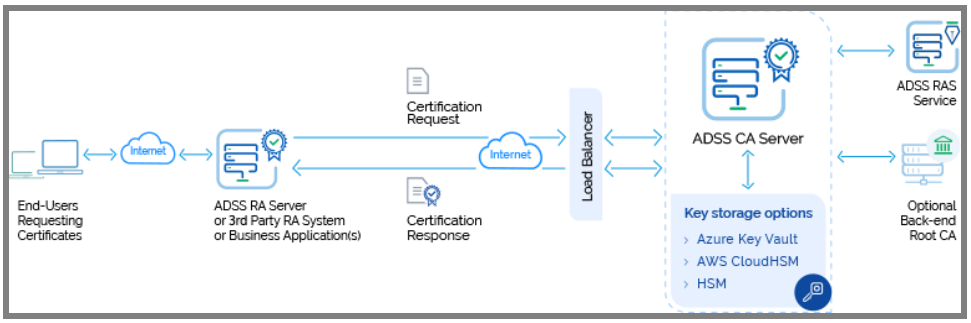
Following operations are accessible using CMC protocol:

- Request and Revoke Certificates

Following operations are accessible using EST protocol:

- Request and Revoke Certificates
- Renew/Rekey certificates
- Get CA Certificates
- Request Private Key and Certificate using server key generation

Registered Business Applications send requests to ADSS Server, referring to a particular Certification Profile, and receive responses back. Normally most certification related parameters do not need to be sent in the request as they are already set up in the Certification Profile.



The 'certifying' Certification Authority (CA) may be a self-signed CA managed by ADSS Server or it may link to a higher level or root CA which may either be at the server or it can be external.

6.1 Certification Use Cases and Ascertia Protocol Schema

Various certification use cases are possible using the ADSS Certification Service and these and the Ascertia proprietary protocol schemas are discussed in section 20.

6.2 Certification Profiles

The ADSS Certification Service requires that Certification Profiles are defined at ADSS Server. These profiles specify which CA will issue the certificates, the key length and key type to be used, the certificate

validity period, any default distinguished name parameters (e.g. country name, organisational unit etc.) and whether the certificate will be automatically renewed on expiry.

Refer to the following online admin guide for an explanation of Certification Profile settings:

Step 2 - Creating a Profile (ascertia.com)

6.3 The Certification Service API

The Certification Service API is provided as part of the ADSS Client SDK and consists of a Certification Request and a Certification Response class.

6.4 Certification Request Class

The following constructor is used to build the initial Certification Request. There are two variants of this depending upon whether the certificate is held at ADSS Server as a database item (and referenced using a certificate name alias or as an X509 certificate object.

```
var certificationRequest = new CertificationRequest(clientID, requestType,
certificateAlias or X509CertificateObject );
```

The `clientID` identifies the business application that is making the call. This `clientID` must already be registered at ADSS Server.

The `requestType` identifies one of the following available services i.e.:

```
REQUEST_TYPE_CREATE_CERTIFICATE     Certificate Creation
REQUEST_TYPE_IMPORT_CERTIFICATE     Certificate Import
REQUEST_TYPE_RENEW_CERTIFICATE      Certificate Renewal
REQUEST_TYPE_REKEY_CERTIFICATE      Certificate Rekey
REQUEST_TYPE_DELETE_CERTIFICATE     Certificate Deletion
REQUEST_TYPE_REVOKE                 Certificate Revocation
REQUEST_TYPE_RECOVER_KEY            Recover a .pfx file from the server
REQUEST_TYPE_CHANGE_PASSWORD        Change password of a server held .pfx
file
REQUEST_TYPE_AUTHORIZE_CERTIFICATE  Bind authorization certificate
REQUEST_TYPE_UNAUTHORIZE_CERTIFICATE Unbind authorization certificate
REQUEST_TYPE_SEND_SCT               Send the SCTs to Certification Service
                                    to add this into the TLSs server
                                    certificate. For details the see section
                                    (6.4.13 Example of Send SCT).
REQUEST_TYPE_CACERT                 CA Certificate Retrieval
REQUEST_TYPE_GET_CERTIFICATES       Certificates Retrieval
```

6.4.1 Certificate Request methods

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

```
ToString, WriteTo, Send (overridden), SetProxy, SetRequestID,
SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion,
SetSSLClientCredentials, SetTimeout, SetVerifyResponse.
```

In addition, the following methods are specific to the Certificate Request class:

Certificate Request Method	Purpose
`AddRespondWithItem(string)`	This specifies the items that will be returned in the response. The method can be called multiple times if multiple items are required. Currently the following items can be requested: - `RESPOND_WITH_CERTIFICATE` - `RESPOND_WITH_PKCS_7` - `RESPOND_WITH_PKCS_12`

	- RESPOND_WITH_PKCS_10 - RESPOND_WITH_EXPIRY_DATE - RESPOND_WITH_PASSWORD
OverrideProfileAttribute (policyOverride, string attributeValue)	This method overrides specific policy values (if this is permitted by the Certification Policy) and again this may be called multiple times. The following are possible policy overrides: - SUBJECT_DN - KEY_SIZE - SECURITY_LEVEL - KEY_TYPE - CURVE_TYPE - VALIDITY_PERIOD - VALIDITY_UNIT - VALID_TO SUBJECT_DN allows various distinguished name fields to be overridden e.g. CN, G, SN, T, OU, O, OI, E, L, ST, S, P, C, SERIALNUMBER, UID, B, EVL, EVS and EVC KEY_SIZE allows an RSA or ECDSA key length to be overridden. SECURITY_LEVEL allows security level of ML-DSA, ML-KEM or CLASSIC_MCELIECE key pair to be overridden. KEY_TYPE allows a key of a different type to be specified (either RSA, ECDSA, ML-DSA, ML-KEM or CLASSIC_MCELIECE). CURVE_TYPE allows the NIST P, SEC2 K or Brainpool curve type to be overridden for ECDSA key type. Supported Curves are: - NIST_P - SEC2_K - BRAINPOOL_R - BRAINPOOL_T Default NIST_P is used when not specified in either request or profile. VALIDITY_PERIOD changes the certificate lifetime and therefore the expiry date of the certificate. VALIDITY_UNIT change the certificate lifetime validity period unit of the certificate in 'MINS', 'HOURS', 'DAYS', 'MONTHS' or 'YEARS' i.e. VALIDITY_UNIT=MONTHS VALID_TO changes the certificate lifetime, it will set the certificate validTo date. It will override certificate validity in profile. The value must be a date string in format “yyyy-MM-dd'T'HH:mm:ss” e.g. 2020-02-10T15:53:23

	Note: It is alternate of VALIDITY_PERIOD/ VALIDITY_UNIT, one need to use VALID_TO or VALIDITY_PERIOD/VALIDITY_UNIT for certificate expiry date. If both are used then it will create problems with the time calculation.
`SetCertificate(X509Certificate or certificateFilePath)`	Specifies a certificate which will be the subject of the request (e.g. for a revocation request).
`SetCmcRequestMode(PKIRequestMode)`	For a CMC request, this specifies whether a simple request `(SIMPLE_PKI_REQUEST)` or full PKI request `(FULL_PKI_REQUEST)` is being made. A simple PKI request consists of a single PKCS#10 whereas the full PKI request consists of a Certificate Request Message Format (CRMF) message.
`SetIssuerDN(issuerDN)`	Specifies the issuer Distinguished Name to be used for CMC revocation request.
`SetOnholdInstructionCode (onHoldInstructionCode)`	Specifies the 'on hold' instruction code, one of: - `CALLISSUER` - `NONE` - `REJECT`
`SetPKCS10 (byte[] or filePath)`	Specifies the PKCS#10 certificate request.
`SetPKCS12 (byte[] or filePath)`	Specifies a PKCS#12 file.
`SetPkcs12NewPassword (string)`	Specifies a new password for the PKCS#12 and is used only if the request type is 'change password'.
`SetPkcs12Password (string)`	Specifies the PKCS#12 password. This method is only used if the key pair is generated at the server and held at the client.
`SetPKCS7 (byte[] or filePath)`	Specifies a PKCS#7 file containing the certificate chain.
`SetProfileId (string)`	Specifies the ADSS Server Certification Profile ID used to service the request.
`SetRequestMode(int)`	Specifies the Request Mode which can be either XML,CMC or EST. XML is the default. Note, if CMC is used the communication must be over mutually authenticated TLS, i.e. port 8779. See also 'SetSslClientCredentials'.
`addSubjectAlternativeName (string, string)`	It is used to add subject alternative name extension in X.509 certificate. The first parameter specifies SAN key. The possible values for key are: - `rfc822Name` - `dNSName` - `iPAddress` - `uniformResourceIdentifier` - `registeredID` - `ediPartyName` - `directoryName` - `otherName` while the second parameter specifies value for the SAN extension. If the provided key is "otherName", the string

	for this attribute should be in one of the following formats.
	`OID=value,encoding=UTF8String`
	`OID=value,encoding=OctetString`
	`OID=value,encoding=PrintableString`
	Consider the following for example:
	`1.2.3.4.5=Other Name, encoding=UTF8String`
	If the provided key is "ediPartyName", the string for this attribute should be in one of the following formats.
	`NameAssigner=DummyName,encoding:PrintableString,`
	`PartyValue=DummyPartyName,encoding:UniversalString`
	Consider the following for example:
	`DummyName,encoding:PrintableString= DummyPartyName,encoding:UniversalString`
	If the provided key is "directoryName", then a custom RDN can also be passed in the following format.
	`OID=value,encoding=UTF8String`
	Consider the following for example:
	`1.2.3.4.5=Custom RDN`
	This method can be called multiple times in order to add multiple names in subject alternative name extension.

`addCustomExtension(string, string)`	It is used to add custom extension in X.509 certificate. The first parameter defines the OID while the second parameter specifies the value of respective custom OID extension. For example: First parameter 1.3.6.1.4.1.59382.3.2 Second parameter = GERALD This method can be called multiple times in order to add multiple custom extension.

`SetRevocationReason (string)`	Specifies the Revocation Reason, one of:
	`REVOCATION_REASON_AACOMPROMISE`
	`REVOCATION_REASON_AFFILIATIONCHANGED`
	`REVOCATION_REASON_CACOMPROMISE`
	`REVOCATION_REASON_CERTIFICATE_HOLD`
	`REVOCATION_REASON_CESSATION_OF_OPERATION`
	`REVOCATION_REASON_KEYCOMPROMISE`
	`REVOCATION_REASON_PRIVILEGEWITHDRAWN`
	`REVOCATION_REASON_REMOVE_FROM_CRL`
	`REVOCATION_REASON_SUPERSEDED`
	`REVOCATION_REASON_UNSPECIFIED`

`SetAuthCertAlias (string)`	Specifies the authorization certificatete alias for second factor authentication to generate qualified signature on the server.
`setUserId (String)`	User ID to identify and authenticate at the RAS/SAM server. It is a mandatory parameter in the user enrollment request.

`addSCT(SctInfo)`	SCT object will be added by the client application with information like version, timestamp, signature etc.
`setClientSecret(String)`	Client Secret to authenticate the client application in caes EST APIs as it requires the Basic Authentication.
`SetSubjectDN(subjectDN)`	Specifies the Subject Distinguished Name to be used to get the certificate. e.g. `CN=Jhon~encoding=UTF8String`
`SetStatus(status)`	Specifies the status of the certificate to be used to get the certificate.
`SetSerialNumber(serialNumber)`	Specifies the serialNumber of the certificate to be used to get or revoke the certificate.
`setCaAlias(caAlias)`	Specifies the CA Alias of the certificate to be used to get the certificate.
`setStartIndex(startIndex)`	In the get Certificates request, specify the index of the first record to include in the results that will be retrieved.
`setMaxRecords(maxRecords)`	In the get Certificates request, specify the max number of records to include in the results that will be retrieved.
`SetSubjectDirectoryAttributes( subjectDirectoryAttributes)`	It is used to add subject directory attributes extension in X.509 certificate. The parameter specifies Subject Directory Attributes value and this parameter must be a comma separated value. The possible values for are: - `dateOfBirth` - `placeOfBirth` Consider the following for example: `dateOfBirth= 19590927120000Z,placeOfBirth=Pakistan`
`setSeatId(seatId)`	Seat ID refers to seat identification of an authorized end user of the service. e.g `some-seat-id@example.com`
`setDeleteParam(deleteParam)`	It is used to instructs the ADSS Certification Service to delete the private key(PKCS12) associated with the specified certificate stored on the ADSS Server. If the server does not contain the private key referenced by the parameter, an error message will be returned. Supported value for deleteParam is pkcs12 irrespective of the case. **Note:** For security purposes, it is recommended to set the **PKCS12 key password** when submitting a delete request using the SetPkcs12Password method. This password should match the one you specified during the certificate generation or issuance process.

6.4.2 Other Certification Request Methods

Some other Certification Request methods are defined such as those for communication purposes (e.g. use of proxy, timeouts etc.):

`SetProxy, SetRequestID, SetRequestRetries, SetTimeout.`

For these and others refer to the JavaDoc and Sandcastle documentation.

6.4.3 Sending the Certification Request

Once the certification request message has been fully built using the above methods, it can be sent to ADSS Server using the following call:

```
var certificationResponse = (CertificationResponse)certificationRequest.Send(URL);
```

The Certification Service URL for **XML** protocol e.g.

http://machine-name:8777/adss/certification/csi

For a mutually authenticated TLS request, it is:

https://machine-name:8779/adss/certification/csi

The Certification Service URL for **CMC** protocol e.g.

https://machine-name:8779/adss/certification/cmc

The Certification Service URL for **EST** protocol e.g.

http://machine-name:8777/ .well-known/est/simpleenroll

http://machine-name:8777/ .well-known/est/simplereenroll

http://machine-name:8777/ .well-known/est/serverkeygen

https://machine-name:8778/ .well-known/est/cacerts

http://machine-name:8777/ .well-known/est/fullcmc

For a mutually authenticated TLS request, it is:

https://machine-name:8779/ .well-known/est/simpleenroll

https://machine-name:8779/ .well-known/est/simplereenroll

https://machine-name:8779/ .well-known/est/serverkeygen

https://machine-name:8779/ .well-known/est/fullcmc

6.4.4 Example of a Certificate Request (Ascertia XML protocol)

```
// Create Certificate Request
var certifyRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_CREATE_CERTIFICATE, certAlias);

certifyRequest.SetProfileId("adss:certification:profile:001");
certifyRequest.SetRequestMode(CertificationRequest.XML);

certifyRequest.SetPkcs12Password("password");          // Password for P12 token

// Specify 'respond with' items
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_CERTIFICATE);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_12);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_7);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_EXPIRY_DATE);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_10);
// Override policy attributes is required and allowed
certifyRequest.OverrideProfileAttribute(CertificationRequest.SUBJECT_DN, "CN=Test");

// Sending the above constructed request to ADSS server
var certifyResponse = (CertificationResponse)
    certifyRequest.Send("http://localhost:8777/adss/certification/csi");
```

6.4.5 Example of a Certificate Revocation Request (Ascertia XML protocol)

```
// Create Certificate Revocation Request
var certifyRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_REVOKE, certAlias);

certifyRequest.SetProfileId("adss:certification:profile:001");
certifyRequest.SetRequestMode(CertificationRequest.XML);

certifyRequest.SetRevocationReason(CertificationRequest.REVOCATION_REASON_PRIVILEGEWITHDRAWN);

// Sending the above constructed request to ADSS server
var certifyResponse = (CertificationResponse)
    certifyRequest.Send("http://localhost:8777/adss/certification/csi");
```

6.4.6 Example of Certificate Deletion Request (Ascertia XML protocol)

```
// Constructing request for deleting certificate
var certifyRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_DELETE_CERTIFICATE, certAlias);

 // Sending the above constructed request to the ADSS server
var certifyResponse = (CertificationResponse)
    certifyRequest.Send("http://localhost:8777/adss/certification/csi");
```

6.4.7 Example of Certificate Change Password Request (Ascertia XML protocol)

```
 // Constructing request
var certifyRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_CHANGE_PASSWORD, certAlias);

//Old Password and New Password must be provided
certifyRequest.SetPkcs12Password("oldpassword");
certifyRequest.SetPkcs12NewPassword("newpassword");

// Specify 'respond with' items
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_CERTIFICATE);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_12);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_7);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_EXPIRY_DATE);

// Sending the above constructed request to the ADSS server
var certifyResponse = (CertificationResponse)
    certifyRequest.Send("http://localhost:8777/adss/certification/csi");
```

6.4.8 Example of Certificate Recover Key Request (Ascertia XML protocol)

```
// Constructing request
var certifyRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_RECOVER_KEY, certAlias);

// Specify 'respond with' items
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_CERTIFICATE);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_12);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_7);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_EXPIRY_DATE);

// Sending the above constructed request to the ADSS server
var certifyResponse = (CertificationResponse)
    certifyRequest.Send("http://localhost:8777/adss/certification/csi");
```

6.4.9 Example of Renew Certificate Request (Ascertia XML protocol)

```
// Constructing request for renewing certificate
var certifyRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_RENEW_CERTIFICATE, certAlias);

certifyRequest.SetPkcs12Password("password");
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_CERTIFICATE);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_12);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_7);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_EXPIRY_DATE);

// Sending the above constructed request to the ADSS server
var certifyResponse = (CertificationResponse)
    certifyRequest.Send("http://localhost:8777/adss/certification/csi");
```

6.4.10 Example of Rekey Certificate Request (Ascertia XML protocol)

```
// Constructing request
var certifyRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_REKEY_CERTIFICATE, certAlias);

certifyRequest.SetPkcs12Password("password");
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_CERTIFICATE);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_12);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_7);
certifyRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_EXPIRY_DATE);

// Sending the above constructed request to the ADSS server
var certifyResponse = (CertificationResponse)
    certifyRequest.Send("http://localhost:8777/adss/certification/csi");
```

6.4.11 Example of a Profile Info Request (Ascertia XML protocol)

```
//Creating request for Certification Profile Info
CertificationRequest obj_certificationRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_GET_PROFILE_INFO, "adss:certification:profile:001");
obj_certificationRequest.SetRequestID("Get_profile_request01");

// Sending the above constructed request to the ADSS server
CertificationResponse obj_certificationResponse = (CertificationResponse)
    obj_certificationRequest.Send("http://localhost:8777/adss/certification/csi");
```

6.4.12 Example of a Get Certificates Request (Ascertia XML protocol)

User can get a certificate on the basis of Subject DN, Serial Number, User ID, Status or CA Alias, or by using any combination of these parameters. However, it should be noted that the user must provide at least one certificate attribute amongst Subject DN, User ID, Serial Number and CA Alias in order to get the certificate. The user can also use pagination when calling this API by specifying the Start index and Max record parameters.

```
// creating request for Certificate Info
CertificationRequest obj_certificationRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_GET_CERTIFICATES, "23f2a52f401aed0776183089bbded973",
    "CN=Jhon Doe,OU=Development,O=Ascertia,C=GB", "TEST-USER", "ACTIVE",
            "ADSS Samples Test CA", 0, 20);

     obj_certificationRequest.SetRequestID("Get_certificate_request01");
     obj_certificationRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_7);

// Sending the above constructed request to the ADSS server
CertificationResponse obj_certificationResponse = (CertificationResponse)
        obj_certificationRequest.Send("http://localhost:8777/adss/certification/csi");
```

6.4.13 Example of Send SCT (Ascertia XML protocol)

This request type is used in a scenario where issueing CA is using the delegation for Certificate Transparency. The following option against a local ADSS must be enabled for this:

ADSS Console > Manage CAs > Local CAs > Certificate Transparency Settings > Delegate PreCertificate Logging process to other entities.

In this case the CA is normally operating in an internal zone and can't communicate with the configured CT Log Servers over the internet so the CA asks the client applications to communicate with the CT Log Servers and get the SCTs. In this use case, when you send a request to Certification Service to create or renew a certificate, the service responds with a "WAITING" status and provides the pre-certificate, its chain,

and the Log Servers addresses. The client application communicates with the Log Servers to log the pre-certificate and get the SCTs.

Below is the example to communicate with a Log Server

http://<server>/ct/v1/add-pre-chain		
HTTP Verb	POST	
Content-Type	application/json	
Accept	application/json	

Request Body	{
	"chain": ["MIICxDCCAawCAQAwfzELM[....]5f52oQ==",
	"MIICxDCCAawCAQAwfzELM[....]5f52oQ==",
	"MIICxDCCAawCAQAwfzELM[....]5f52oQ=="
	]
	
	}	

Status Code	**Message**	**Response Body**

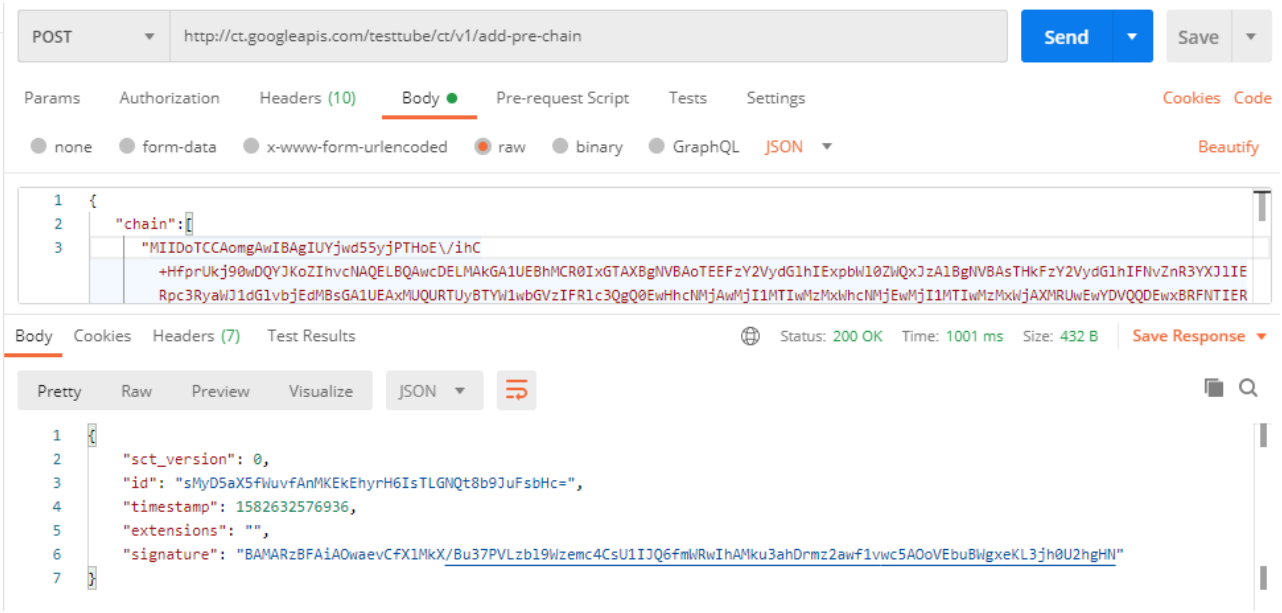
200	OK	{
		"sct_version": 0,
		"id": "o8mYRegKt84AFXs3Qt8CB90nKytgLs+Y7iwS25xa5+c=",
		"timestamp": 1600931634873,
		"extensions": "",
		"signature": "BAMARjBEAiBCM7fivQ="
		}

400	Bad Request	

500	Internal Server Error	

Item Details	
Name	Description
Request Parameters	
chain	Certifiate chain which needs to be submitted to log server
Response Parameters	
Sct_version	Version of SCT e.g v1
id	ID of SCT
timestamps	Time in millisecons
extensions	Extension in SCT
Signature	Signature of SCT

SCT via Postman



Once it has collected the SCTs, the client application will use this request type (Send SCT) to send the SCTs to Certification Service. The Service issues a certificate embedding the SCTs into it and returns the certificate in response.

```
// Constructing request for getting certificate(s) from the ADSS certification service
CertificationRequest obj_certificationRequest = new CertificationRequest("samples_test_client", CertificationRequest.REQUEST_TYPE_ADD_SCT, args[1].Trim());
obj_certificationRequest.SetPkcs12Password(args[2]);
obj_certificationRequest.SetProfileId(args[3]);

String[] arr_sctVersion = args[4].Split(',');
String[] arr_sctId = args[5].Split(',');
String[] arr_sctTimestamp = args[6].Split(',');
String[] arr_sctSignature = args[7].Split(',');
String[] arr_sctServerAddress = args[8].Split(',');
SctInfo obj_sctInfo = null;
for (int i = 0; i < arr_sctId.Length; i++)
{
    obj_sctInfo = new SctInfo();
    obj_sctInfo.StrSctId = arr_sctId[i];
    obj_sctInfo.StrSctVersion = arr_sctVersion[i];
    obj_sctInfo.StrSctServerAddress = arr_sctServerAddress[i];
    obj_sctInfo.StrSctTimestamp = arr_sctTimestamp[i];
    obj_sctInfo.StrSctSignature = arr_sctSignature[i];
    obj_certificationRequest.addSCT(obj_sctInfo);
}

obj_certificationRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_CERTIFICATE);
obj_certificationRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_12);
obj_certificationRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_PKCS_7);
obj_certificationRequest.AddRespondWithItem(CertificationRequest.RESPOND_WITH_EXPIRY_DATE);
```

6.4.14 Example of Certificate Request (CMC Protocol)

The example below shows a certificate request sent with the CMC protocol using HTTPS with mutual authentication. See also section 3.2.1 for a discussion on using SSL/TLS with ADSS Server.

```
// Create CMC Certificate Generation Request
var certifyRequest = new CertificationRequest("samples_test_client",
    CertificationRequest.REQUEST_TYPE_CREATE_CERTIFICATE, certAlias);

certifyRequest.SetProfileId("adss:certification:profile:001");
certifyRequest.SetRequestMode(CertificationRequest.CMC);
certifyRequest.SetCmcRequestMode(CertificationRequest.SIMPLE_PKI_REQUEST);

certifyRequest.SetPKCS10(PKCS10Path);
certifyRequest.SetSslClientCredentials(clientSSLPath,"password");

// Sending the above constructed request to ADSS server
var certifyResponse = (CertificationResponse)
    certifyRequest.Send("https://localhost:8779/adss/certification/csi");
```

6.4.15 Example of Certificate Request Using Simple Enroll API (EST Protocol)

The example below shows a certificate request sent with the EST protocol using HTTPS with mutual authentication. See also section 3.2.1 for a discussion on using SSL/TLS with ADSS Server

This API will be used to issue a new certificate.

Code Example

```
// Create EST Certification Generation Request
var certificateRequest = new CertificationRequest( a_strOriginatorId: "samples_test_client",
       CertificationRequest.REQUEST_TYPE_CREATE_CERTIFICATE, a_strCertAlias: "");
certificateRequest.setProfileId("adss:certification:profile:001");
certificateRequest.setRequestMode(CertificationRequest.EST);
certificateRequest.setPKCS10(PKCS10);
certificateRequest.setClientSecret("762b9a0c421fe4f64f47a6135545d436979450d6");
certificateRequest.setSslClientCredentials(clientSSLPath, a_strPassword: "password");

// sending the above constructed request to ADSS Server
var certificateResponse = (CertificationResponse) certificateRequest.send( a_strUrl: "https://localhost:8779/.well-known/est/simpleenroll");
```

6.4.16 Example of Certificate Renew/Rekey Request Using Simple Re Enroll API (EST Protocol)

The example below shows a certificate request sent with the EST protocol using HTTPS with mutual authentication. See also section 3.2.1 for a discussion on using SSL/TLS with ADSS Server

Following URL we used for simple re enroll API for renew/rekey a certificate using EST protocol.

https://<server>:8779/.well-known/est/simplereenroll

This API will be used to renew/rekey an existing certificate. For both renew/rekey, the client will send us a PKCS#10 in request with the same Subject DN and SubjectAltName that was used in old certificate.

If the PKCS#10 will contain the same old key, it will be a renew request. But if the public key will be different than the old one, it will be treated as a rekey request.

The request/response structure for this API is same as /simpleenroll API defined in section 6.4.12

Code Example

```
// Create EST Certification Generation Request
var certificateRequest = new CertificationRequest( a_strOriginatorId: "samples_test_client",
        CertificationRequest.REQUEST_TYPE_RENEW_CERTIFICATE, a_strCertAlias: "");
certificateRequest.setProfileId("adss:certification:profile:001");
certificateRequest.setRequestMode(CertificationRequest.EST);
certificateRequest.setPKCS10(PKCS10);
certificateRequest.setClientSecret("762b9a0c421fe4f64f47a6135545d436979450d6");
certificateRequest.setSslClientCredentials(clientSSLPath, a_strPassword: "password");

// sending the above constructed request to ADSS Server
var certificateResponse = (CertificationResponse) certificateRequest.send( a_strUrl: "https://localhost:8779/.well-known/est/simplereenroll");
```

Code Example

```
// Create EST Certification Generation Request
var certificateRequest = new CertificationRequest( a_strOriginatorId: "samples_test_client",
        CertificationRequest.REQUEST_TYPE_REKEY_CERTIFICATE, a_strCertAlias: "");
certificateRequest.setProfileId("adss:certification:profile:001");
certificateRequest.setRequestMode(CertificationRequest.EST);
certificateRequest.setPKCS10(PKCS10);
certificateRequest.setClientSecret("762b9a0c421fe4f64f47a6135545d436979450d6");
certificateRequest.setSslClientCredentials(clientSSLPath, a_strPassword: "password");

// sending the above constructed request to ADSS Server
var certificateResponse = (CertificationResponse) certificateRequest.send( a_strUrl: "https://localhost:8779/.well-known/est/simplereenroll");
```

6.4.17 Example of Certificate Request Using Serverkeygen API (EST Protocol)

The example below shows a certificate request sent with the EST protocol using HTTPS with mutual authentication. See also section 3.2.1 for a discussion on using SSL/TLS with ADSS Server.

This API is used to generate a key-pair at server side and then issue a certificate. It also returns the newly generated private key in the response. Since it has to return the private key so this API can only generate software based keys and not hardware.

Code Example

```
// Create EST Certification Generation Request
var certificateRequest = new CertificationRequest( a_strOriginatorId: "samples_test_client",
        CertificationRequest.REQUEST_TYPE_CREATE_CERTIFICATE, a_strCertAlias: "");
certificateRequest.setProfileId("adss:certification:profile:001");
certificateRequest.setRequestMode(CertificationRequest.EST);
certificateRequest.setPKCS10(PKCS10);
certificateRequest.setClientSecret("762b9a0c421fe4f64f47a6135545d436979450d6");
certificateRequest.setSslClientCredentials(clientSSLPath, a_strPassword: "password");

// sending the above constructed request to ADSS Server
var certificateResponse = (CertificationResponse) certificateRequest.send( a_strUrl: "https://localhost:8779/.well-known/est/serverkeygen");
```

6.4.18 Example of Get CA Certificate Request Using CA Cert API (EST Protocol)

This API is used to get CA certificates that client can add to their trust anchors.

Code Example

```
// Create EST Certification Generation Request
var certificateRequest = new CertificationRequest( a_strOriginatorId: "samples_test_client",
        CertificationRequest.REQUEST_TYPE_CACERTS, a_strCertAlias: "");
certificateRequest.setProfileId("adss:certification:profile:001");
certificateRequest.setRequestMode(CertificationRequest.EST);
certificateRequest.setPKCS10(PKCS10);

// sending the above constructed request to ADSS Server
var certificateResponse = (CertificationResponse) certificateRequest.send( a_strUrl: "https://localhost:8778/.well-known/est/cacerts");
```

6.4.19 Example of Certificate Revoke Request Using Full CMC API (EST Protocol)

The example below shows a certificate request sent with the EST protocol using HTTPS with mutual authentication. See also section 3.2.1 for a discussion on using SSL/TLS with ADSS Server

The EST protocol allows clients to send CMC requests using this API. In response, an EST server processes these requests and return CMC responses.

Code Example

```
// Create EST Certification Generation Request
var certificateRequest = new CertificationRequest( a_strOriginatorId: "samples_test_client",
        CertificationRequest.REQUEST_TYPE_REVOKE_CERTIFICATE, a_strCertAlias: "");
certificateRequest.setProfileId("adss:certification:profile:001");
certificateRequest.setRequestMode(CertificationRequest.EST);
certificateRequest.setPKCS10(PKCS10);
certificateRequest.setClientSecret("762b9a0c421fe4f64f47a6135545d436979450d6");
certificateRequest.setSslClientCredentials(clientSSLPath, a_strPassword: "password");

// sending the above constructed request to ADSS Server
var certificateResponse = (CertificationResponse) certificateRequest.send( a_strUrl: "https://localhost:8779/.well-known/est/fullcmc");
```

6.5 Certification Response Class

The following methods of the Certification Response class are inherited from the generic Response and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetRequestID, GetSigningCertificates, GetStatus, IsSuccessful.`

In addition, the following methods are specific to the Certification Response Class:

Certificate Response Method	Purpose
`GetCertificate() returns X509Certificate`	Returns the X509 certificate object.
`GetExpiryDate() returns DateTime`	Returns the expiry date of the certificate.
`GetPKCS12() returns byte[]`	Returns the PKCS#12 file.
`GetPKCS10() returns byte[]`	Returns the PKCS#10 file.
`GetPkcs12Password() returns string`	Returns the PKCS#12 password.
`GetPKCS7() returns string`	Returns the PKCS#7 certificate chain.
`GetProfileId() returns string`	Returns the Certification Profile ID used by the Certification Service to process the request.
`PublishCertificate(string /Stream)`	Publishes the certificate to the specified path or stream.
`PublishPKCS12(string /Stream)`	Publishes the PKCS#12 data to the specified path or stream.

PublishPKCS7(string /Stream)	Publishes the PKCS#7 data to the specified path or stream.
PublishPKCS10(string /Stream)	Publishes the PKCS#10 data to the specified path or stream.
publishPKCS8PrivateKey(string /Stream)	Publishes the PKCS8 data to the specified path or stream.
GetRequestId() returns string	Returns the Request ID of the certification request.
GetProfileInfo() returns ProfileInfoType	Returns the ProfileInfoType object.
getPrecertificate() returns PreCertificateLogInfo	Returns the PreCertificateLogInfo in case of 'Delegate the Precertificate logging process to other entities' check is enabled in Mange CA -> Local CA settings
GetCertificateInfo() returns CertificateInfoListType	Returns the CertificateInfoListType object.

6.6 Certification Service Sample Code

Java and .Net sample code is provided as part of the ADSS Client SDK and can be used to make Certification Service requests and to process the Certification Service responses.

The Java API provides the required classes under the package:

```
com.ascertia.adss.client.api.certification
```

The .Net API provides the required classes under the namespace:

```
Com.Ascertia.ADSS.Client.API.Certification
```

6.6.1 Java API Sample Code

The following sample programs demonstrates how the Java API can be used to send a Certification request and process the response:

```
samples/src/com/ascertia/adss/client/samples/certification/AddSct.java
samples/src/com/ascertia/adss/client/samples/certification/GenerateCertificate.java
samples/src/com/ascertia/adss/client/samples/certification/RevokeCertificate.java
samples/src/com/ascertia/adss/client/samples/certification/DeleteCertificate.java
samples/src/com/ascertia/adss/client/samples/certification/RenewCertificate.java
samples/src/com/ascertia/adss/client/samples/certification/RevokeCertificateWithSerialNumber.java
samples/src/com/ascertia/adss/client/samples/certification/RecoverKey.java
samples/src/com/ascertia/adss/client/samples/certification/ChangePassword.java
samples/src/com/ascertia/adss/client/samples/certification/
GetCertificationProfileInfo.java
samples/src/com/ascertia/adss/client/samples/certification/
GetCertificates.java
```

A precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/AddSct.bat
samples/bin/GenerateCertificate.bat
```

```
samples/bin/RevokeCertificate.bat
samples/bin/RevokeCertificateWithSerialNumber.bat
samples/bin/DeleteCertificate.bat
samples/bin/RenewCertificate.bat
samples/bin/RecoverKey.bat
samples/bin/ChangePassword.bat
samples/bin/CertificateProfileInfo.bat
samples/bin/GetCertificates.bat
```

6.6.2 .Net API Sample Code

The following sample programs demonstrates how the .Net API can be used to send a Certification Service request and process the response:

```
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/AddSCT.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/GenerateCertificate.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/RevokeCertificate.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/
RevokeCertificateWithSerialNumber.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/DeleteCertificate.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/RenewCertificate.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/RecoverKey.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/ChangePassword.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/GetCertificationProfileInfo.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/Certification/GetCertificates.cs
```

A precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/AddSCT.bat
samples/bin/GenerateCertificate.bat
samples/bin/RevokeCertificate.bat
samples/bin/RevokeCertificateWithSerialNumber.bat
samples/bin/DeleteCertificate.bat
samples/bin/RenewCertificate.bat
samples/bin/RecoverKey.bat
samples/bin/ChangePassword.bat
samples/bin/CertificateProfileInfo.bat
samples/bin/GetCertificates.bat
```

6.7 ADSS Certification Service Supported Algorithms

The following is a list of signing/hashing algorithms and key lengths that ADSS Certification Service supports:

ADSS Service	Signature Algorithm	Hashing Algorithm	Algorithm / Key Sizes
Certification using LOCAL CA	SHA1WithRSAEncryption SHA224WithRSAEncryption SHA256WithRSAEncryption SHA384WithRSAEncryption SHA512WithRSAEncryption RipeMD128WithRSAEncryption	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 RipeMD128 RipeMD160	Below are the list of Key Algorithms and Sizes • RSA: 1024, 2048, 3072, 4096, 8192 • ECDSA Curves: ○ NIST

	RipeMD160WithRSAEncryption SHA1withECDSA SHA224withECDSA SHA256withECDSA SHA384withECDSA SHA512withECDSA		P-160, P-192, P-224, P-256, P-384,P-521 ○ SEC secp256k1 ○ TeleTrust (Brainpool) brainpoolp160r1, brainpoolp160t1, brainpoolp192r1, brainpoolp192t1, brainpoolp224r1, brainpoolp224t1, brainpoolp256r1, brainpoolp256t1, brainpoolp320r1, brainpoolp320t1, brainpoolp384r1, brainpoolp384t1, brainpoolp512r1, brainpoolp512t1

6.8 Error Codes

ADSS Certification Service returns the following statuses in case of any failure:

Error Code	Error Message
43016	ADSS Certification service is not enabled in license.
43014	ADSS Certification service license is expired.
43053	ADSS Certification service is not enabled in the system.
43015	ADSS Certification service is stopped.
43041	Manage CAs module is not enabled in license.
43019	Certification request must be signed.
43020	Failed to verify certification request signature.
43021	Request does not comply with Ascertia certification XML schema.
43002	Certificate alias is missing in the request.
43011	Requested certificate alias already exists.
43009	Internal error occurred during processing of the request.
43001	Certification profile attributes not found.
43049	CA configured in the requested profile is inactive.
43054	Key size is not supported.
43027	PKCS#10 does not comply with the certification profile.
43030	Failed to insert certificate in database.
43005	PKCS#10 signature verification failed.
43063	To be issued certificate expiry date/time is beyond the CA Certificate expiry date/time.
43035	Request does not contain PKCS#12/PFX password.
43017	Valid subject DN could not be composed.
43513	Requested certificate does not comply with CA's Extended Key Usages.

43512	Requested certificate does not comply with CA's Basic Constraints.
43511	Requested certificate does not comply with CA's Name Constraints.
43514	Requested certificate does not comply with CA's Certificate Policies.
43003	Request contains insufficient information.
43006	Certificate alias does not exist.
43007	Certificate does not belong to the specified client.
43012	Failed to delete certificate.
43052	Certificate with status NotYetValid cannot be deleted.
43008	Active certificate cannot be deleted.
43026	Revoked certificate cannot be renewed.
43029	Failed to renew certificate, old key pair does not exist.
43034	PKCS#12/PFX not found in the database.
43004	Request contains invalid old password for PKCS#12/PFX.
43038	Failed to process request - Issuer DN in request does not match with issuer DN found in the certificate
43023	Certificate is already revoked.
43031	Certificate is already on hold.
43032	Certificate is not revoked.
43039	Expired certificate cannot be revoked.
43051	Certificate with status NotYetValid cannot be revoked.
43033	Invalid revocation reason or hold instruction code.
43018	Failed to revoke certificate.
43024	Failed to change PKCS#12/PFX password.
43061	Invalid old password in the request.
43062	New password not available in the request.
43040	Failed to verify PKCS#10.
43047	Invalid PKCS#10.
43048	PKCS#10 is missing in the request.
43059	Subject DN is missing in the request.
43050	ADSS Certification service is not allowed for this client.
43055	Certification profile is inactive.
43056	Certification profile does not exist.
43042	Certification profile in request is not allowed to the client.
43043	TLS client authentication certificate does not match with the configured client authentication certificate.
43515	Failed to process request - the signer certificate is in pending state.
43570	Failed to process request - PreIssuedCertificate not found.

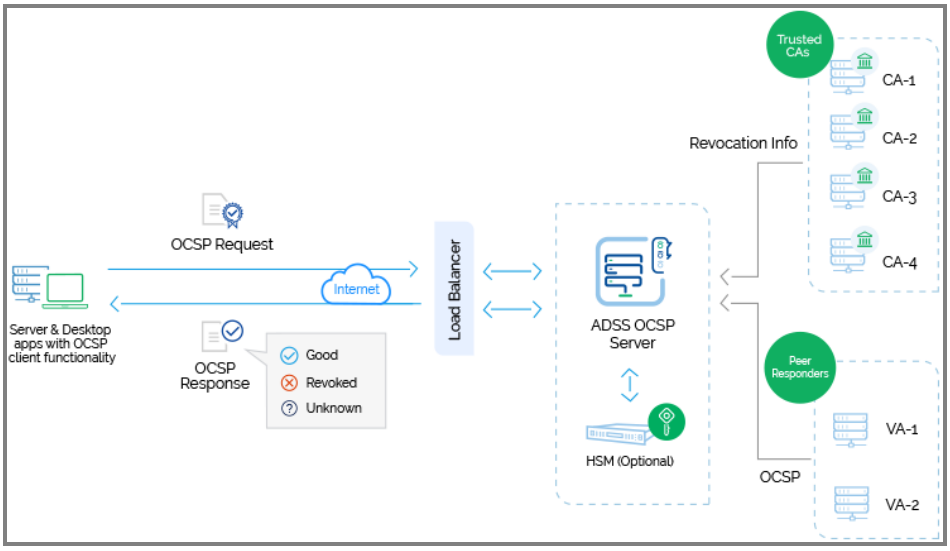
43081	Failed to process request - Basic Authorization Header is missing/invalid
43082	Failed to process request - Content-Type is missing/invalid
43083	Failed to process request - invalid or unsupported EST request type
43084	Failed to process request - Crypto profile must be Software
43085	Failed to process request - SAN in PKCS#10 doesnot match with SAN defined in Issued Certificate
43086	Failed to process request - Subject DN in PKCS#10 doesnot match with Subject DN defined in Issued Certificate
43087	Failed to process request - Multiple certificates with the same Subject DN
43088	Failed to process request - Certificate already exists with the same Subject DN
43089	Failed to process request - Certificate Primary Key structure does not generated
43090	Failed to process request - No CA is selected as Default
43091	Failed to process request - Client ID in authorization header does not match with client configured for TLS Client certificate.
43092	Failed to process request - SImple PKI Request is not allowed
43093	Failed to process request - Certificate serial number does not exist
43098	Failed to process request - Certificate serial number does not exist or it does not match the IssuerDN in the request
43099	Failed to process request - Multiple certificates associated with the given serial number. To facilitate a more targeted search include the issuerDN in the request
43500	Failed to process request – Certificate does not exist with given information
43572	Failed to process request – One value should be present from these values: Subject DN, User ID and Serial Number

7 ADSS OCSP Service

The ADSS Server OCSP Service provides an RFC 6960/FIPS 201 and RFC 8954 compliant real-time digital certificate OCSP validation authority service.

There are two different ways in which the ADSS OCSP Service can be utilised to produce OCSP validation responses:

- As the definitive certificate status responder for a particular CA. The CA can be internal to ADSS Server or external.
- Forwarding OCSP Requests to a peer OCSP Responder. In this scenario, the ADSS OCSP Service may forward requests to multiple OCSP Responders depending upon the issuing CAs involved. An example of this could be a banking network.



7.1 Setting up ADSS OCSP Service Profiles

The ADSS OCSP Service requires that OCSP Profiles are defined at ADSS Server. These enable separate policies for each CA for which the OCSP Service is responding.

Refer to the following online admin guide for an explanation of OCSP Profile settings:

Configuring the OCSP Service (ascertia.com)

7.2 The ADSS OCSP Service API

In order to simplify the use of the RFC 6960 and RFC 8954 OCSP protocol, an OCSP Service API is provided as part of the ADSS Client SDK.

The API consists of the following classes:

- OCSP Request
- OCSP Response

7.3 OCSP Request Class

7.3.1 OCSP Request Constructor

The OCSP Request Class has two constructors depending upon whether the input parameters are supplied as file paths or the actual certificates. Both the certificate to be validated and the issuer certificate are required:

```
var ocspRequest = new OcspRequest(x509CertificateToValidate,
x509IssuerCertificate)
```

7.3.2 OCSP Request Methods

The OCSP Request Class (OcspRequest) inherit a number of methods from the generic Request and Message classes which are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

```
ToString, WriteTo, Send (overridden), SetProxy, SetRequestID,
SetRequestRetries, SetSigningCredentials (overridden),
SetSSLClientCredentials, SetTimeout, SetVerifyResponse.
```

In addition, the following methods are specific to the OCSP Request Class:

OCSP Request method	Purpose
`AddCertificate (string, string) or AddCertificate (X509Certificate, X509Certificate)`	Add an additional certificate to be validated. Both the certificate to be validated and the issuer certificate are required.
`SetNonce(long nonce)`	Specifies a random 'nonce' value to require the OCSP to give a fresh response. As per RFC 8954, the length of the nonce MUST be at least 1 octet and can be up to 32 octets
`SetServiceLocator(bool serviceLocator)`	Sets the service locator flag. This specifies that the request is to be routed to the authoritative OCSP Responder as specified in the target certificate.
`AddPreferredSignatureAlgo( String preferredSignatureAlgo)`	Add preferred signature algorithms in the request as per RFC 6960 to sign the response.
`SetHashAlgo(string a_strHashAlgo)`	It specifies which hash algorithm is used for CertID. Default hash algorithm is SHA256.

7.3.3 Sending the OCSP Request

Once the OCSP request message has been prepared, it is sent to ADSS Server using the following method call:

```
var ocspResponse = (OcspResponse)ocspRequest.Send(ocspServiceAddress);
```

The service address `URL` is that of the OCSP Service e.g. http://machine-name:8777/adss/ocsp

Note that on receiving the response the `GetCertStatus()` method should be called to confirm that the request has been successful and certificate status is returned – see section 7.4 below.

7.3.4 Example of creating and sending an OCSP Request

```
// Specify certificate to validate and issuer certificate
var ocspRequest = new OcspRequest(certToValidate, issuerCert);

Random d_random = new Random();
ASCIIEncoding encoding = new ASCIIEncoding();
ocspRequest.SetNonce(encoding.GetBytes(d_random.Next().ToString()));

// Send OCSP request
var ocspResponse = (OcspResponse)ocspRequest.Send(ocspServiceAddress);
```

7.4 OCSP Response Class

The OCSP Response class (OcspResponse) inherits the following methods from the Response and Message classes. There are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

```
ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage,
GetException, GetSigningCertificates, GetStatus, IsSuccessful.
```

In addition, the following methods are specific to the class:

OCSP Response Method	Purpose

`GetCertStatus() returns int`
`or if multiple certificates were processed`
`GetCertStatus(X509Certificate) returns int`
	Returns the certificate status for the target certificate. The returned integer value is one of:
	`OcspResponse.GOOD`
	`OcspResponse.REVOKED, or`
	`OcspResponse.UNKNOWN`

`GetNextUpdate()returns DateTime`
`or if multiple certificates were processed`
`GetNextUpdate(X509Certificate) returns DateTime`
	Returns the “nextUpdate” value of the relevant CRL.

`GetNonce() returns byte[]`	Returns the nonce value of the OCSP response. This is used to ensure the OCSP Service /Responder has provided a fresh response. As per RFC 8954 If the Nonce extension value in the request will be less than 16 octets, then nonce will not be available in the response
`GetOcspResponse() returns OcspResponse`	Returns the OCSP response from OCSP Service.
`GetThisUpdate() returns DateTime or if multiple certificates were processed GetThisUpdate(X509Certificate)`	It returns the "thisUpdate" of the relevant CRL.
`GetProducedAt() returns DateTime`	It returns the OCSP response producedAt time.

7.4.1 Example of processing an OCSP Service Response

```
// Send OCSP request
var ocspResponse = (OcspResponse)ocspRequest.Send(ocspServiceAddress);

int i_certStatus = ocspResponse.GetCertStatus();
string certStatus;
DateTime thisUpdate;
DateTime nextUpdate;
DateTime producedAt;
OcspResp response;

// Check response status
//
if (ocspResponse.IsSuccessful())
{
    // Get Certificate Status
    if (i_certStatus == OcspResponse.GOOD)
        certStatus = "GOOD";
    else if (i_certStatus == OcspResponse.REVOKED)
        certStatus = "REVOKED";
    else if (i_certStatus == OcspResponse.UNKNOWN)
        certStatus = "UNKNOWN";

    thisUpdate = ocspResponse.GetThisUpdate();
    nextUpdate = ocspResponse.GetNextUpdate();
    producedAt = ocspResponse.GetProducedAt();
    response = ocspResponse.GetOcspResponse();
}
```

7.5 OCSP Service Sample Code

Java and .Net sample code is provided as part of the ADSS Client SDK and can be used to make OCSP Service requests and to process the responses.

The Java API provides the required classes under the package:

`com.ascertia.adss.client.api.ocsp`

The .Net API provides the required classes under namespace:

`Com.Ascertia.ADSS.Client.API.OCSP.`

7.5.1 Java API Sample Code

The following sample programs demonstrate how the Java API can be used to send an OCSP validation request to the OCSP Service and to process the response:

`samples/src/com/ascertia/adss/client/samples/ocsp/OcspRequest.java`

Precompiled and ready to run version of the above sample programs can be found at:

`samples/bin/OcspValidate.bat`

7.5.2 .Net API Sample Code

The following sample programs demonstrate how the .Net API can be used to send an OCSP validation request to the OCSP Service and to process the response:

`samples/src/Com/Ascertia/ADSS/Client/Samples/OCSP/OcspRequest.cs`

A precompiled and ready to run version of the above sample program can be found at:

`samples/bin/OcspValidate.bat.`

7.6 ADSS OCSP Service Supported Algorithms

The following is a list of signing/hashing algorithms and key lengths that ADSS OCSP Service supports:

ADSS Service	Signature Algorithm	Hashing Algorithm	Algorithm / Key Sizes
OCSP	SHA1WithRSAEncryption SHA224WithRSAEncryption SHA256WithRSAEncryption SHA384WithRSAEncryption SHA512WithRSAEncryption SHA3-224WithRSAEncryption SHA3-256WithRSAEncryption SHA3-384WithRSAEncryption SHA3-512WithRSAEncryption RipeMD128WithRSAEncryption RipeMD160WithRSAEncryption SHA1withECDSA SHA224withECDSA SHA256withECDSA SHA384withECDSA SHA512withECDSA	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 SHA3-224 SHA3-256 SHA3-384 SHA3-512 RipeMD128 RipeMD160	RSA: 1024, 2048, 3072, 4096 ECDSA: 192,224,256,384,521

7.7 Error Codes

ADSS OCSP Service returns the following statuses in case of any failure:

OCSP Response Status	OCSP Single Response Status	Description
tryLater	N/A	ADSS OCSP Server is stopped.
HTTP 400 – Bad Request	N/A	Invalid HTTP POST request type. Valid value is “application/ocsp-request”.
malformedRequest	N/A	Invalid OCSP request format.
malformedRequest	N/A	Invalid Nonce value – As per RFC 8954 If the Nonce extension is present, then the length of the nonce MUST be at least 1 octet and can be up to 32 octets
unauthorized	N/A	ADSS OCSP Server license is expired.
unauthorized	N/A	ADSS OCSP Server disabled in license.
unauthorized	N/A	TLS client certificate status is revoked.
unauthorized	N/A	TLS client certificate status is indeterminate.
unauthorized	N/A	ADSS OCSP Service access control check failed.
unauthorized	N/A	Number of certIDs in OCSP request greater than the configured certIDs count.
malformedRequest	N/A	No certID present in the OCSP request.
sigRequired	N/A	ADSS OCSP Server expected signed OCSP request from client.
unauthorized	N/A	OCSP request signing certificate contains unsupported critical extensions.

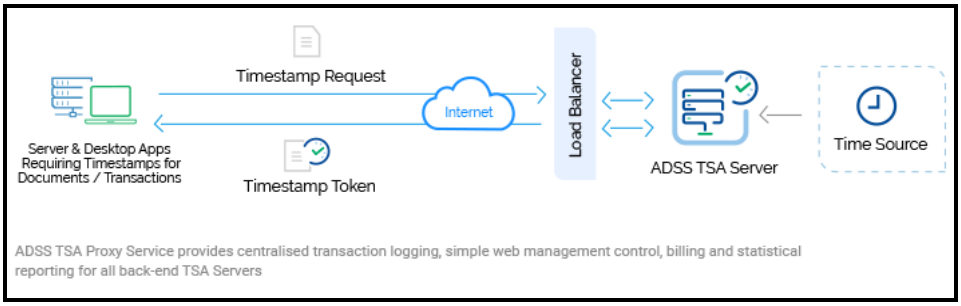
unauthorized	N/A	OCSP request signing certificate does not contain digital signature and non-repudiation KU extension.
unauthorized	N/A	OCSP request signing certificate is expired.
unauthorized	N/A	OCSP request signing certificate is not yet valid.
unauthorized	N/A	OCSP request signature verification failed.
unauthorized	N/A	OCSP request signing certificate is not trusted in ADSS OCSP Server.
unauthorized	N/A	OCSP request signing certificate status is revoked.
unauthorized	N/A	OCSP request signing certificate status is indeterminate.
internalError	N/A	If OCSP request is for foreign CA and OCSP request is unsigned and also default OCSP policy is not configured.
unauthorized	N/A	OCSP request does not contain nonce extension.
successful	revoked	CA status is revoked in ADSS OCSP Server.
successful	unknown	CA status is inactive in ADSS OCSP Server.
successful	revoked	Target certificate status is revoked.
successful	unknown	Target certificate status is unknown.
unauthorized	N/A	Request forwarding is disabled in the selected policy and STATUS_FOR_NON_REGISTERED_CA is set to “unauthorized” in ADSS OCSP Server.
successful	unknown	Request forwarding is disabled in the selected policy and STATUS_FOR_NON_REGISTERED_CA not set to “unauthorized” in ADSS OCSP Server.
trylater	N/A	Unable to communicate with peer OCSP responder.
successful	unknown	Request forwarding is enabled and OCSP server address not available either in service locator extension or manually configured OCSP address.
unauthorized	N/A	Preferred OCSP response signature algorithm set in OCSP request by client is not supported by OCSP server.
internalError	N/A	Internal error in ADSS OCSP Server

8 ADSS TSA Service

The ADSS Server Timestamp Authority (TSA) Service complies with the RFC3161 specifications and its purpose is to produce secure cryptographic timestamp tokens for any type of document, digital signature or data, to prove the existence of the data item at a specific date and time. Additionally, the ADSS TSA Server complies with ISO/IEC guidelines for generating (only Digital signatures using SignedData protection method) and verifying timestamp tokens.

There are two different ways in which the ADSS TSA Service can be utilised to produce timestamp tokens:

- Using the TSA Service local TSA keys.
- Forwarding timestamp requests to an external TSA (in this case the TSA Service acts as a concentrator for timestamp requests, which are being serviced by one or more back-end TSAs).



8.1 Setting up ADSS TSA Profiles

The ADSS TSA Service requires that TSA Profiles are available for the ADSS TSA Service. These profiles define the format and characteristics of the timestamp tokens produced.

Refer to the following online admin guide for an explanation of TSA Profile settings:

Step 2 - Configuring TSA Profile (ascertia.com)

8.2 The ADSS TSA Service API

To facilitate the use of the RFC3161 TSA protocol and also compliance with ISO/IEC guidelines, the ADSS Client SDK provides a TSA Service API.

The API consists of the following classes:

- TSP Request
- TSP Response
- TspVerifyRequest
- TspVerifyResponse

8.3 TSP Request Class

8.3.1 Timestamp Request Constructor

The Timestamp Request Class has a single constructor which is used to supply the data to be timestamped:

```
var timestampRequest = new TspRequest(byteData);
```

8.3.2 Timestamp Request Methods

The TSP Request Class (TspRequest) inherit a number of methods from the generic Request and Message classes described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

```
ToString, WriteTo, Send (overridden), SetProxy, SetRequestRetries,
SetSSLClientCredentials, SetTimeout, SetVerifyResponse.
```

In addition, the following methods are specific to the TSP Request Class:

Timestamp Request method	Purpose
`SetDigestAlgorithm (string digestAlgorithm)`	Specifies the algorithm to be used for hashing/digesting the data to be timestamped.
`SetNonce(long nonce)`	Specifies a random 'nonce' value to require the TSA Service to give a fresh response.
`SetPolicyId(string policy)`	Specifies the TSA policy Id.
`SetRequestCertificate(bool flag)`	Flag to request the TSA certificate is sent in the response. Setting this flag to 'true' explicitly requires you to also call the method SetVerifyResponse(true).
`SetComputeMessageImprint(bool flag)`	Flag either to calculate the message imprint of the data or not.
`AddRequestExtensions(String strOid, String strValue)`	Add the String value to request extensions with specified OID
`AddRequestExtensions(String strOid, Date dValue)`	Add the Date value to request extensions with specified OID
`AddRequestExtensions(String strOid, ASN1Encodable objValue)`	Add the ASN1Encodable value to request extensions with specified OID
`AddMessageImprintToExtHash(String strHashAlgo,byte[] byteContentTBT, boolean bComputeMessageImprint)`	Add the hashes to set in tsp-ext-hash Extension as per ISO/IEC 18014-1 with extension OID 1.0.18014.1.1
`AddProtectionMethodsToExtMethod(String [] methodOIDs)`	Add the protection methods in tsp-ext-meth Extension as per ISO/IEC 18014-1 with extension OID 1.0.18014.1.2
`AddTimeStampTokenToRenew(byte[] bTimeStampTokenToRenew)`	Add the timestamp token bytes in tsp-ext-renewal extension with OID 1.0.18014.1.3 to renew the token in accordance with ISO/IEC 18014-1

8.3.3 Sending the Timestamp Request

Once the timestamp request message has been constructed and fully populated, it is sent to ADSS Server using the following method call:

```
var timestampResponse = (TspResponse)timestampRequest.Send (URL);
```

where `URL` is that of the TSA Service e.g. http://machine-name:8777/adss/tsa

Note that on receiving the response the `GetPkiStatus()` method should be called to confirm that the request has been successful and a timestamp token returned – see section 8.4 below.

8.3.4 Example of creating and sending a Timestamp Request

```
// Construct Timestamp request
var encoding = new ASCIIEncoding();
var timestampRequest = new TspRequest(encoding.GetBytes(timestampData));

var random = new Random();
timestampRequest.SetNonce(random.Next());

// Send the TSP request to the TSA Service
var timestampResponse = (TspResponse)timestampRequest.Send(serviceAddress);
```

8.3.5 Example of creating and sending a Timestamp Request Compliance with ISO/IEC

```
// Construct Timestamp request
ASCIIEncoding encoding = new ASCIIEncoding();
var timestampRequest = new TspRequest(encoding.GetBytes(timeStampedData));
var random = new Random();
timestampRequest.SetNonce(random.Next());

// Add ExtHash extension
timestampRequest.AddMessageImprintToExtHash("SHA512", encoding.GetBytes(timeStampedData), true);
timestampRequest.AddMessageImprintToExtHash("SHA3-384", encoding.GetBytes(timeStampedData), true);
timestampRequest.AddMessageImprintToExtHash("RipeMd128", encoding.GetBytes(timeStampedData), true);

// Add ExtMethod extension
timestampRequest.AddProtectionMethodsToExtMethod(new string[] { "1.0.18014.2.1" });

// Send the TSP requst to the TSA service
var timestampResponse = (TspResponse) timestampRequest.Send(serviceAddress);
```

8.3.6 Example of creating and sending a Timestamp Renew Request Compliance with ISO/IEC

```
// Construct Timestamp request
ASCIIEncoding encoding = new ASCIIEncoding();
var timestampRequest = new TspRequest(encoding.GetBytes(timeStampedData));
var random = new Random();
timestampRequest.SetNonce(random.Next());

// Add ExtHash extension
timestampRequest.AddMessageImprintToExtHash("SHA512", encoding.GetBytes(timeStampedData), true);
timestampRequest.AddMessageImprintToExtHash("SHA3-384", encoding.GetBytes(timeStampedData), true);
timestampRequest.AddMessageImprintToExtHash("RipeMd128", encoding.GetBytes(timeStampedData), true);

// Add ExtMethod extension
timestampRequest.AddProtectionMethodsToExtMethod(new string[] { "1.0.18014.2.1" });

// Add timestamp token to renew
timestampRequest.AddTimeStampTokenToRenew(encoding.GetBytes(timestampeTokenToRenew));

// Send the TSP requst to the TSA service
var timestampResponse = (TspResponse) timestampRequest.Send(serviceAddress);
```

8.4 Timestamp Response Class

The Timestamp Response class (TspResponse) inherits the following methods from the Response and Message classes. There are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetSigningCertificates, GetStatus, IsSuccessful.`

In addition, the following methods are specific to the class:

Timestamp Response Method	Purpose
`GetPkiStatus() returns int`	Returns the status of TSP response with these possible integer values: - `GRANTED` - `GRANTED_WITH_MODS` - `REJECTION` - `REVOCATION_NOTIFICATION` - `REVOCATION_WARNING` - `WAITING`. A value or either `GRANTED` or `GRANTED_WITH_MODS` means that a Timestamp token is provided as part of the response. Any other response indicates that no token is present.
`GetNonce() returns long`	Returns the nonce value of the TSP response. This is used to ensure the TSA Service is providing a fresh response.
`GetTimestampToken() returns TimeStampToken`	Returns the timestamp token object. (Note the `org.Bouncycastle.Tsp.TimeStampToken` is provided as a class by the `itextsharp.dll` library which is therefore required as a reference in the calling application).

8.4.1 Example of processing a TSA Service Response

```
int i_pkiStatus = timestampResponse.GetPkiStatus();
string pkiStatus = "";
TimeStampToken tspToken = null;

// Check TSA Service response status
//
if (timestampResponse.IsSuccessful())
{
    // Status of "GRANTED" or "GRANTED_WITH_MODS" means that a TimeStampToken was returned
    if (i_pkiStatus == TspResponse.GRANTED)
        {pkiStatus = "GRANTED";
        tspToken = timestampResponse.GetTimestampToken();}
    else if (i_pkiStatus == TspResponse.GRANTED_WITH_MODS)
        {pkiStatus = "GRANTED WITH MODS";
        tspToken = timestampResponse.GetTimestampToken();}

    // No TimeStampToken returned
    else if (i_pkiStatus == TspResponse.REJECTION)
        pkiStatus = "REJECTION";
    else if (i_pkiStatus == TspResponse.REVOCATION_NOTIFICATION)
        pkiStatus = "REVOCATION_NOTIFICATION";
    else if (i_pkiStatus == TspResponse.REVOCATION_WARNING)
        pkiStatus = "REVOCATION_WARNING";
    else if (i_pkiStatus == TspResponse.WAITING)
        pkiStatus = "WAITING";
}
```

8.5 Timestamp Verify Request Class

8.5.1 Timestamp verify Request Constructor

The Timestamp Verify Request Class has a single constructor which is used to supply the timestamp token to be verified:

```
var timestampVerifyRequest = new TspVerifyRequest(bytesTimestampToken);
```

8.5.2 Timestamp Verify Request Methods

The TSP verify Request Class (TspVerifyRequest) inherit a number of methods from the generic Request and Message classes described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send (overridden), SetProxy, SetRequestRetries, SetSSLClientCredentials, SetTimeout.`

In addition, the following methods are specific to the TSP Verify Request Class:

Timestamp Request method	Purpose
`SetRequestId(String strRequestId)`	Specifies the request Id
`SetDataContent(byte[] byteDataContent)`	Specifies the original data that the time-stamped token was based on. This data will be included in the extension `tsp-ext-data` with the OID 1.0.18014.1.0. It will be utilized to verify the message imprint of the ExtHash extension.

8.5.3 Sending the Timestamp Verify Request

Once the timestamp verify request has been constructed and fully populated the requied fileds, it is sent to ADSS Server using the following method call:

```
var timestampVerifyResponse = (TspVerifyResponse)
timestampVerifyRequest.Send (URL);
```

where `URL` is that of the TSA Service e.g. http://machine-name:8777/adss/tsa

Note that on receiving the response the `GetPkiStatus()` method should be called to confirm that the request has been successful – see section 8.4 below.

8.5.4 Example of creating and sending a Timestamp Verify Request

```
// Construct Timestamp verification request
var tspVerifyRequest = new TspVerifyRequest(bytesTimestampToken);

// set data content to verify ExtHash extension value
tspVerifyRequest.SetDataContent(originalDataContentBytes);

// Set request Id
tspVerifyRequest.SetRequestId("request-1");

// Send the Timestamp verification requst to the TSA service
var objTspVerifyResponse = (TspVerifyResponse)tspVerifyRequest.Send(serviceAddress);
```

8.6 Timestamp Verify Response Class

The Timestamp verificatopn Response class (TspVerifyResponse) inherits the following methods from the Response and Message classes. There are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetSigningCertificates, GetStatus, IsSuccessful.`

In addition, the following methods are specific to the class:

Timestamp Response Method	Purpose
`GetPkiStatus() returns int`	Returns the status of TSP verify response with these possible integer values: - `GRANTED` - `REJECTION` A value or either `GRANTED` means that a Timestamp token is verified successfull. Any other response indicates that token is invalid.
`GetRequestId () returns String`	Returns the request id value of the TSP verify response.
`GetTimeStampToken () returns TimeStampToken`	Returns the same timestamp token object that was included in the request.

8.6.1 Example of processing a TSA Verify Response

```
int i_pkiStatus = objTspVerifyResponse.GetPkiStatus();
string pkiStatus = "";
string requestId = "";

// check response status
if (objTspVerifyResponse.IsSuccessful())
{
    // get request id from response
    requestId = objTspVerifyResponse.GetRequestId();

    // status of granted means timestamp tpken verified successfully
    if (objTspVerifyResponse.GetPkiStatus() == TspVerifyResponse.GRANTED)
    {
        pkiStatus = "GRANTED";
    }
    else if (objTspVerifyResponse.GetPkiStatus() == TspVerifyResponse.REJECTION)
    {
        pkiStatus = "REJECTION";
    }
}
```

8.7 TSA Service Sample Code

Java and .Net sample code is provided as part of the ADSS Client SDK and can be used to make TSA Service requests and to process the responses.

The Java API provides the required classes under the package:

`com.ascertia.adss.client.api.tsa`

The .NET API provides the required classes under the namespace:

`Com.Ascertia.ADSS.Client.API.TSA`

8.7.1 Java API Sample Code

The following sample programs illustrate how to use the Java API to send a timestamp creation and verification request to the TSA service, as well as how to process the response:

```
samples/src/com/ascertia/adss/client/samples/tsa/TsaRequest.java
samples/src/com/ascertia/adss/client/samples/tsa/TsaVerifyRequest.java
```

Precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/TsaRequest.bat
samples/bin/ISOIEC18014TsaRequest.bat
samples/bin/ISOIEC18014TsaRenewRequest.bat
samples/bin/ISOIEC18014TsaVerifyRequest.bat
```

8.7.2 .Net API Sample Code

The following sample programs demonstrate how the .NET API can be used to send a timestamp creation and verification request to the TSA service, as well as how to handle the response:

```
samples/src/Com/Ascertia/ADSS/Client/Samples/TSA/TsaRequest.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/TSA/TsaVerifyRequest.cs
```

Precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/TsaRequest.bat
samples/bin/ISOIEC18014TsaRequest.bat
samples/bin/ISOIEC18014TsaRenewRequest.bat
samples/bin/ISOIEC18014TsaVerifyRequest.bat
```

8.8 ADSS TSA Service Supported Algorithms

The following is a list of signing/hashing algorithms and key lengths that ADSS TSA Service supports:

ADSS Service	Signature Algorithm	Hashing Algorithm	Algorithm / Key Sizes
TSA	SHA1WithRSAEncryption SHA256WithRSAEncryption SHA384WithRSAEncryption SHA512WithRSAEncryption RipeMD128WithRSAEncryption RipeMD160WithRSAEncryption SHA1withECDSA SHA224withECDSA SHA256withECDSA SHA384withECDSA SHA512withECDSA	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 RipeMD128 RipeMD160 SHA3-224 SHA3-256 SHA3-384 SHA3-512	RSA: 1024, 2048, 3072, 4096 ECDSA: 192,224,256,384,521

8.9 Error Codes

ADSS TSA Service returns the following statuses in case of any failure:

PKI Status	PKI FailureInfo	PKI FreeText
rejection	N/A	ADSS TSA Server disabled in license.
rejection	N/A	ADSS TSA Server is stopped.
HTTP 400 – Bad Request	N/A	Invalid TSA request format.
HTTP 403 – Forbidden	N/A	ADSS TSA Server license is expired.
revocationNotification	N/A	Client application TLS certificate status is revoked.
rejection	N/A	Client application TLS certificate status is unknown.
HTTP 403 – Forbidden	N/A	TSA request authorisation failed.

rejection	unacceptedPolicy	Invalid TSA policy.
rejection	badDataFormat	TSA policy required in TSA request
rejection	systemFailure	Internal error occurred while processing the request.
rejection	unacceptedExtension	TSA request contains one or more unrecognised extensions.
rejection	badDataFormat	Required fields are missing in TSA request.
rejection	badAlg	Hash length in TSA request does not match with hash algorithm.
rejection	badAlg	Unsupported hash algorithm used to compute message imprint.
rejection	badDataFormat	TSP certReq flag not set in timestamp request.
rejection		TSA certificate is expired.
rejection	systemFailure	External TSA address not configured.
rejection	timeNotAvailable	External TSA request timed out.
rejection	timeNotAvailable	External TSA request forwarding failed.
rejection	systemFailure	Failed to communicate with any of the NTP Servers.

9 ADSS XKMS Service

The ADSS Server XKMS Service provides certificate validation services by supporting the XML Key Information Service (X-KISS) part of the XML Key Management Specification (XKMS 2.0) standard (http://www.w3.org/TR/2005/REC-xkms2-20050628/).

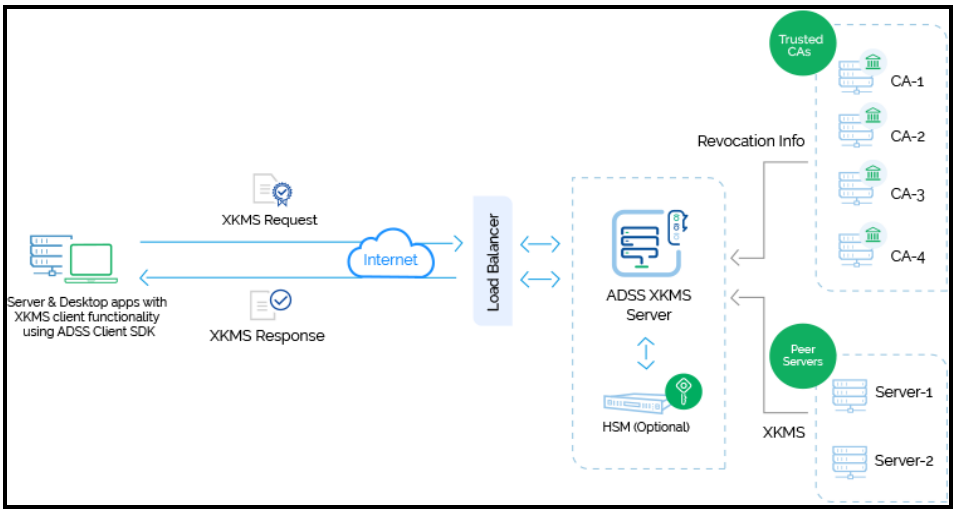
The protocol binding used is SOAP over HTTP(S) and a single stage synchronous protocol is used. Two variants are supported:

- Validation of a single certificate
- Validation of multiple certificates (a compound request)

Business applications typically use the XKMS Service when validating a certificate as part of XML digital signature verification. The ADSS Server Verification Service also makes use of the XKMS Validate Service when verifying XML digital signatures.

Business Client Applications send requests to the XKMS Validation Service to check that the target certificate is trusted, i.e. it is issued by a trusted CA, it has not expired, it is not revoked, it contains valid fields and extensions etc. The business application then receives a response back from the service

The majority of the verification parameters are already set up in profiles at the server but some may be overridden if permitted by the profile (e.g. certificate or key quality). The Business Application just needs to provide a list of the items it requires back in the response.



All the Trust Services shown above are provided either by ADSS Server or they can be external.

9.1 Support for the PEPPOL standard

The XKMS Service also implements PEPPOL Deliverable D1.1, Requirements for Use of Signatures in Public Procurement Processes:

- Part 5: XKMS v2 Interface Specification, and
- Part 7: eID and eSignature Quality Classification.

9.2 Setting up XKMS Validation Profiles

The ADSS XKMS Validation Service requires that XKMS Validation Profiles are defined at ADSS Server. These profiles specify trust anchors and certificate policies; define permitted certificate subject names, key usages, extended key usages, certificate quality levels, certificate path settings etc.

Refer to the following online admin guide for an explanation of Verification Profile settings:

Step 4 - Configuring XKMS Profile (ascertia.com)

9.3 The XKMS Validation Service API

The XKMS Validation Service API consists of the following classes:

- Validate Request – for sending a single certificate validation request
- Validate Result – for receiving the response to the above request
- Compound Request – for sending a multiple certificate validation request
- Compound Result - for receiving the response to the above request

9.4 Validate Request Class

9.4.1 Validate Request Constructor

The XKMS Validate Request Class is used when asking ADSS Server to validate a single X.509 certificate.

The following constructor is used to build the initial Validate Request and specifies a single 'respond with' item that is required in the response, in this case to respond with the actual certificate. Additional 'respond with' items can be added with the `AddRespondWith` method (described below).

```
var validateRequest = new ValidateRequest(ValidateRequest.RESPONDWITH_X509CERT);
```

9.4.2 Validate Request Methods

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send, SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

In addition, the following methods are specific to the Validate Request Class:

Validate Request Method	Purpose

`AddKeyUsage(string keyUsage)`	Specifies the required key usage for the certificate. Values can be:
	`KEYUSAGE_ENCRYPTION`
	`KEYUSAGE_EXCHANGE`
	`KEYUSAGE_SIGNATURE`

`AddRespondWith(string respondWith)`	Adds the 'respond with' item constant into the request. Whatever the constants sent in request, the corresponding values will be returned in the response. This method can be called multiple times if more 'respond with' items need to be set.
	The values that can be set are the same as for the constructor:
	`RESPONDWITH_KEYNAME`
	`RESPONDWITH_KEYVALUE`
	`RESPONDWITH_X509CERT`
	`RESPONDWITH_X509CHAIN`
	`RESPONDWITH_X509CRL`
	`RESPONDWITH_OCSP`
	`RESPONDWITH_SPKI`
	`RESPONDWITH_EIDQUALITY`
	`RESPONDWITH_OCSPNOCACHE`
	`RESPONDWITH_VALIDATIONDETAILS`

`AddUseKeyWith(string application, string identifier)`	Specifies a subject identifier and application identifier that determine a use of the key. The following application values are defined with an indication of the required identifier type in brackets:

	USEKEYWITH_PKIX (Id: Certificate Subject Name) USEKEYWITH_SMIME (Id: SMTP email address of subject) USEKEYWITH_TLS_HTTPS (Id: DNS address of http server) USEKEYWITH_TLS_SMTP (Id: DNS address of mail server)
SetKeyInfo(byte[] certificateToValidate)	Supplies the certificate as an array of bytes.
SetOpaqueClientData(string opaqueClientData)	This is data that is sent in the request and returned in a successful response. The meaning of opaque is that the data should be encrypted by the client as it is deemed to be confidential.
SetOriginatorId(string originatorId)	Specifies the client Id that will be used to authenticate the XKMS validate request.
SetProfileId(string profileId)	Specifies the profile Id that will be used to process this request.
SetTimeInstant(DateTime dateTime)	Specifies the time of production of the XKMS request.

9.4.3 Sending the XKMS Validate Request

Once the request message has been constructed and fully populated, it is sent to ADSS Server using the following method call:

```
var validateResult = (ValidateResult)validateRequest.Send(string URL);
```

The `URL` is that of the XKMS Service e.g. `http://machine-name:8777/adss/xkms`

The XKMS Service returns a response status and if this indicates success then all the requested response items are also included.

9.4.4 Example of creating and sending an XKMS Validate Request

```
// Construct xkms validate request
ValidateRequest validateRequest = new ValidateRequest(ValidateRequest.RESPONDWITH_X509CERT);
validateRequest.SetRequestID("xkmsValidateRequest:001");
validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_KEYNAME);
validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_KEYVALUE);
validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_SPKI);
validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_X509CHAIN);
validateRequest.AddRespondWith(ValidateRequest.RESPONDWITH_X509CRL);

validateRequest.SetKeyInfo(Util.Util.ReadFile(cert));
validateRequest.AddKeyUsage(ValidateRequest.KEYUSAGE_SIGNATURE);
validateRequest.AddUseKeyWith(ValidateRequest.USEKEYWITH_SMIME, "alice@example.com");

// Send request to the ADSS server
ValidateResult validateResult
    = (ValidateResult)validateRequest.Send("http://machine-name:8777/adss/xkms");
```

9.5 Validate Result Class

In common with the other response classes, XKMS Validate Result inherits the following methods from the Response and Message classes. There are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetRequestID, GetSigningCertificates, GetStatus, IsSuccessful.`

In addition, the following methods are specific to the XKMS Validate Result Class:

Validate Result Method	Purpose
`GetCertificate() returns byte[]`	Returns the certificate that was validated.
`GetCertificateChain()returns ArrayList`	Returns the target certificate chain.
`GetCertificateQuality() returns string`	Returns the certificate quality of the issuing CSP.
`GetCRLs() returns ArrayList`	Returns the list of CRLs used to check the revocation status of the target certificate chain.
`GetCspAssurance() returns string`	Returns the CSP assurance level of the issuing CSP.
`GetErrorDetail() returns string`	Returns the error detail defined by PEPPOL.
`GetErrorReason() returns string`	Returns the error reason defined by PEPPOL.
`GetId() returns string`	Returns the response identifier.
`GetIndeterminateReason() returns ArrayList`	Returns the list of indeterminate reasons in the Status element. These are status aspects that could not be evaluated or were evaluated but which returned an indeterminate result.
`GetInvalidReason() returns ArrayList`	Returns the list of invalid reasons in the Status element. These are status aspects that have been evaluated and found to be Invalid.
`GetKeyName() returns string`	Returns the Subject DN of the target certificate.
`GetKeyUsage() returns ArrayList`	Returns the list of key usages that have been matched in the target certificate.
`GetKeyValueExponent() returns byte[]`	Returns the public key exponent of the target certificate.
`GetKeyValueModulus() returns byte[]`	Returns the modulus of the public key of the target certificate.
`GetOcspCacheInterval() returns int`	Returns the life time (in minutes) of the OCSP cache.
`GetOcspResponses() returns ArrayList`	Returns the list of OCSP responses used to check the revocation status of the target certificate chain.
`GetOpaqueClientData() returns string`	Returns the opaque client data supplied in the request.
`GetResponderConfigurationVersion()`	Returns the responder configuration version.
`GetResponderName() returns string`	Returns the responder name.
`GetResponderURI() returns string`	Returns the responder URI.
`GetResultMajor() returns string`	Returns the `MajorResult` of the XKMS request.

	As only synchronous processing is currently supported, the `MajorResult` will be one of the `'Final'` results. These are:
	`Success:`
	`The operation succeeded.`
	`VersionMismatch:`
	`The service does not support the protocol version sent in the request.`
	`Sender:`
	`An error occurred that was due to the message sent by the sender.`
	`Receiver:`
	`An error occurred at the receiver.`

`GetResultMinor() returns string`	Returns the `MinorResult` of the XKMS request. These are the possible values: - `NoMatch` - `TooManyResponses` - `Incomplete` - `Failure` - `Refused` - `NoAuthentication` - `MessageNotSupported` - `UnknownResponseID` - `OptionalElementNotSupported` - `ProofOfPossessionRequired` - `TimeInstantNotSupported` - `TimeInstantOutOfRange.`
`GetRevocationReason() returns string`	Returns the revocation reason.
`GetRevocationTime() returns DateTime`	Returns the revocation time.
`GetService() returns string`	Returns the URL of the XKMS service.
`GetSPKI() returns byte[]`	Returns the hash of the target certificate public key.
`GetUseKeyWith() returns Hashtable`	Returns the list of extended key usages matched in the validated certificate.
`GetValidationModel() returns string`	Returns the validation model. (Only PKIX is supported currently).
`GetValidationScheme() returns string`	Returns the validation scheme i.e. either CRL or OCSP.
`GetValidationTime() returns DateTime`	Returns the time for which target certificate was validated.
`GetValidReason() returns ArrayList`	Returns the list of valid reasons in the Status element. These are status aspects that have been evaluated and found to be valid.
`IsOcspNoCache() returns bool`	Returns a flag stating that the OSCP response is not taken from the cache.

9.6 Compound Request Class

9.6.1 Compound Request Constructor

The XKMS Compound Request Class is used when asking ADSS Server to validate multiple X.509 certificates.

The following constructor is used to build the initial Compound Request and specifies a list of pre-built Validation Requests.

```
var compoundRequest = new CompoundRequest(compoundRequests);
```

9.6.2 Compound Request Methods

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send, SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

In addition, the following method is specific to the Compound Request Class:

Compound Request Method	Purpose
`AddRequest(request)`	Adds a validate request to an existing compound request.

9.6.3 Sending the XKMS Compound Request

Once the request message has been constructed and fully populated, it is sent to ADSS Server using the following method call:

```
var compoundResult = (CompoundResult)compoundRequest.Send(URL);
```

The `URL` is that of the XKMS Service e.g. `http://machine-name:8777/adss/xkms`

The returned XKMS Service returns a response status and if this indicates success then all the requested response items are also included.

9.7 Compound Result Class

The XKMS Compound Result class inherits the following methods from the Response and Message classes. There are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetRequestID, GetSigningCertificates, GetStatus, IsSuccessful.`

In addition, the following methods are specific to the XKMS Validate Result Class:

Compound Result Method	Purpose
`GetId() returns string`	Returns the response identifier.

`GetResultMajor() returns string`	Returns the `MajorResult` of the XKMS request.
	As only synchronous processing is currently supported, the `MajorResult` will be one of the `'Final'` results. These are:
	`Success`
	`The operation succeeded.`
	`VersionMismatch:`
	`The service does not support the protocol version sent in the request.`

	Sender: An error occurred that was due to the message sent by the sender. Receiver: An error occurred at the receiver.
GetResultMinor() returns string	Returns the MinorResult of the XKMS request. These are the possible values - NoMatch - TooManyResponses - Incomplete - Failure - Refused - NoAuthentication - MessageNotSupported - UnknownResponseID - OptionalElementNotSupported - ProofOfPossessionRequired - TimeInstantNotSupported - TimeInstantOutOfRange.
GetResults() returns List<ValidateRequest>	Returns the result of each XKMS validate request.

9.8 XKMS Service Sample Code

Java and .Net sample code is provided as part of the ADSS Client SDK and can be used to make XKMS Service requests and to process the responses.

The Java API provides the required classes under the package:

```
com.ascertia.adss.client.api.xkms
```

The .Net API provides the required classes under the namespace:

```
Com.Ascertia.ADSS.Client.API.XKMS
```

9.8.1 Java API Sample Code

The following sample programs demonstrate how the Java API can be used to send an XKMS Service request and to process the response:

```
samples/src/com/ascertia/adss/client/samples/xkms/CreateValidateRequest.java
samples/src/com/ascertia/adss/client/samples/xkms/CreateCompoundValidateRequest.java
```

Precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/XkmsValidate.bat
samples/bin/XkmsCompoundValidate.bat
```

9.8.2 .Net API Sample Code

The following sample programs demonstrate how the .Net API can be used to send an XKMS Service request and to process the response:

```
samples/src/Com/Ascertia/ADSS/Client/Samples/XKMS/CreateValidateRequest.cs
```

```
samples/src/Com/Ascertia/ADSS/Client/Samples/XKMS/CreateCompoundValidateR
equest.cs
```

Precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/XkmsValidate.bat
samples/bin/XkmsCompoundValidate.bat
```

9.9 ADSS XKMS Service Supported Algorithms

The following is a list of signing/hashing algorithms and key lengths that ADSS XKMS Service supports:

ADSS Service	Signature Algorithm	Hashing Algorithm	Algorithm / Key Sizes
XKMS	SHA1WithRSAEncryption SHA224WithRSAEncryption SHA256WithRSAEncryption SHA384WithRSAEncryption SHA512WithRSAEncryption RipeMD128WithRSAEncryption RipeMD160WithRSAEncryption SHA1withECDSA SHA224withECDSA SHA256withECDSA SHA384withECDSA SHA512withECDSA	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 RipeMD128 RipeMD160	RSA: 1024, 2048, 3072, 4096 ECDSA: 192,224,256,384,521

9.10 Error Codes

ADSS XKMS Service returns the following error codes in case of any failure:

Error Code	Error Message
46001	XKMS service not enabled in license.
46002	XKMS service license has expired.
46003	XKMS service is stopped.
46004	Signed request required.
46005	Request invalid and not according to schema.
46006	Signature verification failed.
46008	Failed to sign XKMS response.
46009	Certificate chain invalid in request.
46010	An internal error occurred while processing the request - see the XKMS service debug logs for details.
46011	TLS client certificate is revoked.
46012	TLS client certificate has unknown status.
46013	TLS client certificate has expired.
46014	Request signer certificate has unknown status.
46015	Request signer certificate is revoked.
46016	Request signer certificate has expired.
46017	XKMS request must use TLS client authentication.
46018	TLS certificate trust building failed.
46019	Certificate subject name is denied within the access control list.
46020	Certificate subject name is not in the include list within the access control list.
46021	IP address is not within the include list in the access control list.

46022	IP address is excluded within the access control list.
46023	Signed request required.
46024	Failed to build trust for request signer certificate
46025	XKMS service not allowed.
46026	XKMS profile is not allowed to the client.
46027	Originator ID not registered for this TLS client certificate.
46028	Originator ID not registered for this request signing certificate.
46029	Required elements are missing.
46030	Quality level not acceptable.
46031	Originator ID not found.
46032	XKMS profile is inactive.
46033	Validation at historic time is not configured.
46034	Wrong certificate format.
46035	Wrong time instance.
46036	XKMS service not enabled in system.
46037	XKMS profile does not exist or marked inactive.
46038	Default profile not configured and neither found in request.

10 ADSS SCVP Service

The IETF RFC 5055 Server-Based Certificate Validation Protocol (SCVP) is a relatively new, flexible, certificate validation protocol that is primarily intended to be used with large and complex PKI deployments (e.g. national PKIs). It can however also be used within smaller digital certificate trust schemes.

SCVP allows a client application to delegate all certificate validation tasks to the server. These include certificate path construction and path validation. The benefits of this are that simple clients and especially mobile devices can understand digital certificate trust. Centralised validation policies can be created and managed and thus client-side complexity, processing overhead and development effort can be substantially reduced.

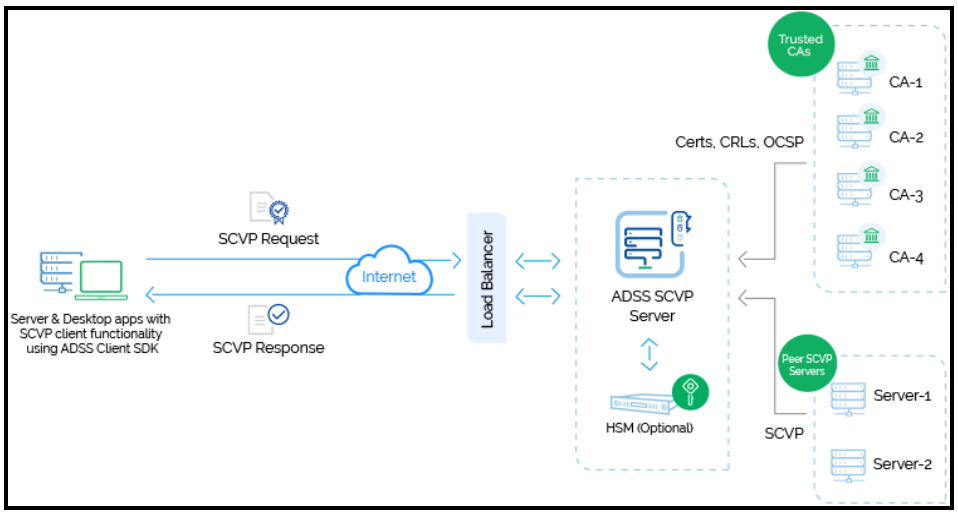
The ADSS SCVP Service follows the RFC 5055 specifications and supports two modes of operation:

- Delegated Path Discovery (DPD) and
- Delegated Path Validation (DPV)

With Delegated Path Discovery (DPD) the SCVP Service is asked to construct a valid certificate path from the supplied signing certificate back to a trusted root certificate but not to perform any validation. Validation is then performed on the client side.

In Delegated Path Validation (DPV) the SCVP Service is asked to construct the path as well as perform the certificate validation. It confirms that the public key belongs to the identity named in the signing certificate and that it can be used for the intended purpose.

ADSS SCVP Service also supports the ValPolRequest specified in RFC 5055. An SCVP client uses the ValPolRequest item to request information about an SCVP server's policies and configuration information, including the list of validation policies supported by the SCVP server. In response to a ValPolRequest, the ADSS SCVP server provides a ValPolResponse. The ValPolResponse may not be unique to any ValPolRequest, so may be reused by the server in response to multiple ValPolRequests.



Note that an SCVP Client can also be another SCVP Server. This is the case in SCVP relaying when one SCVP Server cannot answer and refers to another authoritative SCVP Server.

10.1 Simplified Use of SCVP

A simple use case of SCVP is where the Validation Policy is fully defined at the server. The standard allows complex request parameters, but in many cases simple clients just wish the server to follow the defined policy and give a simple trustworthy answer.

ADSS Server SCVP Validation Policies cover all requirements in the SCVP standard, including items such as establishing trust anchors, specifying certificate policies, defining permitted certificate subject names and key usages etc.

The definition of an SCVP Validation Policy is covered in detail in the ADSS Server Admin Guide an on-line version is provided here:

Configuring the SCVP Service (ascertia.com)

The ADSS Client SDK includes a pre-built SCVP client which can be used by either a Java or a .Net application. In a simple SCVP case, one where the SCVP Client is only interested in the status of a single certificate, only a few lines of code are required as illustrated below (error and other response processing code has been removed for clarity). Each of these .Net calls and the significance of the 'OID' values are described in detail later in this section.

```
public class Scvp
{
    public bool ValidateCertificate(X509Certificate x509Certificate)
    {
        // Construct SCVP Server request specifying the certificate to validate
        // and the ADSS Server Validation Policy OID
        var scvpRequest = new ScvpRequest(x509Certificate, "1.3.6.1.5.5.7.19.1");

        // Specify that end entity revocaton information is required in the response
        scvpRequest.AddWantBack("1.3.6.1.5.5.7.18.13");

        // Specify that ADSS Server is to build a validated path to a defined
        // trust anchor and to check revocation status
        scvpRequest.AddCertChecks("1.3.6.1.5.5.7.17.3");

        // Send the above constructed request to the ADSS Server SCVP Service
        // (in this case ADSS Server is on the local machine)
        var scvpResponse = (ScvpResponse)scvpRequest.Send("http://localhost:8777/adss/scvp");

        // If SCVP Response Status is OK then check certificate status
        if (scvpResponse.GetStatusCode() == ScvpResponse.OKAY)
        {
            List<CertReply> certReply = scvpResponse.GetReplyObjects();
            int certStatus = certReply[0].GetReplyStatus();
            if (certStatus == CertReply.SUCCESS)
            { return true; }
        }
        // otherwise return an error
        return false;
    }
}
```

Following sample can be used to get server policy configurations. It creates a validation policy request and send it to the server which in reponse send the SCVP server policy configuration supported by server.

```
// Create Validation polcy request
ScvpValidationPolicyRequest obj_scvpRequest = new ScvpValidationPolicyRequest();
Console.WriteLine("\n/************************************************************************/");
Console.WriteLine("\nRequest has been sent. Waiting for response...");

// Sending the above constructed request to the ADSS server
ScvpValidationPolicyResponse obj_scvpResponse = (ScvpValidationPolicyResponse) obj_scvpRequest.Send("http://localhost:8777/adss/scvp");

if (obj_scvpResponse.IsSuccessful()) {
    Console.WriteLine("\nRequest has been processed successfully.");
    Console.WriteLine("\nStatus : " + obj_scvpResponse.GetStatus());
    if (obj_scvpResponse.getServerConfigurationId() != -1) {
        Console.WriteLine("\r\n\r\nServer Configuration ID : " + obj_scvpResponse.getServerConfigurationId());
    }
    /** This update**/
    if (obj_scvpResponse.getThisUpDateTime() != null) {
        Console.WriteLine("\nThis Update: " + obj_scvpResponse.getThisUpDateTime());
    }

    /** next update **/
    if (obj_scvpResponse.gettNextUpDateTime() != null)
    {
        Console.WriteLine("Next update : " + obj_scvpResponse.gettNextUpDateTime());
    }
    //supportedChecks
    if (obj_scvpResponse.getSupportedChecks() != null && obj_scvpResponse.getSupportedChecks().Count > 0)
    {
        foreach (String strVal in obj_scvpResponse.getSupportedChecks())
        {
            Console.WriteLine("Supported Checks : " + strVal);
        }
    }
    /** Get all the required information here**/
} else {
    Console.WriteLine("\nStatus : " + obj_scvpResponse.GetStatus());
    Console.WriteLine("\nResponse failed : " + obj_scvpResponse.GetErrorMessage());
}
Console.WriteLine("\n/************************************************************************/");
```

Following sample is used to call the SCVP rest API to get all the SCVP policies configuration information.

```
ASC_GetPolicyInfoRequest policyInfoRequest = new ASC_GetPolicyInfoRequest( a_strRequestType: "policyInfo",  a_strPolicyOID: "1.3.6.1.5.5.7.19.1");

try {
    System.out.println("\n/************************************************************************/");
    System.out.println("\nRequest has been sent. Waiting for response...");

    ASC_GetPolicyInfoResponse response = (ASC_GetPolicyInfoResponse) policyInfoRequest.send( a_strBaseURL: "http://localhost:8777/adss/service/scvp/v1/policyInfo");
    if (response.isSuccessful()) {
        System.out.println("\nRequest has been processed successfully.");
        System.out.println("\nStatus : SUCCESS");
        ASC_PolicyInfoResponseVO dataVO = response.getDataVO();

        System.out.println("\nRequest Time : " + dataVO.getRequestTime());
        System.out.println("Response Time : " + dataVO.getResponseTime());
        System.out.println("\nPolicy count : " + dataVO.getPolicyCount());
        List<ASC_PolicyDetailVO> policies = dataVO.getPolicies();
        System.out.println("\nAvailable Policies : ");
        for (ASC_PolicyDetailVO policyDetailVO : policies) {
            System.out.println("\n****************** " + count++ + " ******************\n");
            System.out.println("Name : " + policyDetailVO.getName());
            System.out.println("Status : " + policyDetailVO.getStatus());
            System.out.println("Description : " + policyDetailVO.getDescription());
            System.out.println("Policy OID : " + policyDetailVO.getPolicyOID());
            System.out.println("Configuration ID : " + policyDetailVO.getConfigurationID());
            ASC_PolicyInfoTrustAnchorsVO trustAnchors = policyDetailVO.getTrustAnchors();
            System.out.println("\nTrust Anchors : ");
            System.out.println("\tAuto Trust New Auth: " + trustAnchors.isAutoTrustNewAuth());


            System.out.println("\n************************************************************");

        }
    } else {
        System.out.println("\nStatus : FAILED");
        System.out.println("\nHttp Response Status Code : "+response.getHttpStatus());
        System.out.println("\nService Error Code : "+response.getError());
        System.out.println("\nError Message : "+response.getErrorDescription());
    }
```

10.2 The SCVP Client API

The SCVP Client API (part of ADSS Client SDK) consists of three classes:

- The SCVP Request Class (ScvpRequest) which is used for creating, populating and sending the request
- The SCVP Response Class (ScvpResponse) which is used to retrieve information from the response message
- Certificate Reply Class (CertReply) which is used to retrieve validation information for each requested certificate.
- The SCVP Validation Policy Request Class (ScvpValidationPolicyRequest) which is used for creating, populating and sending the ValPolRequest.
- The SCVP Validation Policy Response Class (ScvpValidationPolicyResponse) which is used to retrieve policy information from the response message.
- The Get Policy Info Request Class (ASC_GetPolicyInfoRequest) which is used for sending the request to SCVP rest API to get policy information.
- The Get Policy Info Response Class (ASC_GetPolicyInfoResponse) which is used to retrieve policy information from the response message.

10.3 SCVP Request Class

The SCVP Request Class is used to create SCVP Requests for sending to the ADSS SCVP Service.

The following constructor is used to build an initial SCVP request message. This specifies a target certificate for validation plus a Validation Policy OID:

```
var scvpRequest = new ScvpRequest(x509Certificate,validationPolicyOid);
```

Validation Policy OIDs must match those defined within Validation Policies at ADSS Server. The special policy OID “1.3.6.1.5.5.7.19.1" is the default policy but other policy OIDs can be specified.

To add additional certificates into the request the following method is used:

```
scvpRequest.AddCertToValidate(anotherX509Certificate);
```

In the following sections the various methods of the SCVP Request Class are described. These are in addition to the methods inherited from the generic Request and Message classes described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

```
ToString, WriteTo, Send (overridden), SetProxy, SetRequestRetries,
SetSigningCredentials, SetSslClientCredentials (overridden), SetTimeout,
SetVerifyResponse.
```

10.3.1 Specifying Delegated Path Discovery or Validation

To specify whether Delegated Path Discovery (DPD) or Delegated Path Validation (DPV) is required the following method is used, supplying an appropriate OID parameter in the call:

```
scvpRequest.AddCertChecks(string);
```

The following Delegated Path OIDs are currently supported and have the following meaning:

For Delegated Path Discovery (DPD) use:

```
"1.3.6.1.5.5.7.17.1" (Build path to a defined trust anchor)
```

For Delegated Path Validation (DPV) use:

```
"1.3.6.1.5.5.7.17.2" (Build validated path to a defined trust anchor)
"1.3.6.1.5.5.7.17.3" (Build validated path to trust anchor and check
revocation)
```

10.3.2 Specifying the information the client 'wants back' about each certificate

The SCVP Client can specify what information it requires back about each certificate. It does this by making one or more calls to the following method:

```
scvpRequest.AddWantBack(string Oid);
```

These are the 'Want Back' OIDs currently supported (with pre-defined OID strings in brackets):

SCVP 'Want Back OID	Information Required in Response
"1.3.6.1.5.5.7.18.1" (BEST_CERT_PATH)	Return the certification path for the certificate including the certificate that was validated.
"1.3.6.1.5.5.7.18.2" (REVOCATION_INFO)	Return proof of revocation status for each certificate in the certification path.
"1.3.6.1.5.5.7.18.4" (PUBLIC_KEY_INFO)	Return the public key from the certificate that was the subject of the request.
"1.3.6.1.5.5.7.18.10" (CERT)	Return the public key certificate that was the subject of the request.
"1.3.6.1.5.5.7.18.12" (ALL_CERT_PATHS)	Return a set of certification paths for the certificate that was the subject of the request.
"1.3.6.1.5.5.7.18.13" (EE_REVOCATION_INFO)	Return proof of revocation status for the end entity certificate in the certification path.
"1.3.6.1.5.5.7.18.14" (CA_REVOCATION_INFO)	Return proof of revocation status for each CA certificate in the certification path.

For each specified 'want back' the requested information will be returned in the SCVP response.

10.3.3 Including Other SCVP Request Items

By using the following SCVP Request methods, a number of additional response items may be requested from the SCVP Service or additional information supplied to the service:

SCVP Request Method	Purpose
`SetFullRequestInResponse(bool)`	Determines whether full request details are returned in response.
`SetResponseValidationPolByRef (bool)`	Determines whether full policy details are returned in the response.
`SetProtectResponse(bool)`	Determines whether the response is protected (e.g. by signing).
`SetCachedResponse(bool)`	Specifies whether the SCVP Client will accept cached responses.
`SetNonce (byte[])`	Sets a 'nonce' value for the request. This requires the SCVP Server to send a 'fresh', non-cached, response.
`SetRequestorName(string, string)`	Specifies the requestor name (as a key/value pair) to be returned by the SCVP Server in the response.
`AddRequestorRef(string, string)`	Specifies a unique reference (as a key/value pair) within an SCVP Server network for the requesting SCVP Server. This is used to detect looping between SCVP Servers when SCVP Relaying is used.
`SetResponderName(string, string)`	Specifies (as a key/value pair) the identity of the SCVP Server that the client expects will sign the response.
`SetValidationTime(DateTime)`	Specifies the date and time for which the certificate validation is required.
`SetRequestorText(string)`	Specifies text for inclusion in the response, e.g. this could be text that describes the reason for the request.
`AddIntermediateCertificate (X509Certificate)`	Supplies certificates which the SCVP Server may use when forming a certification path.
`SetSignatureAlgorithm(string)`	Specifies the signature algorithm to be used by the SCVP Server to sign the response message.
`SetHashAlgorithm (string)`	Specifies the hash algorithm to be used by the SCVP Server for its response.
`SetSigningCredentials(string, string)`	Provides the path and password for a 'pfx' file to be used for signing SCVP Requests – overrides the method provided in the generic request class.

10.3.4 Validation Policy Overrides (PKIX Certificate Validation Settings)

If permitted by the Validation Policy defined for the SCVP Service, the following policy attributes may be overridden or added to by calling the associated 'Set' or 'Add' request methods:

SCVP Request Method	Policy being overridden
`SetBasicValidationAlgorithmOID(string)`	Specifies a different 'Base Validation Algorithm' to be used for certificate path validation.
`SetInhibitPolicyMapping(bool)`	Determines whether policy mapping is allowed during certificate path validation.
`SetRequireExplicitPolicy(bool)`	Determines whether there must be at least one valid policy in the certificate policies extension.
`SetInhibitAnyPolicy(bool)`	Determines whether the anyPolicy OID is processed or ignored when evaluating certificate policy.
`AddUserPolicySet(string)`	Adds to a list of certificate policy identifiers that the SCVP Service must use when constructing and validating a certificate path.
`SetNameValidationAlgorithmOID(string)`	Specifies one or more validation algorithm OIDs which specify subject name matching rules for the end entity certificate.
`AddKeyUsage(string)`	Adds a required key usage.
`AddExtendedKeyUsage(string)`	Adds an allowed Extended Key Usage
`AddSpecifiedKeyUsage(string)`	Adds a required Extended Key Usage
`AddTrustAnchor(string)`	Adds a Trust Anchor which can be used for certificate path validation.

10.3.5 Sending the SCVP Request

Once the request message has been constructed and fully populated, it is sent to ADSS Server using the following method call:

```
var scvpResponse = (ScvpResponse)scvpRequest.Send(URL);
```

The `URL` is that of the SCVP Service e.g. http://machine-name:8777/adss/scvp

The returned SCVP Response contains a response status and if this indicates success then all the requested response items are also included.

10.4 SCVP Response Class

10.4.1 SCVP Response Status Processing

The Response Status gives status information to the SCVP client about its request. This consists of a numerical error code and an optional human readable error message (currently this latter item is not supported).

The Response Status Code can be retrieved from the response with the following method:

```
int status = scvpResponse.GetStatusCode();
```

Various status codes are defined in the SCVP standard with values 0-9 reserved for successful responses (meaning that the server has processed them successfully, not that the validation results are positive). Codes 10 and above are reserved for error responses.

To simplify response processing, the following SCVP Response Status Code constants are defined:

Success Codes:

```
OKAY                                (status code = 0)
SKIP_UNRECOGNIZED_ITEMS             (status code = 1)
(Meaning that there were some non-critical extensions, however processing
was able to continue ignoring them)
```

Error Codes:

```
TOO_BUSY                                    (status code = 10)
INVALID_REQUEST                             (status code = 11)
INTERNAL_ERROR                              (status code = 12)
BAD_STRUCTURE                               (status code = 20)
UNSUPPORTED_VERSION                         (status code = 21)
ABORT_UNRECOGNISED_ITEMS                    (status code = 22)
UNRECOGNIZED_SIG_KEY                        (status code = 23)
BAD_SIGNATUREORMAC                          (status code = 24)
UNABLE_TO_DECODE                            (status code = 25)
NOT_AUTHORIZED                              (status code = 26)
UNSUPPORTED_CHECKS                          (status code = 27)
UNSUPPORTED_WANT_BACKS                      (status code = 28)
UNSUPPORTED_SIGNATUREORMAC                  (status code = 29)
INVALID_SIGNATUREORMAC                      (status code = 30)
PROTECTED_RESPONSE_UNSUPPORTED              (status code = 31)
UNRECOGNIZED_RESPONDER_NAME                 (status code = 32)
RELAYING_LOOP                               (status code = 40)
UNRECOGNIZED_VAL_POL                        (status code = 50)
UNRECOGNIZED_VAL_ALG                        (status code = 51)
FULL_REQUEST_IN_RESPONSE_UNSUPPORTED        (status code = 52)
FULL_POL_RESPONSE_UNSUPPORTED               (status code = 53)
INHIBIT_POLICY_MAPPING_UNSUPPORTED          (status code = 54)
REQUIRE_EXPLICIT_POLICY_UNSUPPORTED         (status code = 55)
INHIBIT_ANY_POLICY_UNSUPPORTED              (status code = 56)
VALIDATION_TIME_UNSUPPORTED                 (status code = 57)
UNRECOGNIZED_CRIT_QUERY_EXT                 (status code = 63)
UNRECOGNIZED_CRIT_REQUEST_EXT               (status code = 64)
```

10.4.2 SCVP Response Items

Assuming the Response Status Code is one of the success responses then the following methods retrieve relevant information from the SCVP Response.

SCVP Response Method	Purpose
`GetBasicValidationAlgorithmOID () returns string`	Returns the OID for the Basic Validation Algorithm
`GetExtendedKeyUsages() returns List<string>`	Returns a list of the Extended Key Usages.
`GetFullRequest() returns byte[]`	Returns a single data structure suitable for archiving the transaction.
`GetKeyUsages() returns List<string>`	Returns a list of the Key Usages.
`GetNameValidationAlgorithmOID( ) returns string`	Returns the OID for the Name Validation Algorithm.
`GetNonce() returns byte[]`	Returns the ‘nonce’ value for comparison with the one sent in the request.

`GetProducedAt() returns DateTime`	Returns the date and time when the response was produced.
`GetReplyObjects() returns List<CertReply>`	Returns the list of Certificate Reply objects. These contain validation information for each of the certificates in the request. The first 'CertReply' relates to the first certificate in the request, the second to the second, and so on. The CertReply class is covered in the next section.
`GetRequestHash() returns byte[]`	Returns the hash of the request which allows the client to match the response with a request message.
`GetRequestorNames() returns Hashtable`	Returns the requestor identities set by the client in the request.
`GetRequestorRef() returns Hashtable`	Returns the requestor reference information for the case where SCVP relay is used.
`GetRequestorText() returns string`	Returns the 'Requestor Text' sent by the client in the request.
`GetResponseValidationPolicy() returns string`	Returns a reference to the validation policy used to process the request.
`GetServerConfigurationId() returns int`	Returns the Server Configuration ID.
`GetSpecifiedKeyUsages() returns List<string>`	Returns the list of specified Extended Key Usages.
`GetTrustAnchors() returns X509Certificate[]`	Returns the list of trust anchors.
`GetUserPolicySet() returns List<string>`	Returns the list of certificate policy identifiers used by the SCVP Service when constructing and validating a certificate path.
`GetValidationNames() returns List<string>`	Returns the list of validation algorithm OIDs which were used to match subject names in the validated end entity certificates.
`GetVersion() returns int`	Returns the version of the SCVP protocol used.

In addition to the above, the SCVP Response class also inherits the following methods from the generic Response and Message classes. These are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetSigningCertificates, GetStatus, IsSuccessful.`

10.4.3 Cert Reply Class

The Certificate Reply Class provides methods for accessing validation information for each certificate sent in the SCVP request. The list of CertReply objects is retrieved from the SCVP Response as described in the previous section.

10.4.4 Accessing Certificate Validation Information

For each validated certificate, access is provided to the certificate validation information using the following methods:

Cert Reply Method	Purpose

`GetReplyCheckStatus() returns int`	Return the status for the DPD or DPV certificate check. The value of the reply depends upon the requested check.
	For DPD, the values are:
	`0: Built a path;`
	`1: Could not build a path`
	For DPV without revocation checking, they are:
	`0: Valid;`
	`1: Not valid`
	For DPV with revocation checking, they are:
	`0: Valid;`
	`1: Not valid;`
	`2: Revocation off-line;`
	`3: Revocation unavailable;`
	`4: No known source for revocation information`

`GetReplyStatus() returns int`	Returns the overall validation status for the requested certificate.
	The following status codes are defined:
	`SUCCESS (status code = 0)`
	`MALFORMED_PKC (status code = 1)`
	`MALFORMED_AC (status code = 2)`
	`UNAVAILABLE_VALIDATION_TIME (status code = 3)`
	`REFERENCE_CERT_HASH_FAIL (status code = 4)`
	`CERT_PATH_CONSTRUCT_FAIL (status code = 5)`
	`CERT_PATH_NOT_VALID (status code = 6)`
	`CERT_PATH_NOT_VALID_NOW (status code = 7)`
	`WANT_BACK_UNSATISFIED (status code = 8)`

`GetAllCertPaths() returns List<X509Certificate>`	Returns a set of certification paths for the validated certificate.
`GetBestCertPath() returns List<X509Certificate>`	Returns the certification path for the certificate including the certificate that was validated.
`GetCAsRevocationInfo() returns List<Hashtable>`	Returns the proof of revocation status for each CA certificate in the certification path.
`GetCertificate() returns X509Certificate`	Returns the validated certificate from the response.
`GetEndEntityRevocationInfo() returns List<Hashtable>`	Returns the proof of revocation status for the end entity certificate in the certification path.
`GetNextUpdate() returns DataTime`	Returns the date and time when the SCVP Server expects a refresh of the certificate validity information.
`GetPublicKeyInfo() returns byte[]`	Returns the public key from the end entity certificate.
`GetReplyCheckOID() returns string`	Returns the OID that identifies which type of certificate check was requested i.e. DPD or DPV (with or without revocation checking).

`GetReplyValidationTime() returns DateTime`	Returns the date and time for which the validation information is correct.
`GetRevocationInfo() returns List<Hashtable>`	Returns revocation information for the certificate.
`GetValidationErrors() returns List<string>`	Returns the validation errors for the certificate as a set of OIDs. These OIDs identify basic validation errors and name validation errors.
`GetWantBacks() returns List<string>`	Returns a list of the 'want back' OIDs sent in the request.

10.5 SCVP Validation Policy Request Class

The SCVP Validation Policy Request Class is used to create SCVP Requests for sending to the ADSS SCVP Service.

The following constructor is used to build an initial SCVP request message.

```
ScvpValidationPolicyRequest obj_scvpRequest= new ScvpValidationPolicyRequest();
```

If validation policy request have requestNonce then a fresh response is composed including this NONCE value from request and if request not including requestNonce then cached response is returned.

```
obj_scvpRequest.setNonce("abc".getBytes());
```

10.5.1 Sending the SCVP Validation Policy Request

Once the request message has been constructed and fully populated, it is sent to ADSS Server using the following method call:

```
var obj_scvpResponse = (ScvpValidationPolicyResponse) obj_scvpRequest.Send(URL);
```

The `URL` is that of the SCVP Service e.g. http://machine-name:8777/adss/scvp

The returned SCVP Response contains a response status and if this indicates success then all the requested response items are also included.

10.6 SCVP Validation Policy Response Class

The Response Status gives status information to the SCVP client about its request. If the response is successful it will return with status 200.

```
obj_scvpResponse.IsSuccessfull() {
   obj_scvpesponse.GetStatus();
}
```

10.6.1 SCVP Validation Response Items

Assuming the Response Status Code is one of the success responses then the following methods retrieve relevant information from the SCVP Valiadtion Policy Response.

SCVP Response Method	Purpose
`GetBasicValidationAlgorithmOID () returns string`	Returns the OID for the Basic Validation Algorithm
`GetExtendedKeyUsages() returns List<string>`	Returns a list of the Extended Key Usages.
`GetKeyUsages() returns List<string>`	Returns a list of the Key Usages.
`GetNameValidationAlgorithmOID( ) returns string`	Returns the OID for the Name Validation Algorithm.
`GetNonce() returns byte[]`	Returns the 'nonce' value for comparison with the one sent in the request.
`GetThisUpdate() returns DateTime`	Returns the date and time when the response was produced.
`GetNextUpdate() returns DateTime`	Returns the date and time when the new response will be computed next during cache process.
`GetDefaultValidationPolicy() returns string`	Returns a reference to the validation policy used by default by the server.
`GetServerConfigurationId() returns int`	Returns the Server Configuration ID.
`GetSpecifiedKeyUsages() returns List<string>`	Returns the list of specified Extended Key Usages.
`GetTrustAnchors() returns X509Certificate[]`	Returns the list of trust anchors.
`GetUserPolicySet() returns List<string>`	Returns the list of certificate policy identifiers used by the SCVP Service when constructing and validating a certificate path.
`GetValidationNames() returns List<string>`	Returns the list of validation algorithm OIDs which were used to match subject names in the validated end entity certificates.
`GetVersion() returns int`	Returns the version of the SCVP protocol used.
`GetSignatureGeneration() returns List<String>`	Returns the Signature generation algorithms supported by the server.
`GetSignatureVerification() returns List<String>`	Returns the Signature verification algorithms supported by the server.
`GetHashAlgorithms() returns List<String>`	Returns the hash algorithms supported by the server.
`GetResponseType()returns int`	Returns the response type supported by the server i.e. 2 (cached-and-non-cahed)
`GetMaxCVRequestVersion() returns int`	Returns the max CV request version supported y the server.
`GetMaxVPRequestVersion() returns int`	Returns the max VP request version supported y the server.
`GetSupportedChecks() returns List<String`	Returns the cert checks list supported by the server
`GetWantBacks()`	Returns the want backs supported by the server
`GetValidationAlgs() returns List<String>`	Returns the validation algorithms supported by the server.

`GetAuthPolicies() returns List<String>`	Returns the auth policies supported by the server.
`GetClockSkew() returns int`	Resturn the clock skew interval configured at server,
`GetInhibitPolicyMapping() retutns boolean`	Returns the Inhibit policy mapping flag configured in default validation policy
`GetRequireExplicitPolicy() retutns boolean`	Returns the Require explicit policy flag configured in default validation policy
`GetInhibitAnyPolicy() retutns boolean`	Returns the Inhibit any policy flag configured in default validation policy

In addition to the above, the SCVP Response class also inherits the following methods from the generic Response and Message classes. These are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetSigningCertificates, GetStatus, IsSuccessful.`

10.7 SCVP Get Policy Info Request Class

The Get Policy Info Request Class is used to create SCVP Request for sending to the ADSS SCVP Service.

The following constructor is used to build an initial SCVP request message.

```
ASC_GetPolicyInfoRequest obj_scvpRequest = new ASC_GetPolicyInfoRequest ("policyInfo", "1.3.6.1.5.5.7.19.1");
```

Policy OID is optinal here. If not specified the ADSS SCVP Service will return all the policies information.

10.7.1 Sending the SCVP Get Policy Info Request

Once the request message has been constructed and fully populated, it is sent to ADSS Server using the following method call:

```
var obj_scvpResponse = (ASC_GetPolicyInfoResponse) obj_scvpRequest.Send(URL);
```

The `URL` is that of the SCVP Service e.g.

http://machine-name:8777/adss/service/scvp/v1/policyInfo

The returned SCVP Response contains a response status and if this indicates success then all the requested response items are also included.

10.8 SCVP Get Policy Info Response Class

The Response Status gives status information to the SCVP client about its request. If the response is successful it will return with status 200.

```
obj_scvpResponse.IsSuccessfull() {
   obj_scvpesponse.GetStatus();
}
```

10.8.1 Get Policy Infor Response Items

Assuming the Response Status Code is one of the success responses then the following methods retrieve relevant information from the Get Policy Info Response.

SCVP Response Method	Purpose
`GetDataVO () returns ASC_PolicyInfoResponseVO`	Returns ASC_PolicyInfoResponseVO object containing the policy response values.

In addition to the above, the SCVP Response class also inherits the following methods from the generic Response and Message classes. These are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetSigningCertificates, GetStatus, IsSuccessful.`

10.9 SCVP Sample Code

Java and .Net sample code is provided as part of the ADSS Client SDK and this can be used to make SCVP Service requests and to process the SCVP Service response.

The Java API provides the required classes under the package:

`com.ascertia.adss.client.api.scvp`

The .Net API provides the required classes under the namespace:

`Com.Ascertia.ADSS.Client.API.SCVP.`

10.9.1 SCVP Service, Java API Sample Code

The following sample programs demonstrate how the Java API can be used to send a SCVP request and to process the response:

```
samples/src/com/ascertia/adss/client/samples/scvp/CreateCertValidateRequest.java
samples/src/com/ascertia/adss/client/samples/scvp/CreateValidationPolicyRequest.java
```

A precompiled and ready to run version of the above sample program can be found at:

```
samples/bin/ScvpCertValidate.bat
samples/bin/ScvpValidationPolicy.bat
```

10.9.2 SCVP Service .Net API Sample Code

The following sample programs demonstrate how the .Net API can be used to send a SCVP request and process the response:

```
samples/src/Com/Ascertia/ADSS/Client/Samples/SCVP/CreateCertValidateRequest.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/SCVP/CreateValidationPolicyRequest.cs
```

A precompiled and ready to run version of the above sample program can be found at:

```
samples/bin/ScvpCertValidate.bat
samples/bin/ScvpValidationPolicy.bat
```

10.10 ADSS SCVP Service Supported Algorithms

The following is a list of signing/hashing algorithms and key lengths that ADSS SCVP Service supports:

ADSS Service	Signature Algorithm	Hashing Algorithm	Algorithm / Key Sizes
SCVP	SHA1WithRSAEncryption SHA224WithRSAEncryption SHA256WithRSAEncryption SHA384WithRSAEncryption SHA512WithRSAEncryption RipeMD128WithRSAEncryption RipeMD160WithRSAEncryption SHA1withECDSA SHA224withECDSA SHA256withECDSA SHA384withECDSA SHA512withECDSA	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 RipeMD128 RipeMD160	RSA: 1024, 2048, 3072, 4096 ECDSA: 192,224,256,384,521

10.11 Error Codes

ADSS SCVP Service returns the following error codes in case of any failure:

Error Code	Error Message
47001	SCVP service not enabled in license.
47002	SCVP service license has expired.
47003	SCVP service is stopped.
47004	Signed request required.
47005	An internal error occurred while processing the request - see the SCVP service debug logs for details.
47006	SCVP request signature verification failed.
47007	Failed to authenticate SCVP request.
47008	SCVP version not supported.
47009	SCVP request contains unsupported request items.
47010	SCVP request references an unknown validation policy.
47011	SCVP server does not support sending validation policy value.
47012	SCVP response signing certificate not found.
47013	SCVP server does not have certificate matching the requested responder name.
47014	Invalid SCVP request.
47015	Certificate validation failed.
47016	Certificate validation using SCVP request validation time is not enabled.
47017	An internal error occurred while processing the request - see the SCVP service debug logs for details
47018	TLS client certificate is revoked.
47019	TLS client certificate has an unknown status.
47020	TLS client certificate has expired.
47021	Request signer certificate has unknown status.
47022	Request signer certificate is revoked.

47023	Request signer certificate has expired.
47024	SCVP request must use TLS client authentication.
47025	TLS certificate trust building failed.
47026	Certificate subject name is excluded within the access control list.
47027	Certificate subject name is not included within the access control list.
47028	IP address is not include within the access control list.
47029	IP address is excluded within the access control list.
47030	The request must be signed.
47031	Request signer certificate trust building failed.
47032	Request signer certificate subject name is excluded within the access control list.
47033	Request signer certificate subject name is not included within the access control list.
47034	Failed to authenticate SCVP request, required extended key usage OID does not exist in request signing certificate.
47035	Failed to authenticate SCVP request, required key usage OID does not exist in request signing certificate.
47036	SCVP service not enabled in system.
47037	SCVP request references an inactive validation policy.
47038	SCVP request contains unknown cert check.
47039	SCVP response signing certificate revoked.

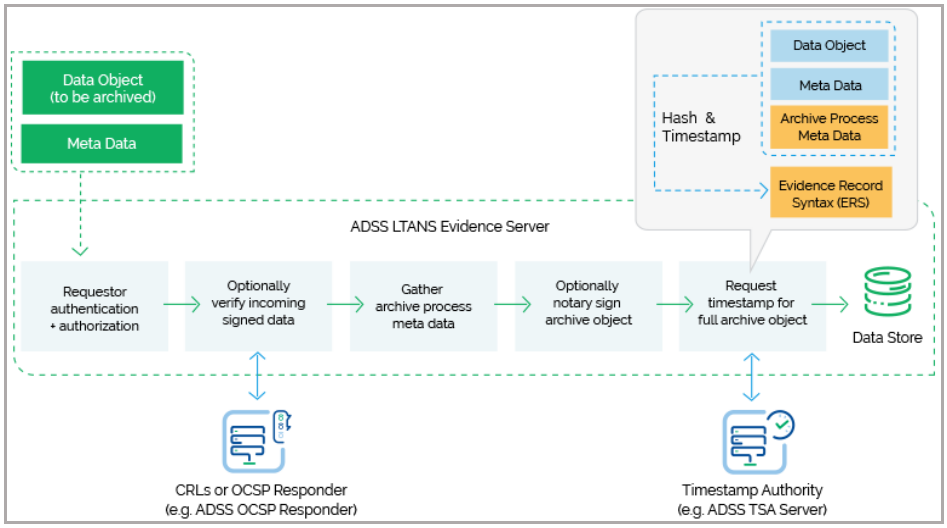
11 ADSS LTANS Service

11.1 LTANS Service

The ADSS LTANS Service provides long-term archiving and notary services that follow the draft IETF LTANS standard and IETF XML Evidence Record Syntax (XMLERS) standard. The service generates evidence records (XMLERS data) for the requesting client application which would typically be a document or record management system.

To utilise the ADSS LTANS Service, the client business application sends an Archive Request to the ADSS LTANS Service containing the data to be archived (e.g. a document, data, a signed transaction etc.). The service creates a secure archive object for the data, using the configured Time Stamp Authority (TSA), and in compliance 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) or HTTP Protocol as illustrated below:



Note ADSS LTANS Service can either store the archive object internally in its database or return it to the client application for local storage (e.g. the archive object may be stored in the Document Management System).

If the ADSS LTANS Service is responsible for storing the archive object then the business application can at a later date export the archive object out of the archive.

ADSS LTANS Service implements the LTAP interface protocol operations (archive, export, delete, verify, status and 'listids') to provide an industry way to securely store, retrieve and verify the documents and other important data objects for a longer period. Thus the client business application may ask the ADSS LTANS Service to:

- Verify or delete a particular archive object
- Request status for a particular archive object and
- List the archive objects based upon specific filter criteria.

ADSS LTANS service also supports Renew Evidence operation using HTTP Interface only to renew evidence record.

11.2 LTANS Service Profiles

The ADSS LTANS Service requires that LTANS Profiles are defined for the ADSS LTANS Service. These specify the policy for the archive e.g. the archive lifetime, what happens at the end of this period, whether archive objects need to have their evidence records periodically refreshed, and which Timestamp Authorities to use for time-stamping the archive objects

Refer to the following online admin guide for a full explanation of LTANS Profile settings:

Configuring the LTANS Service (ascertia.com)

11.3 The LTANS Service API

In order to simplify the use of the LTAP protocol an LTANS Service API is provided as part of the ADSS Client SDK.

The API consists of two classes:

- Archiving Request
- Archiving Response

11.4 Archiving Request Class

11.4.1 Archiving Request Constructor

The Archiving Request Class has a single constructor with three parameters: `clientID`, `serviceType`, `serviceID`.

The `serviceType` can take one of the following values, depending upon the service required:

```
ArchivingRequest.SERVICE_TYPE_ARCHIVE
ArchivingRequest.SERVICE_TYPE_DELETE
ArchivingRequest.SERVICE_TYPE_EXPORT
ArchivingRequest.SERVICE_TYPE_LISTIDS
ArchivingRequest.SERVICE_TYPE_STATUS
ArchivingRequest.SERVICE_TYPE_VERIFY
ArchivingRequest.SERVICE_TYPE_RENEW
```

The `serviceID` is a unique identifier for the service.

```
var archivingRequest = new ArchivingRequest(clientID, serviceType,
serviceID);
```

11.4.2 Archiving Request Methods

The Archiving Request Class inherits the following methods from the generic Request and Message classes. These are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send, SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials (overridden), SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

In addition, the following methods are specific to the Archiving Request Class:

Archiving Request method	Purpose
`AddMetaItem(string metaItemType, string metaItemValue)`	Provides some meta data to be associated with the archive data object.
`SetData(byte[] /string)`	Provides the data to be archived.
`SetDataType(string mimeType)`	Specifies the MIME type of the data to be archived.
`SetFilePath(string filePath)`	As an alternative to sending the data to be archived, it can be provided as a network file path.
`SetNonce(string nonce)`	Provides a nonce value (a random value).
`SetPolicyID(string profile)`	Specifies the LTAN profile to be used for the request.
`SetReference(string reference)`	Specifies a reference for the archived data.

`SetRequestTime(DateTime requestTime)`	Specifies the request creation time.
`SetSerial(string serialNumber)`	Specifies a serial number for the request.
`SetTransactionId(string transactionIdentifier)`	Specifies the unique transaction Id to be assigned to the request.
`SetVersion(string version)`	Specifies the version number of the protocol.
`SetRequestMode(int mode)`	Specifies the request mode which can be either XML or HTTP (ArchivingRequest.XML, ArchivingRequest.HTTP) – XML is the default mode.

11.4.3 Sending the Archiving Request

Once the Archiving request message has been prepared, it is sent to ADSS Server using the following method call:

```
var archivingResponse =
(ArchivingResponse)archivingRequest.Send(ltansServiceAddress);
```

An example of the `ltansServiceAddess URL` is: http://machine-name:8777/adss/ltap

11.4.4 Example of creating and sending an Archiving Request

```
// Construct Archiving request
var archivingRequest
    = new ArchivingRequest(clientID, ArchivingRequest.SERVICE_TYPE_ARCHIVE, serviceID);
archivingRequest.SetVersion("1.0");
archivingRequest.SetRequestID(requestID);
archivingRequest.SetPolicyId(profileID);
archivingRequest.SetTransactionId(transID);
archivingRequest.SetNonce(randomNonce);
archivingRequest.SetRequestTime(DateTime.Now);

// Prepare data string element so that it can be added into the archiving request
byte[] archiveData = Util.ReadFile(inFile);
archivingRequest.SetData(archiveData);
archivingRequest.SetDataType(mimeType);

// Send LTAP request to the ADSS LTANS Service
var archivingResponse = (ArchivingResponse)archivingRequest.Send(ltanServiceAddress);
```

11.5 Archiving Response Class

The Archiving Response class inherits the following methods from the Response and Message classes. There are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetRequestID, GetSigningCertificates, GetStatus, IsSuccessful.`

Note, `GetStatus()` returns the status of the request, either `granted` or `rejected`.

In addition, the following methods are specific to the class:

Archiving Response Method	Purpose
`GetCoreServiceType() returns string`	Returns the type of LTAN operation.
`GetData() returns byte[]`	Returns the archived data.
`GetDataAsString()returns string`	Returns the archived data in string format.
`GetDataOpaque() returns string`	Returns the data as a Base 64 encoded string.
`GetListIds() returns ArrayList`	Returns the ListIDs if these were requested in the Archive Request.
`GetMetaItems() returns Hashtable` `or GetMetaItems (string refID) returns Hashtable`	Returns the Meta Data (optionally supplying a reference identifier).
`GetNonce() returns string`	Returns the nonce value for comparison with the one sent in the request.
`GetPolicyId() returns string`	Returns the LTAN profile Id.
`GetReference() returns string`	Returns the reference of the archived data.
`GetRequesterId() returns string`	Returns the unique identifier of the request.
`GetRequestTime() returns DateTime`	Returns the request time that was sent in the request.
`GetSerial() returns string`	Returns the serial number of the request.
`GetServiceId() returns string`	Returns the unique identifier of the service.
`GetTransactionId() returns string`	Returns the unique transaction identifier of the request.
`GetVersion() returns string`	Returns the version of the protocol.
`GetNotarySignatureStatus() returns VerificationResponse`	Returns the Verification response of Notary signature.
`getDataSignatureStatusWhenArchived() returns VerificationResponse`	Returns the Verification response of signed object when archived.
`getDataSignatureStatus () returns VerificationResponse`	Returns the Verification response of signed object.
`getTimestampTokens() returns TimeStampToken`	Returns the array of timestamp tokens.
`getErsInputXML () returns Document`	Returns the XML that was used in ERS computation.
`getNotarySignature() returns Document`	Returns the Notary signature.

11.5.1 Example of processing the Archiving Response

```
// Send LTAP request to the ADSS LTANS Service
var archivingResponse = (ArchivingResponse)archivingRequest.Send(ltanServiceAddress);

// Check Response Status
if (!archivingResponse.IsSuccessful())
{
    displayMessage = (" Status: " + archivingResponse.GetStatus()
    + "; Error Code: " + archivingResponse.GetErrorCode().ToString()
    + "; Error Message: " + archivingResponse.GetErrorMessage());
}
else
{
    displayMessage = (" Status: " + archivingResponse.GetStatus()
         + "  Service Type : " + archivingResponse.GetCoreServiceType()
         + ";  Nonce : " + archivingResponse.GetNonce()
         + ";  Request Time : " + archivingResponse.GetRequestTime()
         + ";  Requester ID : " + archivingResponse.GetRequesterId()
         + ";  Reference ID : " + archivingResponse.GetReference());
}
```

11.6 LTANS Service Sample Code

Java and .Net sample code is provided as part of the ADSS Client SDK and can be used to make LTANS Service requests and to process the responses.

The Java API provides the required classes under the package:

```
com.ascertia.adss.client.api.ltan
```

The .Net API provides the required classes under the namespace:

```
Com.Ascertia.ADSS.Client.API.LTAN.
```

11.6.1 Java API Sample Code

The following sample programs demonstrate how the Java API can be used to send an archiving request to the LTANS Service and to process the response:

```
samples/src/com/ascertia/adss/client/samples/ltan/CreateLtanArchivingRequ
est.java
samples/src/com/ascertia/adss/client/samples/ltan/CreateLtanExportRequest
.java
samples/src/com/ascertia/adss/client/samples/ltan/CreateLtanDeleteRequest
.java
samples/src/com/ascertia/adss/client/samples/ltan/CreateLtanListIDsReques
t.java
samples/src/com/ascertia/adss/client/samples/ltan/CreateLtanRenewRequest.
java
```

Precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/LtanArchive.bat
samples/bin/LtanExport.bat
samples/bin/LtanDelete.bat
samples/bin/LtanListIDs.bat
samples/bin/LtanRenew.bat
```

11.6.2 .Net API Sample Code

The following sample programs demonstrate how the .Net API can be used to send an archiving request to the LTANS Service and to process the response:

```
samples/src/Com/Ascertia/ADSS/Client/Samples/LTAN/CreateLtanArchivingRequest.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/LTAN/CreateLtanExportRequest.cs
samples/src/Com/Ascertia/ADSS/Client/Samples/LTAN/CreateLtanDeleteRequest.cs
samples/src/com/ascertia/adss/client/samples/ltan/CreateLtanListIDsRequest.cs
samples/src/com/ascertia/adss/client/samples/ltan/CreateLtanRenewRequest.cs
```

Precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/LtanArchive.bat
samples/bin/LtanExport.bat
samples/bin/LtanDelete.bat
samples/bin/LtanListIDs.bat
samples/bin/LtanRenew.bat
```

11.7 ADSS LTANS Service Supported Algorithms

The following is a list of signing/hashing algorithms and key lengths that ADSS LTANS Service supports:

ADSS Service	Signing Algorithms	Hashing Algorithms	Signing Key Lengths
LTANS	SHA1WithRSAEncryption SHA224WithRSAEncryption SHA256WithRSAEncryption SHA384WithRSAEncryption SHA512WithRSAEncryption RipeMD128WithRSAEncryption RipeMD160WithRSAEncryption SHA1withECDSA SHA224withECDSA SHA256withECDSA SHA384withECDSA SHA512withECDSA	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 RipeMD128 RipeMD160	RSA: 1024, 2048, 3072, 4096 ECDSA: 192,224,256,384,521

11.8 Error Codes

ADSS LTANS Service returns the following error codes in case of any failure:

Error Code	Error Message
48001	LTANS service not enabled in license.
48002	LTANS web service not enabled.
48003	LTANS service license has expired.
48004	LTANS service is stopped.
48005	The request must be signed.
48006	Request does not comply with LTAP XML schema.
48007	An internal error occurred while processing the request - see the LTANS service debug logs for details.

48008	Request signature does not verify.
48009	See the LTANS service debug logs for details.
48010	Request contains unsupported parameters.
48011	Requested profile name is not found.
48012	Timestamp token not received for archive data.
48013	Evidence record not generated.
48014	Archived object could not be signed.
48015	An internal error occurred while processing the request - see the LTANS service debug logs for details.
48016	Request does not contain data to archive.
48017	Archived object does not exist.
48018	Requested operation not supported.
48019	Default profile not configured and neither found in request.
48020	Archive object signing certificate has expired.
48021	Archive object signing certificate is revoked or has a status of unknown.
48022	Archive object cannot be published on the URL.
48023	Archive object could not be written to file system.
48024	Profile does not allow archive object deletion.
48025	Operation not allowed by profile.
48026	Archive data signature not verified.
48027	Archive data not found.
48028	Profile does not allow verification of archived objects.
48029	Failed to create XAdES-X-L signature for archive object.
48030	No meta data found in request.
48031	Either LTANS profile does not exist or marked inactive.
48032	LTANS profile is not allowed to the client.
48033	Either LTANS default profile is inactive or not allowed to client.
48034	An internal error occurred while processing the request - see the LTANS service debug logs for details.
48035	No LTAP operation specified.
48036	Requested LTAP operation not allowed.
48037	LTANS service not allowed.
48038	Failed to read data from specified file path.
48039	An internal error occurred while processing the request - failed to insert meta data and/or archived object - see the LTANS service debug logs for details.
48040	Archived object does not exist.

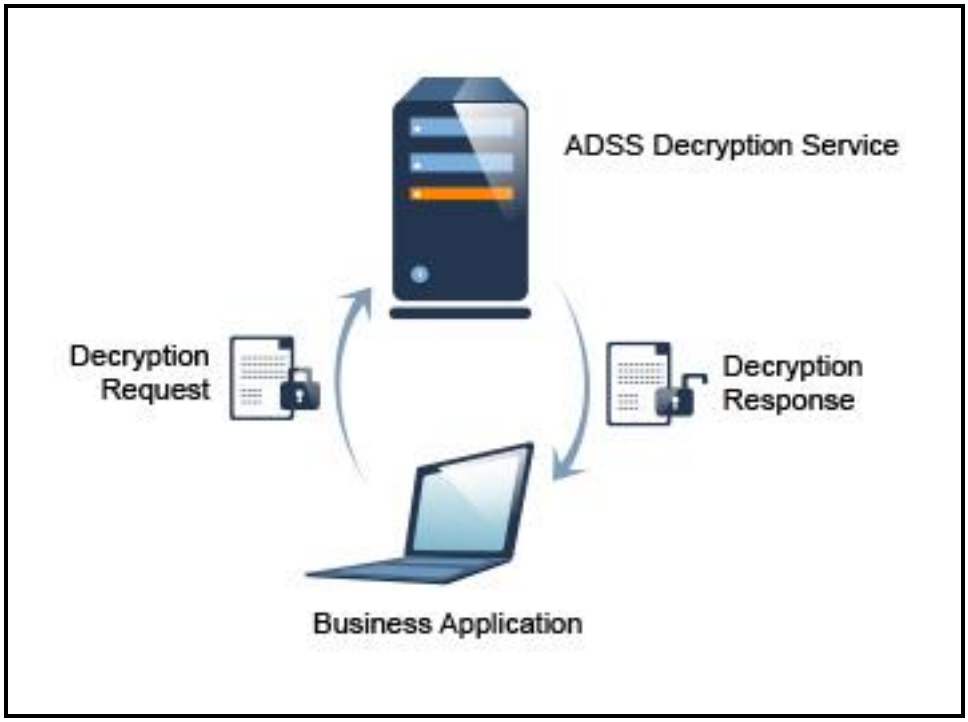
48041	Cannot save archived object at the specified file path.
48042	Archived object does not exist.
48043	See the LTANS service debug logs for details.
48044	Invalid archived object deletion setting.
48045	Archived object does not exist.
48046	Archived object does not exist.
48047	Archived object not signed.
48048	Archive object hash not available.
48049	Archived object hash and archive signature hash does not match.
48050	Hash could not be computed.
48051	Evidence record data hash does not match archive hash.
48052	Archived data signature does not verify.
48053	LTANS service not enabled in system
48054	No archived object found matching the provided meta data.
48055	Data type of archive object not found in request.
48056	ERS or XML has altered.
48057	Meta item type has changed.
48058	Meta item value has changed.
48059	LTANS profile is inactive.
48060	LTANS default profile is inactive.
48061	Archived object is inactive.
48062	Data type length should not be greater than 20 characters.
48063	Length of meta item type should not be greater than 200 characters.
48064	Length of meta item value should not be greater than 500 characters.
48065	Archive object does not exist against the reference number.

12 ADSS Decryption Service

The Decryption Service provides a centralised document and data decryption service under controlled and authorised conditions. The decryption protocol is based on the OASIS DSS-X decryption profile.

End users submit encrypted (possibly signed and encrypted) documents to a business application and these are decrypted at ADSS Server. For example, in an e-Tendering application, a special version of Go>Sign Desktop is used to Encrypt XML documents which the business application then asks ADSS Server to decrypt according to a Decryption Profile. The Decryption Profile specifies which keys the Decryption Service should use for the decryption.

The following diagram illustrates the process:



12.1 ADSS Decryption Service Profiles

The ADSS Decryption Service requires that Decryption Profiles are defined at ADSS Server. These specify how an encrypted object will be decrypted by the service.

Refer to the following online admin guide for an explanation of Decryption Profile settings:

Configuring the Decryption Service (ascertia.com)

12.2 The ADSS Decryption Service API

In order to simplify the use of the OASIS DSS-X Decryption protocol a Decryption Service API is provided as part of the ADSS Client SDK.

The API consists of the following classes:

- Decryption Request
- Decryption Response

12.3 Decryption Request Class

12.3.1 Decryption Request Constructor

The Decryption Request Class has four constructors which allow for different ways to specify the source of the data to be decrypted. Currently just XML decryption is supported and the XML data can be provided as a `file path string`, `byte[]`, `Stream` or `XmlDocument`.

Below is an example of the constructor where the data is provided as a file path:

```
var decryptionRequest = new DecryptionRequest(clientID, filePath,
DecryptionRequest.MIME_TYPE_XML);
```

12.3.2 Decryption Request Methods

The Decryption Request Class (DecryptionRequest) inherit a number of methods from the generic Request and Message classes which are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send, SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

In addition, the following methods are specific to the Decryption Request Class:

Decryption Request method	Purpose
`SetCertificateAlias(string certAlias)`	Specifies the key used for decryption. This will override any key set up in the referenced profile.
`SetPassword(string password)`	Specifies the PKCS12 password for accessing the decryption key.
`SetProfileId(string profileId)`	Sets the profile ID.

12.3.3 Sending the Decryption Request

Once the Decryption request message has been prepared, it is sent to ADSS Server using the following method call:

```
var decryptionResponse =
(DecryptionResponse)decryptionRequest.Send(decryptionServiceAddress);
```

The `decryptionServiceAddress URL` is that of the Decryption Service e.g. http://machine-name:8777/adss/decryption

12.3.4 Example of creating and sending a Decryption Request

```
// Construct Decryption Request
var decryptionRequest = new DecryptionRequest(clientID, inFile, DecryptionRequest.MIME_TYPE_XML);

decryptionRequest.SetCertificateAlias(certAlias);

// Send decryption request to the ADSS server
DecryptionResponse decryptionResponse = (DecryptionResponse)decryptionRequest.Send(decryptionServiceAddress);
```

12.4 Decryption Response Class

The Decryption Response class (DecryptionResponse) inherits the following methods from the Response and Message classes. There are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

```
ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage,
GetException, GetRequestID, GetSigningCertificates, GetStatus,
IsSuccessful.
```

In addition, the following methods are specific to the class:

Decryption Response Method	Purpose
`GetDocument() returns byte[]`	Returns the plain (clear text) document.
`GetProfileId() returns string`	Returns the Decryption Profile Id used to process the request.
`GetXmlDocument() returns XmlDocument`	Returns the plain (clear text) XML document.
`PublishDocument(string /Stream)`	Publishes the plain (clear text) to the specified path or stream.

12.5 Error Codes

ADSS Decryption Service returns the following error codes in case of any failure:

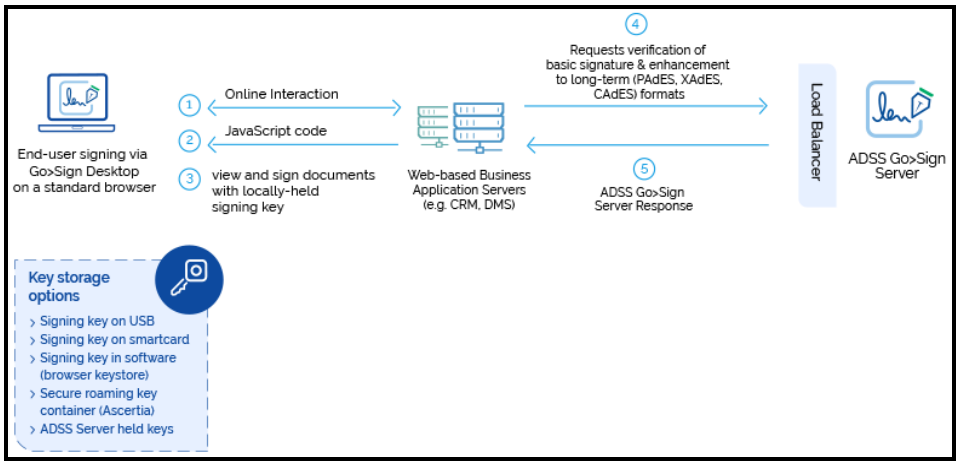
Error Code	Error Message
49001	An internal error occurred while processing the request - see the Decryption service debug logs for details.
49002	Originator authentication failed.
49003	Decryption service license has expired.
49004	Decryption service is stopped.
49005	Decryption service not enabled in license.
49006	Signed request required.
49007	Signature verification failed.
49008	Request is invalid and not according to schema.
49009	The decryption profile is not appropriate for this file type.
49010	Private key is not available.
49011	Failed to decrypt document.
49012	Invalid encrypted document structure.
49013	Encryption certificate is not available.
49014	Decryption key is not available.
49015	Decryption profile is not allowed to the client.
49016	Certificate for decryption not allowed for this client.
49017	Decryption service not allowed.
49018	Decryption service not enabled in system.
49019	Decryption profile is inactive.

49020	Decryption profile does not exist or marked inactive.
49021	Default decryption profile not configured. Provide decryption profile in request.

13 ADSS Go>Sign Service

13.1 ADSS Go>Sign Service Overview

ADSS Go>Sign Service empowers business applications to perform document signing on user's machines using the credentials held either locally by the user or server-side keys. ADSS Go>Sign Service also enables business applications to show PDF documents to users using a server-side HTML-based Go>Sign Document Viewer.



The above diagram describes how the ADSS Go>Sign Service and business application interact with each other. The high-level process is as follows:

- Business application specific web page sends a request to the ADSS Go>Sign Service providing information about its document signing needs.
- The ADSS Go>Sign Service receives the request and responds to web page with the relevant JavaScript code to service its needs.
- The web page receives the JavaScript code and renders it for the user.
- The user can then optionally view the document and sign it using either locally-held or server-held signing key (note Go>Sign service also supports key generation and certification services).
- During the signing process, the ADSS Go>Sign Service may use the backend ADSS Services, e.g. to generate server-side signatures, verify signatures created by the user, and to enhance basic user signatures into long-term signature formats. Furthermore, if the Go>Sign Service is being used for key generation and certification, then the back ADSS Server can be used to issue the certificates for the user and securely store the user's private key container.

The ADSS Go>Sign Service consists of two major components: The Go>Sign Desktop and the Go>Sign Document viewer. A business application can use any of the following combination based on its requirements:

- Go>Sign Desktop only (e.g. if the business application will display the document by itself)
- Go>Sign Document Viewer only (this is not so common, as the primary purpose of the Go>Sign service is to sign documents)
- Go>Sign Desktop and Document Viewer (this is where the business application is relying on the Go>Sign Service to display the PDF document to the user and also to get the user to sign it).

Note: In order to learn how the business applications can integrate ADSS Go>Sign Service and utilize it features, read the "**ADSS-Go-Sign-Developers-Guide.pdf**" shipped within the ADSS Client SDK package.

13.2 Error Codes

ADSS Go>Sign Service returns the following error codes in case of any failure:

Error Code	Error Message
52551	User information is not available.
52552	PDF is not available.
52553	Document is not available in request.
52254	Originator ID not found in the request.
52255	Internal Error
52256	Go>Sign profile is inactive.
52257	Go>Sign profile does not exist.
52258	Go>Sign profile is not allowed to this client.
52259	Go>Sign service is not allowed to this client.
52260	Incorrect file format and cannot be converted into PDF.
52261	Field information is not available to the Go>Sign Service.
52262	Field signing failure.
52263	Error occurred during processing request.
52264	Signing service not available.
52265	Failed to create fields.
52266	Document conversion is not allowed.
52268	Go>Sign transaction id not available.
52269	Invalid originator ID in the request.
52270	Default profile not configured and neither found in request.
52271	The field name is not specified.
52272	Invalid field coordinates.
52273	Failed to create fields in the document.
52274	A field with same name already exists.
52275	Some mandatory request parameter(s) are missing.
52277	Client session timed out.
52278	PDF form filling failure.
52279	PDF form filling not allowed.
52280	Certificate not found.
52281	Go>Sign service is not enabled in license.
52282	Go>Sign service license is expired.
52283	Go>Sign service is not enabled in the system.
52284	Go>Sign service is stopped.
52285	Authentication failed.
52286	The document already contains Document Timestamp signature.
52287	The document permissions do not allow this operation.

52288	Downloading of unprocessed document is not allowed.
52289	Signature timeout reached. Retry.
52290	Mobile signing failure.
52292	User cancelled signing operation.
52293	Invalid transaction id.
52294	Signing certificate is revoked.
52295	Signature not ok.
52296	Signature invalid.
52297	Assembly operation failure.
52298	Unsupported signature type.
52299	Input is not a valid MS Word document or corrupted.

14 ADSS RA Service

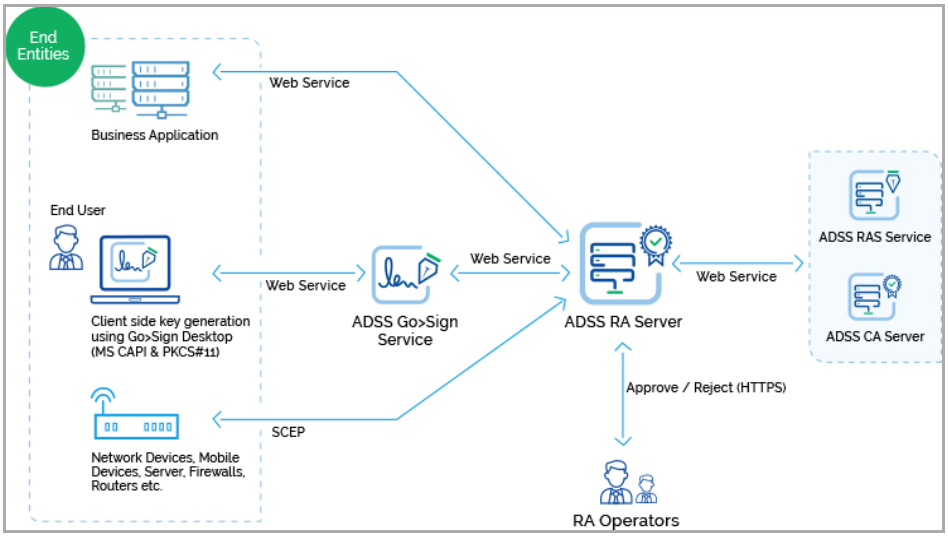
The ADSS Server RA Service provides the ability to:

- Manage the RAS/SAM users
- Register and Revoke Certificates
- To retrieve the certificates issued asynchronously

These operations are accessible either:

- directly via an Ascertia proprietary XML protocol. Revoke certificate is only supported in Ascertia proprietary XML protocol.
- using the Simple Certificate Enrollment Protocol (SCEP). Recover Certificate operation is only supported in SCEP interface
- via a Client API.

Registered Business Applications send requests to ADSS Server, referring to a particular RA Profile, and receive responses. Normally most of the RA related parameters do not need to be sent in the request as they are already configured in the RA Profile.



14.1 RA Use Cases and Ascertia Protocol Schema

Various RA use cases and Ascertia proprietary protocol schemas for RA are discussed in section 21.

14.2 RA Profiles

The ADSS RA Service requires that RA Profiles are defined at ADSS Server. These profiles specify which ADSS CA server and profile will be used to issue certificates, the key length and key type to be used, the certificate validity period, any default distinguished name parameters (e.g. country name, organisational unit etc.).

Refer to the following link in the online admin guide for an explanation of RA Profile settings:

Step 2 - RA Profiles (ascertia.com)

14.3 The RA Service API

The RA Service API is provided as part of the ADSS Client SDK and consists of a Registration Request, Registration Response , ScepRequest and ScepResponse classes.

14.3.1 Registration Request Class

The following constructor is used to build the initial Registration Request.

```
RegistrationRequest registrationRequest = new RegistrationRequest
(clientID, requestType, certificateAlias );
```

The `clientID` identifies the business application that is making the call. This `clientID` must already be registered at ADSS Server Client Manager.

The `requestType` identifies one of the following available services i.e.:

```
REQUEST_TYPE_CREATE_CERTIFICATE     Certificate Creation
REQUEST_TYPE_RENEW_CERTIFICATE      Certificate Renew
REQUEST_TYPE_REKEY_CERTIFICATE      Certificate Rekey
REQUEST_TYPE_DELETE_CERTIFICATE     Certificate Delete
REQUEST_TYPE_IMPORT                 Certificate Import
REQUEST_TYPE_REVOKE                 Certificate Revocation
REQUEST_TYPE_STATUS                 Certificate Status
REQUEST_TYPE_REGISTER_USER          User Registration
REQUEST_TYPE_UPDATE_USER            User Update
REQUEST_TYPE_DELETE_USER            User Delete
REQUEST_TYPE_GET_USER               User Get
REQUEST_TYPE_GET_USER_DEVICES       User Devices Get
REQUEST_TYPE_DELETE_USER_DEVICE     User Device Delete
REQUEST_TYPE_GET_USER_CERTIFICATES  User Certificates Get
REQUEST_TYPE_GET_USERS              Users Get
REQUEST_TYPE_CHANGE_PASSWORD        User Password Change
REQUEST_TYPE_RECOVER_PASSWORD       User Password Recover
REQUEST_TYPE_CONFIRM_RECOVER_PASSWORD User Password Confirm Recover
REQUEST_TYPE_CHANGE_EMAIL           User Email Change
REQUEST_TYPE_CONFIRM_CHANGE_EMAIL   User Email Confirm Change
REQUEST_TYPE_CHANGE_MOBILE          User Mobile Change
REQUEST_TYPE_CONFIRM_CHANGE_MOBILE  User Mobile Confirm Change
```

14.3.2 Registration Request methods

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

```
ToString, WriteTo, Send (overridden), SetProxy, SetRequestID,
SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion,
SetSSLClientCredentials, SetTimeout, SetVerifyResponse.
```

In addition, the following methods are specific to the Registration Request class:

Registration Request Method	Purpose
`SetProfileId (string)`	This specifies the registration profile Id that will be used by RA Service to serve the request.
`SetPkcs12Password (string)`	Specifies the PKCS#12 password. This method is only used if the key pair is generated at server and held in software.
`SetSubjectDN (string)`	Specifies the Subject DN of the requested certificate. Multiple values are separated by comma (,). Possible values are: `CN, OU, O, L, S, E, C, SN, B, ST, P, EVL, EVS and EVC`
`SetCertAlias (string)`	Specifies the certificate alias that is used as reference at ADSS RA Server. Moverover, it would be used to identify the user's signing certificates on the RAS/SAM servers.

	It's mandatory parameter in the user enrollment at RAS/SAM. For RAS/SAM requests (allowed characters are a-zA-Z0-9_.@-)
`SetPKCS10 (byte[])`	Specifies the pkcs#10 certificate requests if a key pair is generated at client.
`SetValidityPeriod`	Specify the validity period for the certificate life to be created or renewed e.g. 12
`SetValidityUnit`	Specify the validity period unit of the certificate life in `'MINS','HOURS','DAYS','MONTHS'` or `'YEARS'`
`SetRevocationReason (string)`	Specifies the Revocation Reason.
`SetOnholdInstructionCode (onHoldInstructionCode)`	Specifies the 'on hold' instruction code.
`SetChallengePassword (string)`	Specifies the challenge password for device certificate.
`SetEmailAddress (string)`	Specifies the email address of requester. Moreover, the email address would be used to register user at RAS/SAM. Later, it would be used to send OTPs by RAS. It is a mandatory parameter in the user registration request. For RAS/SAM requests (max. 100 characters)
`SetUserName (string)`	Specifies the user name that will be displayed on mobile device. It is an optional parameter in the user registration request. For RAS/SAM requests (max. 50 characters)

`addSubjectAlternativeName (string, string`	It is used to add subject alternative name extension in X.509 certificate. The first parameter specifies the name while the second parameter specifies the value. The possible values for first parameter are:
	`rfc822Name`
	`dNSName`
	`iPAddress`
	`uniformResourceIdentifier`
	`directoryName`
	`otherName`
	This method can be called multiple times in order to add multiple names in subject alternative name extension.

`addCustomExtension(string, string)`	It is used to add custom extension in X.509 certificate. The first parameter defines the OID while the second parameter specifies the value of respective custom OID extension. For example: First parameter 1.3.6.1.4.1.59382.3.2 Second parameter = GERALD This method can be called multiple times in order to add multiple custom extension.

`SetTransactionID (string)`	Specifies the transaction Id of the certificate request to find the status of certificate request in asynchronous mode.
`SetUserID (string)`	User ID to identify a user at the RAS/SAM server. Later, its used to authenticate a user, fetch user devices and certificates info. It is a mandatory parameter in the user registration request. For RAS/SAM requests (max. 50 characters and allowed characters are a-zA-Z0-9_.@-)
`SetUserStatus (string)`	Specifies the user status
`SetUserNewEmail (string)`	Specifies the new email address of a user to replace the old email address For RAS/SAM requests (max. 100 characters)
`SetUserNewMobile (string)`	Specifies the user new mobile number to replace the old user mobile number
`SetUserOldPassword (string)`	Specifies the user old password
`SetUserNewPassword (string)`	Specifies the new password of a user to replace the old password For RAS/SAM requests (max. 50 characters)
`SetMobileNumber (string)`	It uses in user registration at RAS/SAM. Later, it would be used to send OTPs by RAS. It is a mandatory parameter in the user registration request. For RAS/SAM requests (max. 100 characters)
`SetCertificate (byte[])`	Specifies the certificate to be import after certification from the external CA. It will be in uses when the RA configured to certify the key pairs asynchronously.
`SetCertificate (string)`	Specifies the certificate path to be import after certification from the external CA. It will be in uses when the RA configured to certify the key pairs asynchronously
`SetCertificateID (string)`	Specifies the certificate ID
`SetDeviceID (string)`	Specifies the device ID to delete the user device
`SetEmailOtp (string)`	Specifies the email OTP to update the user mobile no, user email address and recover user password
`SetMobileOtp (string)`	Specifies the mobile OTP to update the user mobile no, user email address and recover user password

14.3.3 Other Registration Request Methods

Some other Registration Request methods are defined such as those for communication purposes (e.g. use of proxy, timeouts etc.):

```
SetProxy, SetRequestID, SetRequestRetries, SetTimeout.
```

For these and others refer to the JavaDoc and Sandcastle documentation.

14.3.4 Sending the Registration Request

Once the registration request message has been fully built using the above methods, it can be sent to ADSS Server using the following call:

```
var registerationResponse =
(RegiserationResponse)registerationRequest.Send(URL);
```

The `URL`, is that of the RA Service e.g.

http://machine-name:8777/adss/ra/cri

For a mutually authenticated TLS request, it is:

https://machine-name:8779/adss/ra/cri

14.3.5 Example of a Registration Request using the Ascertia XML protocol

```
// Constructing RA request to create certificate
RegistrationRequest obj_registrationRequest = new
RegistrationRequest("samples_test_client",
RegistrationRequest.REQUEST_TYPE_CREATE_CERTIFICATE, certAlias);
obj_registrationRequest.setRequestId("create-request-001");
obj_registrationRequest.setProfileId("adss:ra:profile:001");
obj_registrationRequest.setUserName("sample_user");
obj_registrationRequest.setEmailAddress("sample@ascertia.com");
obj_registrationRequest.setSubjectDN("CN=Sample,OU=Dev,O=ASC,C=GB");
obj_registrationRequest.setPkcs12Password("password");
// Sending the above constructed request to the ADSS RA service
RegistrationResponse obj_registrationResponse =(RegistrationResponse)
obj_registrationRequest.send("http://localhost:8777/adss/ra/cri");
```

14.3.6 Example of a Certificate Renew Request

```
// Creating request to Renew Certificate
RegistrationRequest obj_registrationRequest = new
RegistrationRequest("samples_test_client",
    RegistrationRequest.REQUEST_TYPE_RENEW_CERTIFICATE, "certAlias");
obj_registrationRequest.SetRequestID("Rekey-request-01");
obj_registrationRequest.SetProfileID("adss:ra:profile:001");
obj_registrationRequest.SetSubjectDN("CN=Sample,OU=Dev,O=ASC,C=GB ");
obj_registrationRequest.SetPkcs12Password ("password");

// Sending the above constructed request to the ADSS server
RegistrationResponse obj_registrationResponse = (RegistrationResponse)
    obj_registrationRequest.Send("http://localhost:8777/adss/ra/cri");
```

14.3.7 Example of a Certificate Rekey Request

```
// Creating request to Rekey Certificate
RegistrationRequest obj_registrationRequest = new
RegistrationRequest("samples_test_client",
    RegistrationRequest.REQUEST_TYPE_REKEY_CERTIFICATE, "certAlias");
obj_registrationRequest.SetRequestID("Rekey-request-01");
obj_registrationRequest.SetProfileID("adss:ra:profile:001");
obj_registrationRequest.SetSubjectDN("CN=Sample,OU=Dev,O=ASC,C=GB ");
obj_registrationRequest.SetPkcs12Password ("password");

// Sending the above constructed request to the ADSS server
RegistrationResponse obj_registrationResponse = (RegistrationResponse)
```

```
    obj_registrationRequest.Send("http://localhost:8777/adss/ra/cri");
```

14.3.8 Example of a Revocation Request using the Ascertia XML protocol

```
// constructing RA request to revoke certificate
RegistrationRequest obj_registrationRequest = new
RegistrationRequest("samples_test_client",
RegistrationRequest.REQUEST_TYPE_REVOKE, certAlias);
obj_registrationRequest.setRequestId("revoke-request-001");
obj_registrationRequest.setProfileId("adss:ra:profile:001");
obj_registrationRequest.setRevocationReason(RegistrationRequest.
REVOCATION_REASON_AFFILIATIONCHANGED);
// Sending the above constructed request to the ADSS RA service
RegistrationResponse obj_registrationResponse = (RegistrationResponse)
obj_registrationRequest.send("http://localhost:8777/adss/ra/cri");
```

14.3.9 Example of a Certificate Status Request

```
// Constructing certificate status request
RegistrationRequest obj_registrationRequest = new
RegistrationRequest("samples_test_client",
RegistrationRequest.REQUEST_TYPE_STATUS);
obj_registrationRequest.setRequestId("status-request-001");
obj_registrationRequest.setProfileId("adss:ra:profile:001");
obj_registrationRequest.setTransactionID("TransactionID");
// Sending the above constructed request to the ADSS server
RegistrationResponse obj_registrationResponse = (RegistrationResponse)
obj_registrationRequest.send("http://localhost:8777/adss/ra/cri");
```

14.3.10 Example of a Profile Info Request

```
// Creating request for RA Profile Info
RegistrationRequest obj_registrationRequest = new
RegistrationRequest("samples_test_client",
    RegistrationRequest.REQUEST_TYPE_GET_PROFILE_INFO, "adss:ra:profile:001");
obj_registrationRequest.SetRequestID("Get_profile_request01");

// Sending the above constructed request to the ADSS server
RegistrationResponse obj_registrationResponse = (RegistrationResponse)
    obj_registrationRequest.Send("http://localhost:8777/adss/ra/cri");
```

14.3.11 Registration Response Class

The following methods of the Registration Response class are inherited from the generic Response and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetRequestID, GetSigningCertificates, GetStatus, IsSuccessful.`

In addition, the following methods are specific to the Registration Response Class:

Registration Response Method	Purpose
`GetCertificate() returns X509Certificate`	Returns the X509 certificate object.
`GetProfileId () returns string`	Returns the RA profile Id used by RA Service to process this request.
`GetTransactionId () returns string`	Returns the transaction Id of the corresponding request.
`GetPKCS7() returns byte[]`	Returns the PKCS#7 certificate chain.
`GetPKCS12 () returns byte[]`	Returns the PKCS#12.
`PublishCertificate(string /Stream)`	Publishes the certificate to the specified path or stream.
`PublishPKCS12(string /Stream)`	Publishes the PKCS#12 data to the specified path or stream.
`PublishPKCS7(string /Stream)`	Publishes the PKCS#7 data to the specified path or stream.
`GetEmail() returns string`	Returns the user email address.
`GetUserCertificates() returns List<UserCertificateType>`	Returns the user all certificates.
`GetUserDevices() return List<UserDeviceType>`	Returns the user all devices.
`GetUsers() return List<UserType>`	Returns the all users.
`GetProfileInfo() returns ProfileInfoType`	Returns the ProfileInfoType object.

14.3.12 ScepRequest

Scep request class is used to send registration request to RA Service for a device certificate. The following constructor is used to build the registration request.

```
var registrationRequest = new ScepRequest (publicKey, privateKey,
encryptionCertificate,  subjectDN, transactionId, senderNonce);
```

The `publicKey` identifies the public key of requested certificate, `privateKey` identifies the private key of the requested certificate, `encryptionCertificate` identifies the certificate to encrypt the request and `transactionId` identifies the unique identifier of the request. The `privateKey` is also used to sign the Scep request.

14.3.13 ScepRequest Methods

The following methods are inherited from the generic Request and Message classes and are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, Send (overridden), SetProxy, SetRequestID, SetRequestRetries, SetSigningCredentials, SetSigningMode, SetSoapVersion, SetSSLClientCredentials, SetTimeout, SetVerifyResponse.`

In addition, the following methods are specific to the SCEP Request class:

Scep Request Method	Purpose
`SetDigestAlgorithmOID (String)`	Specifies the digest algorithm OID.

`SetSignatureAlgorithm (String)`	Specifies the signature algorithm for signing SCEP request.
`SetRequestMethod (String)`	Specifies the request method GET/POST.
`SetChallengePassword (String)`	Specifies the challenge password for device certificate.

14.3.14 Other ScepRequest Methods

Some other Scep Request methods are defined such as those for communication purposes (e.g. use of proxy, timeouts etc.):

`SetProxy, SetRequestID, SetRequestRetries, SetTimeout.`

For these and others refer to the JavaDoc and Sandcastle documentation.

14.3.15 Sending the ScepRequest

Once the scep request message has been fully built using the above methods, it can be sent to ADSS Server using the following call:

```
var registrationResponse = (ScepResponse) registrationRequest.Send(URL);
```

The `URL` is that of the RA Service e.g.

http://machine-name:8777/adss/ra/scep

For a mutually authenticated TLS request, it is:

https://machine-name:8779/adss/ra/scep

14.3.16 Example of a Registration Request using the SCEP protocol

```
// Constructing RA request to create certificate
ScepRequest obj_scepRequest = new ScepRequest("publicKey","privateKey",
"encryptionCertificate","subjectDN", "transactionId", "senderNonce");
obj_scepRequest.setRequestMethod(ScepRequest.POST);
obj_scepRequest.setChallengePassword("password");
// Sending the above constructed request to the ADSS server
ScepResponse obj_scepResponse=(ScepResponse)
obj_scepRequest.send(http://localhost:8777/adss/ra/scep");
```

14.3.17 Example of a Certificate Retrieval Request using the SCEP protocol

```
// Constructing RA request to recover certificate
ScepRequest obj_scepRequest = new ScepRequest("publicKey","privateKey",
"encryptionCertificate","issuerName", "serialNumber", "transactionId",
"senderNonce");
obj_ scepRequest.setRequestMethod(ScepRequest.POST);
// Sending the above constructed request to the ADSS server
ScepResponse obj_scepResponse=(ScepResponse)
obj_scepRequest.send((http://localhost:8777/adss/ra/scep");
```

14.3.18 Example of a get CA Certificate Request using the SCEP protocol

```
// Constructing RA request to get CA certificate

ScepRequest obj_scepRequest=new ScepRequest(ScepRequest.OPERATION_TYPE_GETCACERT);

// Sending the above constructed request to the ADSS server

ScepResponse    obj_scepResponse=(ScepResponse)
obj_scepRequest.send(http://localhost:8777/adss/ra/scep");
```

14.3.19 Example of a get CA Capabilities Request using the SCEP protocol

```
// Constructing RA request to get CA capabilities

ScepRequest obj_scepRequest=new ScepRequest(ScepRequest.OPERATION_TYPE_GETCACAPS);

// Sending the above constructed request to the ADSS server

ScepResponse    obj_scepResponse=(ScepResponse)
obj_scepRequest.send(http://localhost:8777/adss/ra/scep");
```

14.3.20 ScepResponse Class

The following methods of the ScepResponse class inherited from the generic Response and Message classes. These classes are described in section 3 as well as in the JavaDoc and Sandcastle class documentation:

`ToString, WriteTo, ContainsException, GetErrorCode, GetErrorMessage, GetException, GetRequestID, GetSigningCertificates, GetStatus, IsSuccessful.`

In addition, the following methods are specific to the Scep Response Class:

SCEP Request Method	Purpose
`GetCaRaCertificate() returns X509Certificate`	Returns the X509 certificate object.
`GetSigningTime() returns DateTime`	Returns the signing time of response.
`GetRecipientNonce () returns byte[]`	Returns the recipient nonce value.
`GetIssuedCerts () returns X509Certificate[]`	Returns the issued X509 certificate array.
`GetRequestId() returns string`	Returns the Request ID of the certification request.
`GetSenderNonce () returns byte[]`	Returns the sender nonce value.
`GetFailInfo () returns int`	Returns the fail info value.
`GetMessageType () returns int`	Returns message type value i.e. application/x-pki-message, application/x-x509-ca-cert etc.
`GetPkiStatus () returns int`	Returns PKI status value.
`GetTransactionID () returns string`	Returns transaction id of request.
`GetPKCS7 () return byte[]`	Returns PKCS#7 bytes.
`GetCACaps () returns string`	Returns CA capabilities in string.
`PublishCertificate(string /Stream)`	Publishes the certificate to the specified path or stream.
`PublishPKCS7(string /Stream)`	Publishes the PKCS#7 data to the specified path or stream.

PublishIssuedCerts (string)	Publishes the issued certificates to the specified path.

14.4 RA Service Sample Code

Java sample code is provided as part of the ADSS Client SDK and can be used to make RA Service requests and to process the RA Service responses.

The Java API provides the required classes under the package:

```
com.ascertia.adss.client.api.ra
```

14.4.1 Java API Sample Code

The following sample programs demonstrate how the Java API can be used to send a Registeration request and process the response:

```
samples/src/com/ascertia/adss/client/samples/ra/CreateCertificate.java
samples/src/com/ascertia/adss/client/samples/ra/RevokeCertificate.java
samples/src/com/ascertia/adss/client/samples/ra/StatusCertificate.java
samples/src/com/ascertia/adss/client/samples/ra/RegisterUser.java
samples/src/com/ascertia/adss/client/samples/ra/UpdateUser.java
samples/src/com/ascertia/adss/client/samples/ra/DeleteUser.java
samples/src/com/ascertia/adss/client/samples/ra/GetUser.java
samples/src/com/ascertia/adss/client/samples/ra/GetUsers.java
samples/src/com/ascertia/adss/client/samples/ra/GetUserDevices.java
samples/src/com/ascertia/adss/client/samples/ra/GetUserCertificates.java
samples/src/com/ascertia/adss/client/samples/ra/ChangePassword.java
samples/src/com/ascertia/adss/client/samples/ra/RecoverPassword.java
samples/src/com/ascertia/adss/client/samples/ra/ConfirmRecoverPassword.ja
va
samples/src/com/ascertia/adss/client/samples/ra/ChangeEmail.java
samples/src/com/ascertia/adss/client/samples/ra/ConfirmChangeEmail.java
samples/src/com/ascertia/adss/client/samples/ra/ChangeMobile.java
samples/src/com/ascertia/adss/client/samples/ra/ConfirmChangeMobile.java
samples/src/com/ascertia/adss/client/samples/ra/DeleteDevice.java
samples/src/com/ascertia/adss/client/samples/ra/GetRaProfileInfo.java
```

A precompiled and ready to run version of the above sample programs can be found at:

```
samples/bin/RACreateCertificate.bat
samples/bin/RARevokeCertificate.bat
samples/bin/RAStatusCertificate.bat
samples/bin/RARegisterUser.bat
samples/bin/RAUpdateUser.bat
samples/bin/RADeleteUser.bat
samples/bin/RAGetUser.bat
samples/bin/RAGetUsers.bat
samples/bin/RAGetUserDevices.bat
samples/bin/RAGetUserCertificates.bat
samples/bin/RAChangePassword.bat
samples/bin/RARecoverPassword.bat
samples/bin/RAConfirmRecoverPassword.bat
samples/bin/RAChangeEmail.bat
samples/bin/RAConfirmChangeEmail.bat
samples/bin/RAChangeMobile.bat
samples/bin/RAConfirmChangeMobile.bat
samples/bin/RADeleteUserDevice.bat
```

```
samples/bin/RAProfileInfo.bat
```

14.5 Error Codes

ADSS RA Service returns the following error codes in case of any failure:

Error Code	Error Message
54001	RA profile does not exist or marked inactive.
54002	RA service is stopped.
54003	RA service not enabled in license.
54004	RA service license has expired.
54005	RA service is not enabled in system.
54006	RA request must be signed.
54007	RA request signature verification failure.
54008	RA request is not schema compliant.
54009	RA profile not found.
54010	RA service not allowed.
54011	An internal error occurred while processing the request - see the RA service debug logs for details.
54012	Authentication failed.
54013	RA default profile does not exist or marked inactive.
54014	RA profile is inactive.
54015	RA originator authentication failed.
54016	Device challenge password does not match.
54017	Device subject DN does not match.
54018	CA server address is not accessible.
54019	CA server address is not configured properly.
54020	Subject DN is invalid.
54021	Invalid request status.
54022	Invalid transaction ID.
54023	Certificate does not exist.
54024	Certificate chain does not exist.
54025	Required parameter(s) are missing.
54026	PKCS#10 is not compatible with profile.
54027	Certificate alias already exists.
54028	User name missing in request.
54029	User email address missing in request.
54030	Certificate alias missing in request.
54031	Subject DN missing in request.

54032	PFX password missing in request.
54033	Certificate alias length exceeds the limit.
54034	Default profile not configured and neither found in request.

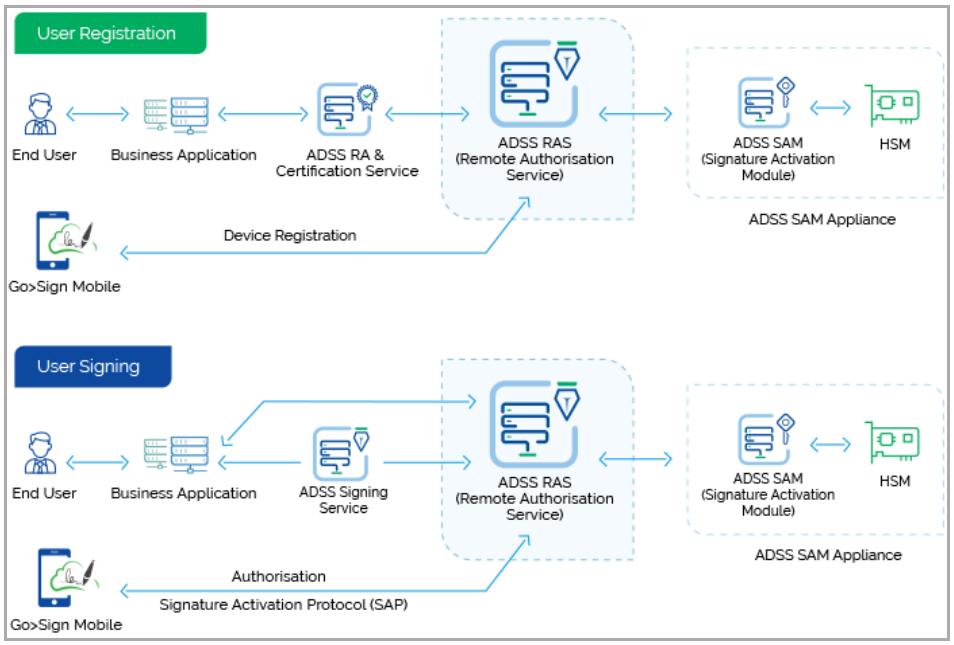
15 ADSS RAS Service

ADSS Remote Authorisation Signing (RAS) Service provides the capability to shield SAM from outside world and act as a bridge between business/Go>Sign Mobile Application and the SAM Service. It provides the required API interface for business applications to register users, send hash signing request, checking the status of pending signing requests and getting the signed hash (i.e. PKCS#1 signature). It also provides the required API interfaces for the Go>Sign Mobile app to allow users login to the app after authentication via SMS and EMAIL OTPs, registering the mobile device with authorisation public key, sending push notifications, fetching the authorisation request and sending the signed authorisation request (i.e. Signature Activation Data – SAD).

RAS Service acts as a RSSP for Signing Service implementing Adobe CSC interfaces.

These operations are accessible either:

- Via ADSS RA, Certification and Signing Service
- directly via an Ascertia proprietary JSON protocol through RESTful APIs.



15.1 RAS Profiles

The ADSS RAS Service requires that RAS Profiles are defined at ADSS Server. These profiles specify ADSS SAM Service to get user authorisation, optionally mutual authentication and define user authentication mechanism either basic (user ID and password) or SAML assertion.

Refer to the following link in the online admin guide for an explanation of RAS Profile settings:

Step 1 - Configuring RAS Profile (ascertia.com)

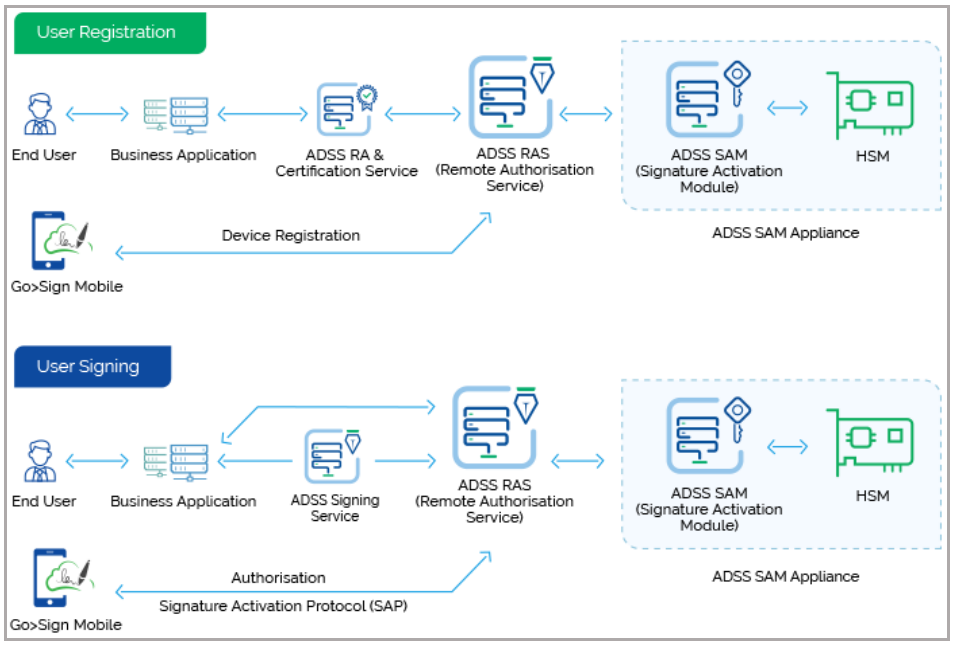
Note: In order to learn how the business applications can integrate ADSS RAS Service and utilize it features, read the "**ADSS-RAS-Developers-Guide.pdf**" shipped within the ADSS Client SDK package.

Refer to the following link in the online admin guide for an explanation of RAS Service:

ADSS RAS Service (ascertia.com)

16 ADSS SAM Service

ADSS SAM Service which supports Remote Authorised Signing for end users. eIDAS Regulation compliant solution against EN 419 241 – 2. This operates within the same framework but supports use of a standard PKCS#11 HSM. It provides the capability to manage users and their signing keys. It provides the required API interfaces to manage users, signing keys, authorised devices, authorization requests, signing requests, getting the signed hash (i.e. PKCS#1 signature) and their current statuses.



16.1 SAM Profiles

The ADSS SAM Service requires that SAM Profiles are defined at ADSS Server. These profiles specify user signing key type, key length and crypto mode (Software, HSM or Cloud). It defines PKCS#1 signature generation mechanism. It also defines the user authorization key type, length and some more constraints on the user devices.

Refer to the following link in the online admin guide for an explanation of SAM Profile settings:

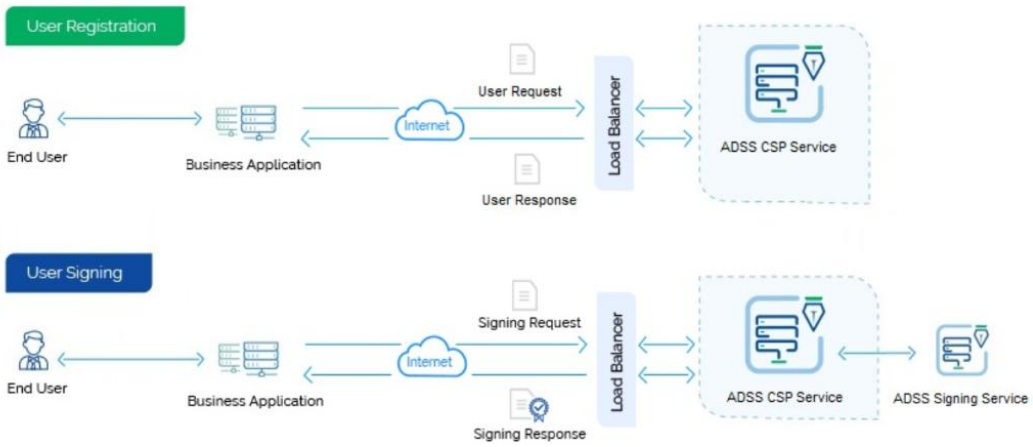
Step 2 - Configuring SAM Profile (ascertia.com)

Note: Business applications can use ADSS SAM Service via ADSS RAS Service. Follow the ADSS RAS Service section for integration.

17 ADSS CSP Service

ADSS CSP (Cryptographic Service Provider) is an ADSS Server service which allows applications to create user accounts and sign data. Currently we have a product named Virtual CSP (Windows plugin to initiate signing process), it is a windows plugin developed in .Net/C++ technology. It is a plugin for windows letting users sign emails / documents using keys/certificates held at server. Currently it is working with DigitalSign developed CSP product, now we are going to have our own service named CSP Service to work with VCSP to complete the signature process using keys held at ADSS Server. The CSP Service will also have an interface for business applications like WebRA.

The CSP Service operations are accessible through an Ascertia proprietary JSON protocol through RESTful APIs.



17.1 CSP Profiles

The ADSS CSP Service requires that CSP Profiles are defined at ADSS Server. These profiles specify ADSS Signing Service forcryptographic operations, optionally mutual authentication and define user password validation mechanism.

Refer to the following link in the online admin guide for an explanation of CSP Profile settings:

Step 3 - Configuring CSP Profile (ascertia.com)

Note: In order to learn how the business applications can integrate ADSS RAS Service and utilize it features, read the "**ADSS-CSP-Developers-Guide.pdf**" shipped within the ADSS Client SDK package.

Refer to the following link in the online admin guide for an explanation of CSP Service:

ADSS CSP Service (ascertia.com)

18 Utility Classes

Currently there are two utility classes:

- AuthorisationData Class (to create an Authorisation Data XML file for creating authorised signing requests), and
- Util Class (for document creation, writing, reading and encoding)

The Java API provides the required classes under the package:

`com.ascertia.adss.client.api.util and`

The .Net API provides the required classes under the namespace:

`Com.Ascertia.ADSS.Client.API.Util.`

18.1 AuthorisationData Class

The Authorisation Data class is used to prepare an XML Authorisation file which is later signed by one or more individuals to authorise the use of a server held signing key.

The signed Authorisation files are attached to the signing request as explained in section 4.4.1.

For an explanation of authorisation profiles refer to the online admin guide:

Authorisation Profiles (ascertia.com)

Authorisation profiles may be selected for use in Signing profiles – again see the online admin guide for details:

Advanced Settings (ascertia.com)

The following is a list of methods of the Authorisation Data class:

Authorisation Data Method	Purpose
`AuthorisationData( string originatorId,byte[]/Stream/string data, int contentType)`	This first step when creating the XML Authorisation file is to construct an initial Authorisation Control object. As input, this takes the originator ID, it's the client ID that must be registered within ADSS Server. Origiantor ID is mandatory and cannot be null or empty otherwise the server would refuse the authorisation file. Then it takes the first document (or document hash) for which signing is going to be authorised. The document or hash can be provided as a byte array, stream or file path. If supplied as a document, it will be hashed later during publication `(PublishAuthorisationData)` using the algorithm supplied by the `SetHashAlgorithm` method. The content type is one of: - `CONTENT_TYPE_DOCUMENT` - `CONTENT_TYPE_HASH`
`AddDocument( byte[]/Stream/string data)`	Adds a further document or document hash into the Authorisation Control object. This has to have the same content type as specified in the constructor above.
`AddMetaDataElement(string key, string value)`	Adds a meta data element into the Authorisation Control object. This can be any information that will be used to help identify the Authorisation Control file or the documents being signed.

`ComputeHash(string hashAlgorithm, Stream stream)` `returns byte[]`	Calculates a hash of an input Stream using the indicated hash algorithm. The hash algorithm can be one of: - `HASH_ALGO_SHA1` - `HASH_ALGO_SHA224` - `HASH_ALGO_SHA256` - `HASH_ALGO_SHA384` - `HASH_ALGO_SHA512` This method can be used to create a document hash instead of relying upon the `GetAuthorisationData` method.
`GetAuthorisationData() returns byte[]`	Creates the Authorisation XML object from the Authorisation Control object. The hashing of the documents in the Authorisation Control object takes place at this time (unless they are already supplied as hashes). The resultant DocumentDigest element contains the concatenated hash of the documents that will be later signed by the Signing Service and therefore have to be matched up with the authorisation signatures.
`PublishAuthorisationData (string filePath /Stream stream)`	Publishes the Authorisation Data Xml to the specified output stream or file path.
`SetHashAlgorithm(string hashAlgorithm)`	Sets the hash algorithm for calculating hash of documents in the Authorisation Data object. The value can be one of: - `HASH_ALGO_SHA1` - `HASH_ALGO_SHA224` - `HASH_ALGO_SHA256` - `HASH_ALGO_SHA384` - `HASH_ALGO_SHA512`
`SetOriginatorID(string originatorID)`	Sets the originator ID. The originator ID must be set either through the constructors or using this method.
`GetOriginatorID()`	It returns the originator ID.
`SetValidFrom(string validFrom)`	Sets the start of the validity period of the authorisation control file. Validity period defines the time period during which an authorisation control file could be considered valid. Its an xml datetime of the format (YYYY-MM-DDThh:mm:ssZ)
`GetValidFrom()`	Returns the starting date of the validity period.
`SetValidTo(string validTo)`	Sets the end of the validity period of authorisation control file. After this time the authorisation control file would be rejected by server if presented for authentication. Its an xml datetime of the format (YYYY-MM-DDThh:mm:ssZ)
`GetValidTo()`	Return the end date of validity period.

The originator ID must be provided in the XML Authorisation control file otherwise the server would refuse the request. Morever, this originator ID must be the same that would be sent in the signing request.

The validity period is optional. If the validity period is provided in the authorisation control file, the server would validate this time period and reject the request if the validity period is expired.

or not yet started. Moreover, the Valid From and Valid To dates must be provided, absense of any of these dates would result in the failure and request would be rejected by the server.

18.1.1 Example of Creating an XML Authorisation Control File

This example makes use of the above class to build and publish the (unsigned) XML Authorisation Control file:

```
//Creating initial (in memory) Authorisation Control Object
AuthorisationData authData =
    new AuthorisationData("samples_test_client",inFile1, AuthorisationData.CONTENT_TYPE_DOCUMENT);
//Add a second file to be authorised
authData.AddDocument(inFile2);
//Use this hash algorithm
authData.SetHashAlgorithm(AuthorisationData.HASH_ALGO_SHA256);
//Add validity period if required
authData.SetValidFrom("2016-05-01T14:05:22.416+01:00");
authData.SetValidTo("2016-05-15T14:05:22.416+01:00");
//Add some meta-data elements
authData.AddMetaDataElement(key1, value1);
authData.AddMetaDataElement(key2, value2);

//Get Authorisation Control Data and publish as a file
byte[] authControlXml = authData.GetAuthorisationData();
authData.PublishAuthorisationData(authControlFilePath);
```

Later, after the XML Authorisation Control file has been signed by the authoriser(s) it is added into the signing request:

```
// ... after XML Authorisation file is signed by the authoriser(s)
// add into document signing request

byte[] signedAuthorisation = Util.ReadFile(signedAuthControlFilePath);
signRequest.AddSignedAuthorisation(signedAuthorisation);
```

18.2 Util Class

The Util class provides a number of general purpose methods for data conversion and file handling etc. The methods are all defined as `static` so no constructor is required in order to use them:

Utility Class Method	Purpose
`ConvertDateToString(DateTime date) returns string`	Converts a DateTime object to an ISO 8601 date format string.
`ConvertStringToDate(string date) returns DateTime`	Converts string (in ISO 8601 date format) to date.
`CreateDocument( XmlElement bodyElement) returns XmlDocument`	Reads an XmlElement object and treats this as the root element of a new Xml document (with org.w3c.dom.Document headers etc).
`CreateDocument(byte[] data) returns XmlDocument`	Reads a byte array and converts this to a new Xml document (with org.w3c.dom.Document headers etc).
`CreateDocument(InputStream is) returns XmlDocument`	Reads data from the given stream and converts this to a new Xml document (with org.w3c.dom.Document headers etc).
`CreateDocument(string a_strXML) returns XmlDocument`	Reads XML data from the given file path and converts this to a new Xml document (with org.w3c.dom.Document headers etc).

`CreateDocument() returns XmlDocument`	Creates a new (empty) Xml document (without org.w3c.dom.Document headers).
`SignDocument(Document,PrivateKey,X509Certificate[])`	Signs XML document using the provided private key and certificate chain.
`Decode(string encodedData) returns byte[]`	Decodes a base64 encoded string to a byte array.
`Encode( byte[] data) returns string`	Encodes a byte array to a base64 encoded string.
`FormatAsBase64String(string data)`	Formats the given string into base64 formatted string.
`DocumentToBytes(Document) returns byte[]`	Converts an XML Document object to a byte array.
`GetContentInfo( byte[] requestData, string OID) returns byte[]`	Constructs a 'Content Info' element.
`GetSignedData( string contentType, byte[] scvpReqData, AsymmetricKeyParameter privateKeyParam, X509Certificate signingCert, IX509Store certStore) returns byte[]`	Constructs a 'CMS Signed Data' object.
`GetDocumentBuilderFactory()`	Returns the DocumentBuilderFactory object
`GetTransformerFactory()`	Returns the TransformerFactory object
`ConvertXmlCalendar(Calendar)`	Converts the XMLGregorianCalendar to Calendar
`GetSubjectAttributeValue( string subject, string attribute) returns string`	Returns the 'Subject DN' attribute from the subject e.g. CN,OU,O.
`ConvertDateToString(Date)`	Converts the Date object into String.
`ConvertStringToDate(string)`	Converts the String into Date object.
`GetFormattedDN(string DN)`	Converts the given 'SubjectDN' into formatted 'SubjectDN'.
`IsArchiveDataSigned(Document)`	Checks whether the data to be archived is signed or not.
`RemoveSignatureElementFromXML(Document)`	Removes the signature element from the given XML document.
`ReadFile( string filePath) returns byte[]`	Reads the file from the given path and returns a byte array.
`ReadStream( Stream stream) returns byte[]`	Reads the file from the given file stream and returns a byte array.
`WriteToFile(byte[] data, string filePath)`	Writes the byte array to the provided file path.
`WriteToFile(XmlDocument xmlDoc, string filePath)`	Writes an Xml document to the provided file path.

`WriteToStream(byte[] data, Stream stream)`	Writes the byte array to the provided file stream.
`WriteToStream(XmlDocument xmlDoc, Stream stream)`	Writes an Xml document to the provided file stream.
`ConvertToDoc(byte[] data) returns byte[]`	Converts byte array data into XML document.

19 ADSS Signing Service - Use Cases and Schema

The following sections provide insight into the possibilities available for creating signatures with the Signing Service. Additionally, for the Ascertia proprietary protocols (e.g. Empty Signature Field Creation, Document Hashing and Assembly) a description of the XML protocol schemas is provided.

19.1 Server-side Document Signing

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

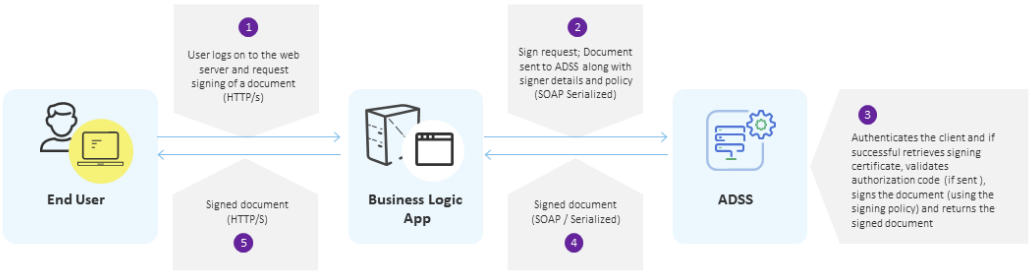
- Server-side signing – using the web services APIs
- Client-side signing – using the Go>Sign Desktop with browser based clients OR with thick client applications along with web service APIs for server-side processing

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 authorisation 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:



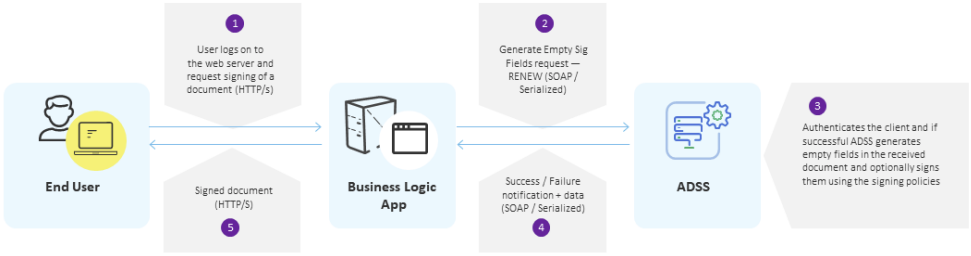
19.2 Client-Side Document Signing

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.

To perform client side document signing, Go>Sign Desktop and ADSS Go>Sign Service are used. For further details about how to use Go>Sign Desktop and ADSS Go>Sign Service refer to the Go>Sign Developers Guide in the ADSS Client SDK package. The source code for the Go>Sign demonstration application is also available in ADSS Client SDK.

19.3 Creating an Empty Signature Field in PDF Documents

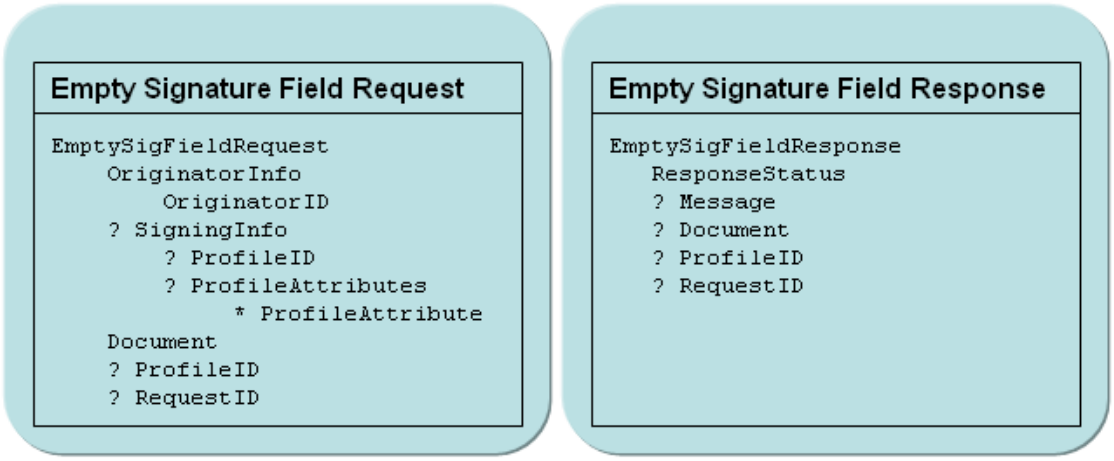
In some instances an empty signature field may need to be inserted within a PDF document – for example when creating additional signature fields prior to certifying (and thus locking) a PDF. To achieve this, a business application can call ADSS Server requesting it to create an empty signature field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



19.3.1 Empty Signature Field Creation

The purpose of the Empty Signature Field Creation service is to generate empty signature fields in a target PDF document. Optionally it is possible to use this service to sign or certify an empty signature field within the document.

The Empty Signature Field web service interface has a flexible XML schema, a summary of which is shown in the following diagram. For a detailed description refer to the XML Schema file **signing.xsd** provided within the ADSS Server software installation:



19.3.2 Empty signature field generation request

Setting these elements can be accomplished using the API calls described in section 4.6.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
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 (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 ADSS Server and if client-authenticated TLS is used then it must match the Common Name within the client TLS

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
		Certificate. If the OriginatorID is not registered, the request will be rejected.
	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 the ADSS Signing Service to sign the empty field(s) in the PDF. **ProfileAttribute (O) - (Container) - (Multiple)** Contains Profile attribute(s) and values to be overridden during processing by ADSS Server. The list of possible profile attributes that can be altered from the default values in the ADSS profile is: **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: 1 for TOP LEFT 2 for TOP RIGHT 3 for CENTER 4 for BOTTOM LEFT 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 the 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 The following figure shows a signature appearance with the above elements set. Note the contact information is only shown when reviewing the signature properties: **DOCUMENT_SIGNATURE_RELATIONSHIP (O) - (String)** This is not applicable.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
		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. **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 key 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. The Alias element must be set to sign a signature field although it is marked as optional in the protocol.
	Document (M) - (Base64) Contains the Base64 encoded document in which empty signature fields are to be created. **RequestID (O) - (String)** Contains a unique identifier assigned by the requesting application. This will be returned in the ADSS Server response.	
	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.	

19.3.3 Empty Signature Field Response

Retrieving **i**nformation from these elements can be accomplished using the API calls described in section 4.6.

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 at ADSS Server while processing the request.
	Document (O) - (Base64) Contains a PDF document formed after generating the empty field signature and optionally signing/certifying them.
	ResponseStatus (M) - (ResponseStatusEnum) Provides success or failure status information for the request. Possible values are: - `SUCCESS` - `FAILED`
	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.

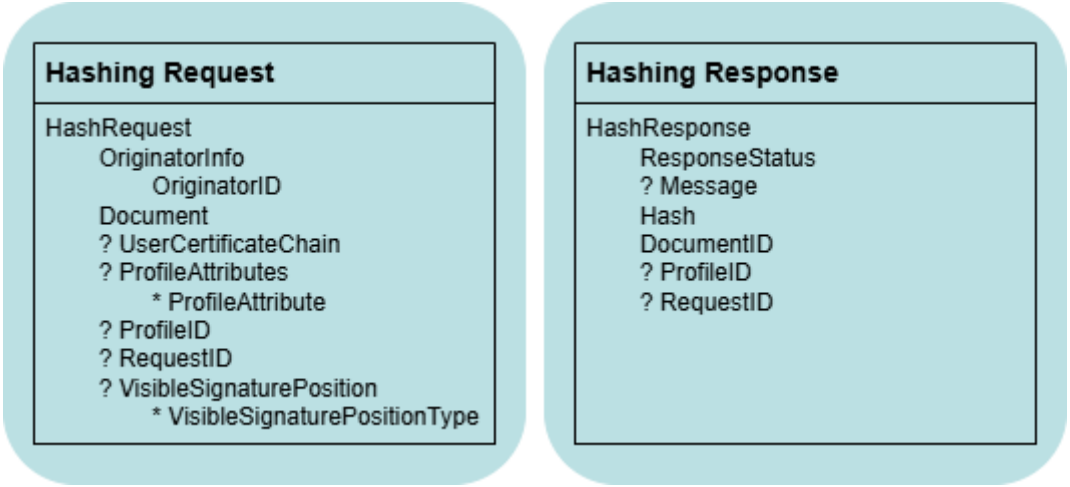
19.4 Document Hashing and Assembly

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 Go>Sign 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.

19.5 Document Hashing

An application can use the Document 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 hashing service requests and responses is shown in the following diagram. For a detailed description refer to the XML Schema file **signing.xsd** within the ADSS Client SDK download. The high level structure of the schema is as follows:



19.5.1 Hashing Request

Setting these elements can be accomplished using the API calls described in section 0.

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 six child elements named Document, ProfileAttrbutes, 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 ADSS Server and if client-authenticated TLS is used then it must match the Common Name within the client TLS 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 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:

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
		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 the 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: 1 for TOP LEFT 2 for TOP RIGHT 3 for CENTER 4 for BOTTOM LEFT 5 for BOTTOM RIGHT **SIGNED_BY (O) - (String)** Specifies the signer name who signed the document e.g. “John Doe” **PDF_PROTECTED_PASSWORD (O) - (String)** Specifies the permission passphrase that will be set on the PDF document to change the permissions of the document. **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 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 a 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 information is not visible on the document but can be seen when viewing the signature properties:
		DOCUMENT_SIGNATURE_RELATIONSHIP (O) - (String) This is not applicable.
	VisibleSignature Position (O) - (Container)	The values inside the **VisibleSignaturePositionType** is used to supply a list of parameters to fully specify a signature position and appearance. **FieldName (O) - (String)** Name of an existing blank signature field to be signed. **SignatureAppearanceId (O) - (String)** Signature appearance to be used while computing the hash e.g. default_sig_appearance.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
		PageNumber (O) - (Integer) Page number on which the signature should be placed (applicable only if the document allows visible signatures) e.g. 10. **X1 (O) - (Float)** X1 coordinate for visible signature position where the signature will be placed. **Y1 (O) - (Float)** Y1 coordinate for visible signature position where the signature will be placed. **X2 (O) - (Float)** X2 coordinate for visible signature position where the signature will be placed. **Note:** For iText version 7 or later, the X2 coordinate is interpreted as the width value. **Y2 (O) - (Float)** Y2 coordinate for visible signature position where the signature will be placed. **Note:** For iText version 7 or later, the Y2 coordinate is interpreted as the height value.

19.5.2 Hashing Response Element

Retrieving information from these elements can be accomplished using the API calls described in section 0.

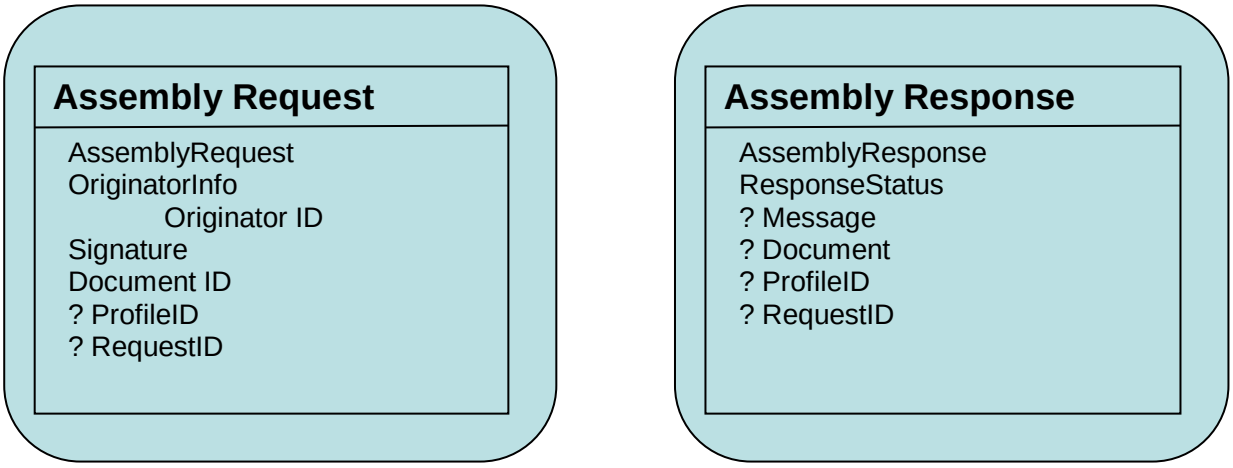
XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
HashResponse (M) - (Container)	This is the top level element of the Hashing Response message. It has three child elements named DocumentID, Message and Hash and three attributes named ResponseStatus, RequestID and ProfileID.
	Message (O) - (String) Contains the description of 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 Go>Sign Desktop. **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 the request was processed successfully or if it failed. Possible values are: SUCCESS FAILED
	RequestID (O) - (String) Contains the same unique identifier sent earlier in the HashingRequest > RequestID received by ADSS Server.
	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.

19.6 Signature Assembly

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

Currently the Assembly Service supports PKCS#7 and CAdES signatures to allow both the ISO Standard PDF signature and PAdES Baseline/Extended signature to be embedded in the PDF document. To use the Assembly Service, the application should first call the Hashing Service to get the Hash value and DocumentID, then have this Hash signed externally using the Go>Sign Desktop or another signing facility. The generated PKCS#7 or the CAdES signature plus DocumentID then has to be sent to ADSS Server for the PKCS#7 signature to be embedded within the document.

The Assembly Service interface uses a flexible XML Schema an overview of which as 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 **signing.xsd** provided within the ADSS Client SDK:



19.6.1 Assembly Request

Setting these elements can be accomplished using the API calls described in section 0.

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 and ProfileID. It is used to send the PKCS7 signature created by signing the hash received earlier from the ADSS Signing Service.	
	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 TLS is used then it must match the Common Name within the client TLS Certificate. If the OriginatorID is not registered, the request will be rejected.
	Signature (M) - (Base64) Contains the Base64 encoded PKCS#7 or CAdES 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 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.	

19.6.2 Assembly Response

Retrieving information from these elements can be accomplished using the API calls described in section 0.

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 and ResponseStatus and two attributes RequestID and ProfileID.
	Message (O) - (String) Contains the description of any error that occured within ADSS Server during the 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: SUCCESS FAILED
	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.

20 ADSS Certification Service – Use Case Overview

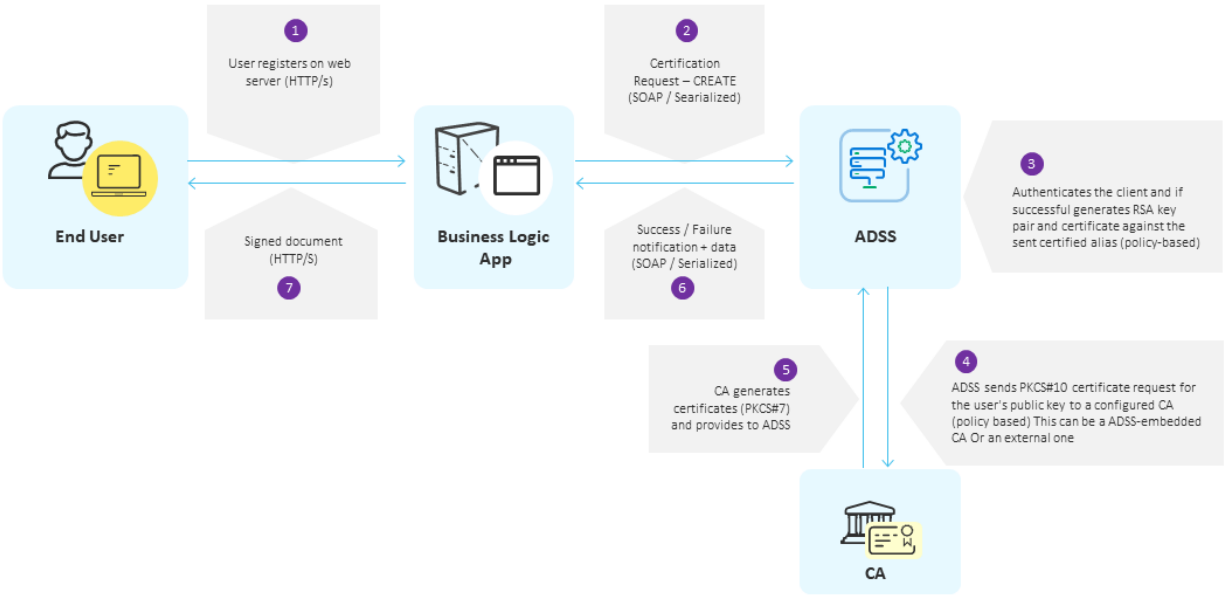
The following sections provide an insight into the possibilities available for certificate creation, renewal and key authorisation code update. Additionally, when the request is made using the Ascertia proprietary protocol (as opposed to CMC), descriptions of the certificate request/response XML protocol schemas are provided.

20.1 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 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 ADSS Server, e.g. for server-side signature creation.

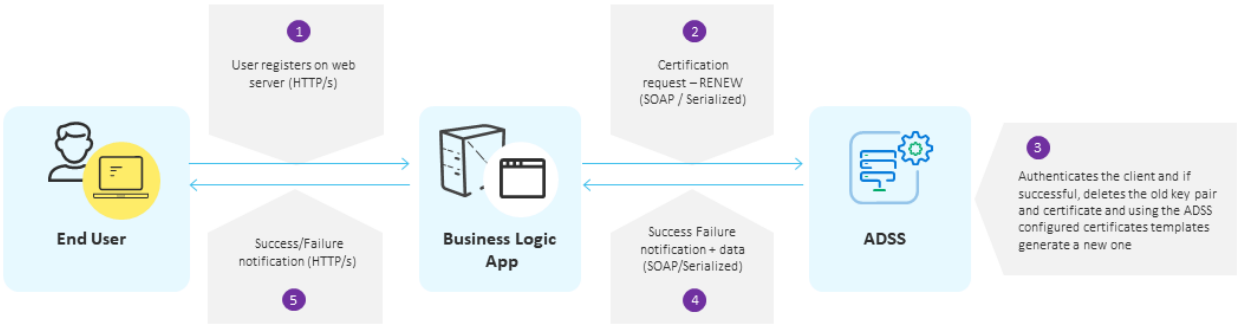
A high-level review of the process involving the user, business application, ADSS Server and the CA system is illustrated below:



20.2 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). ADSS Server also supports renewing of certificates using existing key pairs:

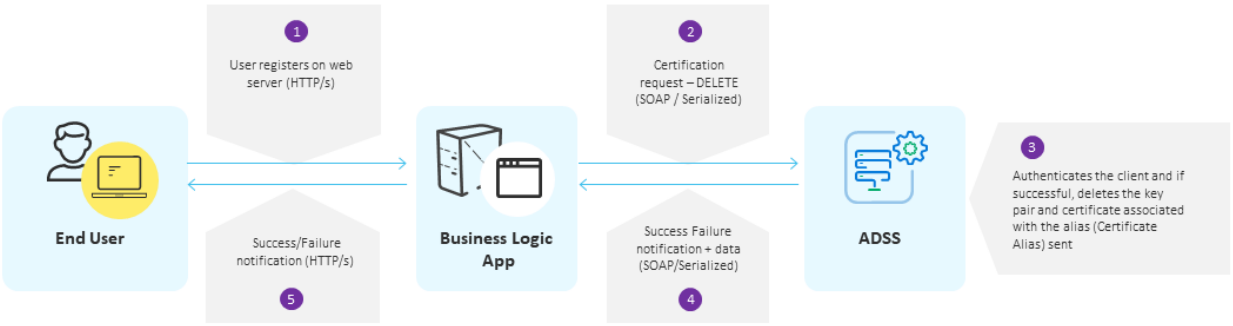


20.3 Retrieving Private Key (PKCS#12 object) and Certificate

The business application can also request ADSS Server 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. Go>Sign Desktop) can then open the PKCS#12 using a locally provided password to extract the Private Key and then use the Private Key for client side signing operations.

20.4 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:



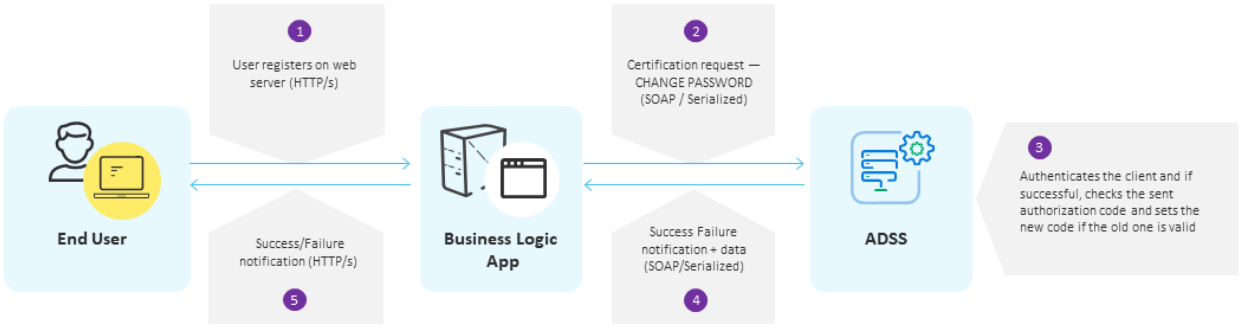
20.5 Changing an end-user key Authorisation Code

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 an 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 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. The reason for this is that when an HSM is used, PKCS#12 files are not required or maintained by ADSS Server:



20.6 Operation of the Certification Service

The Certification 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 **certification.xsd** provided within the ADSS Client SDK download.

Certification Request

```
CertificateRequest
    OriginatorInfo
        OriginatorID
    RequestType
    ? ProfileAttributes
        * ProfileAttribute
    CertificateInfo
         Alias
         Password
         ? NewPassword
    ? RespondWith
         * RespondItem
    ? ProfileID
    ? RequestID
```

Certification Response

```
CertificateResponse
   ResponseStatus
   ? Message
   ? ResponseData
      * ResponseItem
   ? ProfileID
   ? RequestID
```

20.6.1 Certification request

Setting these elements can be accomplished using the API calls described in section 6.4.

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 TLS is used then it must match the Common Name within the Client's TLS Certificate. If the OriginatorID is not registered, the request will be rejected.
	RequestType (M) - (Enumeration) Identifies the purpose of the request. The following set of values can be used for this element: 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 issued certificates and PKCS#12 object stored in the ADSS Server database REVOKE to revoke the certificate	
	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.	
	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. The value of this attribute is of the form adss:module:cert:001. If a profile is not identified then the ADSS Server default certification ProfileID is used.	
	ProfileAttributes (O) - (Container)	Used to customise 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). **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: **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, G, SN, T, OU, O, OI, C, L, S, E, SERIALNUMBER, B, ST, P, EVL, EVS and EVC can be over-ridden. **KEY_SIZE:** Size of the key pair to be generated in bits (1024, 2048, 3072 and 4096 for RSA; 192, 224, 256, 384 and 521 for ECDSA). **SECURITY_LEVEL:** Security level of key pair to be generated (2, 3 and 5 for ML-DSA; 1, 3 and 5 for ML-KEM; 1, 2, 3, 4, 5, 6 (or HIGHER) and 7 (or HIGHEST) for CLASSIC_MCELIECE).

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
		KEY_TYPE: Type of the key pair to be generated (RSA, ECDSA, ML-DSA, ML-KEM or CLASSIC_MCELIECE). **VALIDITY_PERIOD:** Validity period of certificate in numbers e.g. 12. **VALIDITY_UNIT** validity period unit of certificate lifetime in 'MINS', 'HOURS', 'DAYS', 'MONTHS' or 'YEARS' e.g MONTHS. **VALID_TO:** Expiry date of the certificate in string format "yyyy-MM-dd'T'HH:mm:ss" e.g. 2020-02-10T15:53:23 **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: • INTERNAL - specifies that the ADSS Server internal CA service is to be used • Any alias assigned to an external CA
	CertificateInfo (M) - (Container)	This element is used to provide the certificate specific data to ADSS Server. The list of certificate info elements which can be provided in certificate request is detailed below: **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 already used for this particular OriginatorID. **PKCS_10 (O) - (byte[])** The PKCS10 sent in the certification request contains the private key and public key used to certify the certificate at ADSS Certification Service. **PKCS7 (O) - (byte[])** The PKCS7 sent in the certification request is used to store the certificate at ADSS Certification Service. **PKCS12 (O) - (byte[])** The PKCS12 sent in the certification request is used to store the private key and certificate chain at ADSS Certification Service. **CERTIFICATE (O) - (byte[])** The certificate sent in the certification request is used to store the certificate at ADSS certification server or even use to delete or revoke the certificate if request type is "DELETE" or "REVOKE". **PASSWORD (M) - (String)** Password of the PFX or PKCS#12 private key file. If not provided in Certificate CREATE or RENEW request then the server assigns a secure password itself which can later be retrieved by specifying PASSWORD in the RespondWith element. **NEW_PASSWORD (O) - (String)** New password for the private key file. This is required only if the RequestType is "CHANGE_PASSWORD" **REVOCATION_REASON (O) - (String)** Revocation reason provided in the certification request to revoke certificate. This is required only if the RequestType is "REVOKE" **INVALIDITY_DATE (O) - (Date)** The date provided in the request on which it is known or suspected that the private key was compromised. This is required only if the RequestType is "REVOKE"

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
	RespondWith (O) - (Container)	Used to specify the items the calling application wants to receive within the certification web service response message. **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: **CERTIFICATE (Base64):** The X509 certificate **PKCS_12 (Base64):** The PKCS#12 private key file **PKCS_7 (Base64):** The PKCS#7 certificate chain **EXPIRY_DATE (String):** The expiry date and time of the certificate 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 certificates so that it can notify users that their certificates need to be renewed. **PASSWORD (String):** The server generated password of the PKCS#12 object

An external CA must be able to process the certification request from 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.

20.6.2 Certification Response

Retrieving information from these elements can be accomplished using the API calls described in section 6.5.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values	
CertificateResponse (M) - (Container)	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.
	ResponseStatus (M) - (Enumeration) Provides information regarding whether the request was processed successfully or failed. Possible values are: SUCCESS FAILED PENDING **Note:** PENDING response status is returned only when the certification request is to be manually reviewed and approved by the Admin. The RECOVER_KEY request should be used to retrieve the certificate once it approved and issued by the Certification Service.
	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 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 the ResponseStatus attribute is FAILED 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.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
	ResponseData (O) - (String)	If the value of the ResponseStatus attribute is SUCCESS, then this contains the RespondWith items requested in the corresponding request.
		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: **CERTIFICATE (Base64)** The X509 certificate **PKCS_12 (Base64)** The PKCS#12 private key file **PKCS_7 (Base64)** The PKCS#7 certificate chain **EXPIRY_DATE (DateTime)** The expiry date of the certificate If a value is not available or ADSS Server cannot produce the value then it is not included in the response **PASSWORD (String):** The server generated password of the PKCS#12 object

21 ADSS RA Service – Use Case Overview

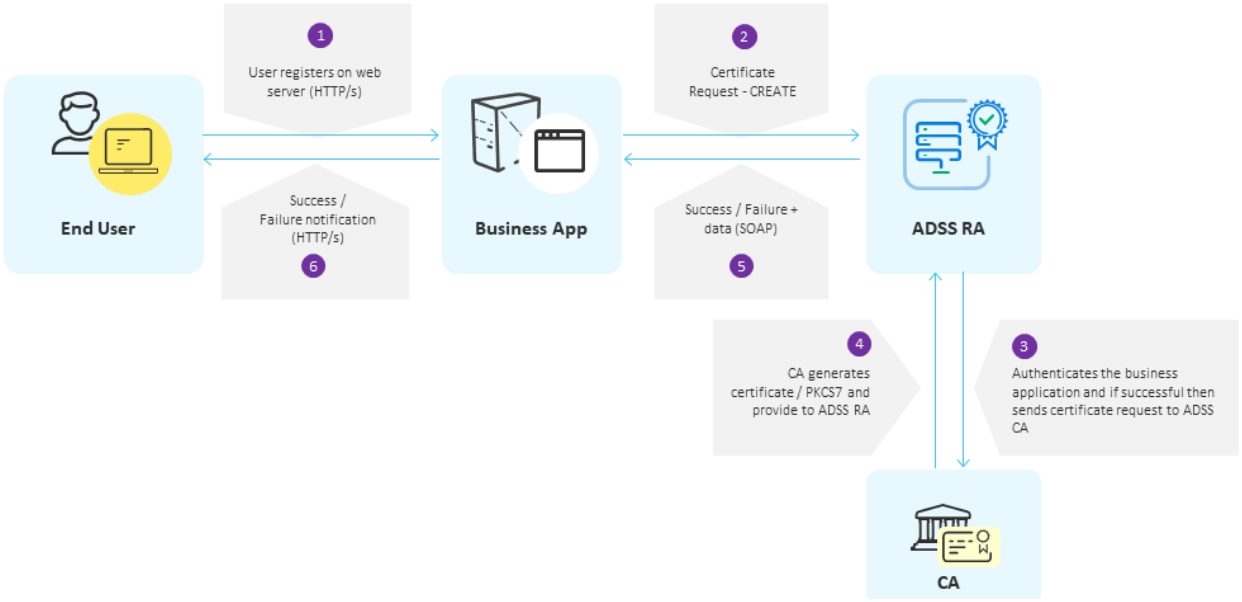
The following sections provide an insight into the possibilities available for certificate creation, revocation and status checking. Additionally, when the request is made using the Ascertia proprietary protocol, descriptions of the certificate request/response XML protocol schemas are also provided.

21.1 Generating a Key Pair and Certificate

When an end-entity requires a key pair and certificate then the business application sends a certificate creation request to ADSS RA Server. The RA server obtains a key pair and the associated certificate from the ADSS CA Server. The keys and certificates generated can now be used to sign documents or data.

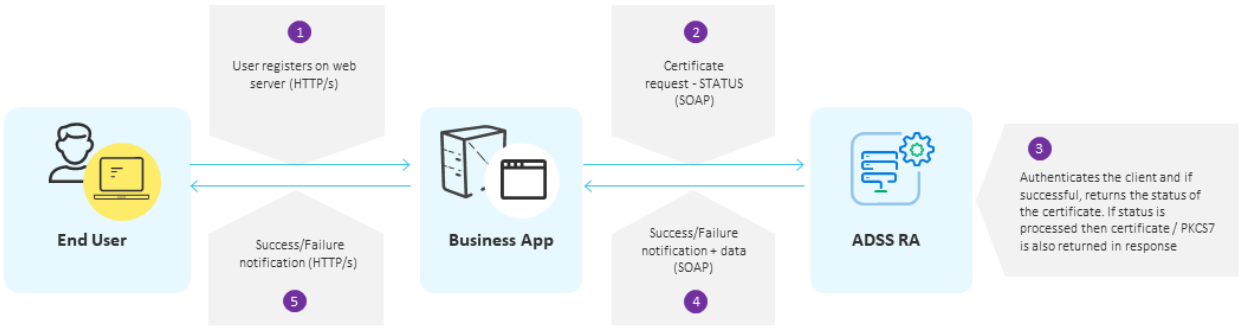
If the business application has already generated/obtained a key pair and a certificate signing request (CSR or PKCS#10) then this can also be sent to ADSS RA Server for certification.

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



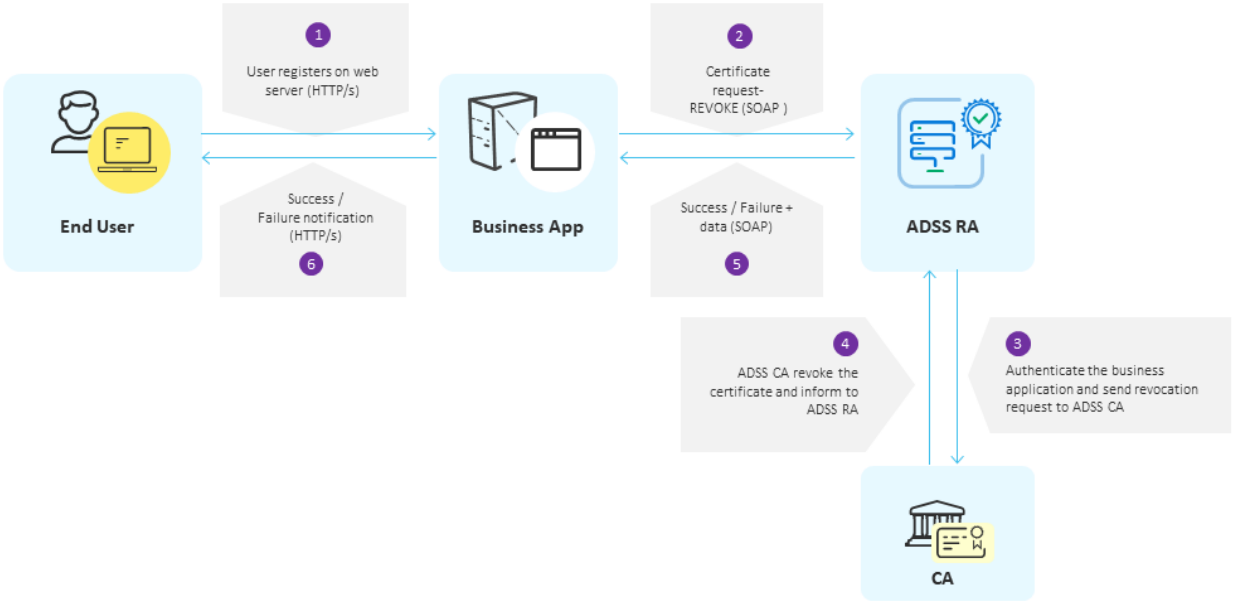
21.2 Status of a Certificate

The business application can also request ADSS RA Server to return the status of the requested certificate and the associated certificate chain:



21.3 Revoking a Certificate

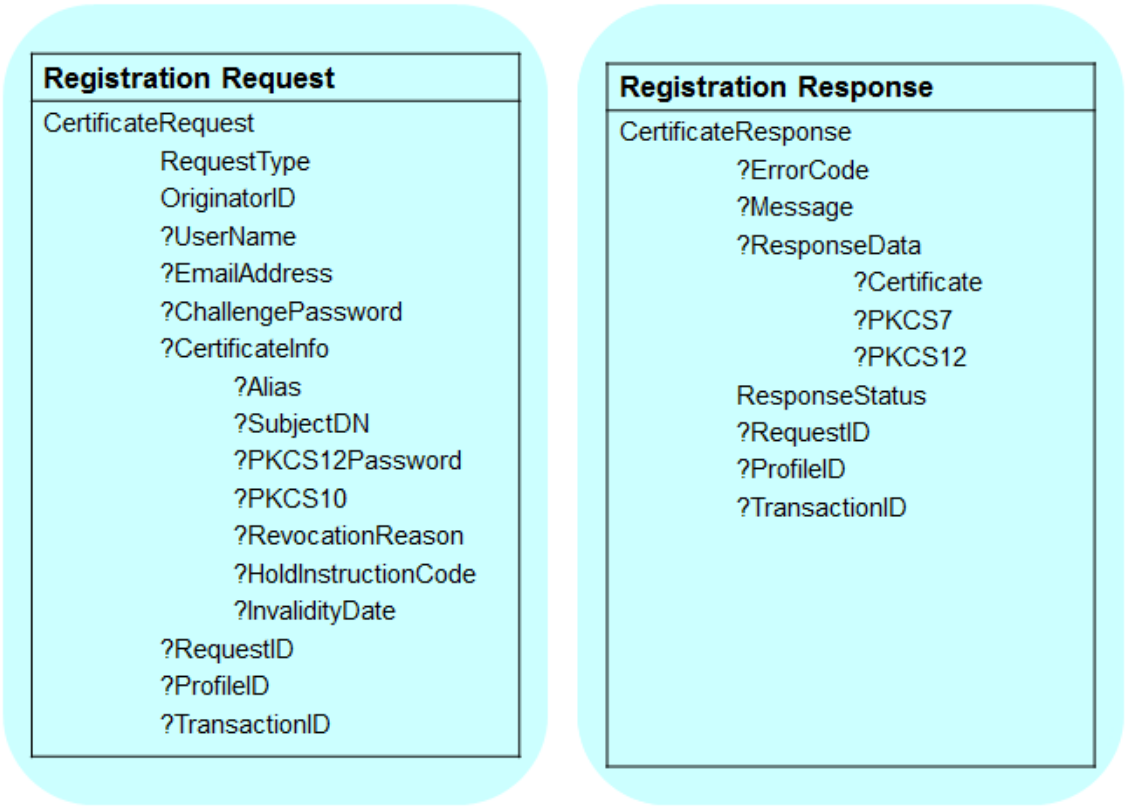
A business application can revoke an existing certificate by sending a certificate revocation request to the ADSS RA Server. After authenticating the client, ADSS Server revokes the the relevant certificate:



21.4 Operations of the RA Service

The RA manages all requests from end-entities that include human users, servers or devices that require X.509 digital certificates from the defined Certification Authorities (CAs) which actually issue these certificates.

The ADSS RA Service is based on a flexible XML schema. An overview of the XML elements used in the ADSS RA Service request and response messages are provided below. For a detailed description refer to the XML Schema file **adss-ra.xsd** provided within the ADSS Client SDK download.



21.4.1 RA Request

Setting these elements can be accomplished using the API calls described in section 14.3.

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 three attributes named RequestID, ProfileID and TransactionID and seven child elements named EntityType, RequestType, OriginatorID, UserName, EmailAddress, ChallengePassword and CertificateInfo.
	RequestType (M) - (Enumeration) Identifies the purpose of the request. The following set of values can be used for this element: CREATE: To create a new key pair and certificate. REVOKE: To revoke the certificate (that already issued). STATUS: To check request status whether the certificate is generated, pending or declined
	OriginatorID (M) - (String) Contains details of the entity that is sending the request**.** A unique ID for the originator. The unique ID must be registered within ADSS Server and if client-authenticated TLS is used then it must match the Common Name within the Client's TLS Certificate. If the OriginatorID is not registered, the request will be rejected.
	RequestID (O) - (String) This attribute helps to uniquely identify a RA request. Any arbitrary string can be used as the value of this attribute. This is expected in the response back from ADSS RA Server.
	ProfileID (O) - (anyURI) This attribute is used to identify a RA profile that ADSS RA Server must use to process this request. See the ADSS RA Server Admin Manual for further details on how to configure RA profiles. The value of this attribute is of the form adss:ra:profile:001. If a profile is not identified then the ADSS Server default RA ProfileID is used.
	TransactionID (O) – (String) This attribute is used to uniquely identify a certificate request in asynchronous mode.
	UserName (O) – (String) This contains the user name of the request.

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
	EmailAddress (O) – (String) This contains the EmailAddress of the requested user. This email address is used for further correspondence with the user.	
	ChallengePassword (O) – (String) This contains the device password to uniquely identify the registered device.	
	CertificateInfo (O) - (Container)	This element is used to provide the certificate specific data to ADSS RA Server. The list of certificate info elements which can be provided in certificate request is detailed below: **ALIAS (M) - (String):** The alias of the certificate for this certification request. ADSS RA Server will generate the certificate using this ALIAS if this alias is not already used for this particular OriginatorID. **PKCS12Password (O) - (String)** Password of the PFX or PKCS#12 private key file. This is required only if the key-pair is generated at server **SubjectDN (O) – (DistinguishedName):** 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, SN, B, ST, P, EVL, EVS and EVC are supported and can be over-ridden depending upon the RA profile settings. **PKCS10 (O) –(base64Binary)** contains PKCS10 /CSR bytes **RevocationReason (O) – (Enumeration)** If the certificate Request type is Revoke then this contains the revocation reason value. The following set of values can be used for this element: • unspecified • keyCompromise • cACompromise • affiliationChanged • superseded • cessationOfOperation • certificateHold • removeFromCRL • privilegeWithdrawn • aACompromise **HoldInstructionCode (O) – (Enumeration)** The following set of values can be used for this element: • id-holdinstruction-none • id-holdinstruction-callissuer • id-holdinstruction-reject

21.4.2 RA Response

Retrieving information from these elements can be accomplished using the API calls described in section 14.3.8

XML element name (M: Mandatory, O: Optional) - Data-type - Description/Allowed Values		
CertificateResponse (M) - (Container)	This is the top level element of Certification Response. It has four attributes named ResponseStatus, RequestID, ProfileID and TransactionID and three child elements named ErrorCode, Message and ResponseData. The detail of each element is provided below.	
	ResponseStatus (M) - (Enumeration) Provides information regarding whether the request was processed successfully or failed. Possible values are: • SUCCESS • FAILED • PENDING • DECLINED	
	RequestID (O) - (String) This attribute helps to uniquely identify a request. If included in the response, the value for this attribute is taken exactly as from the corresponding request message.	
	ProfileID (O) - (anyURI) This attribute identifies the RA 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 the ResponseStatus attribute is FAILED 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)	If the value of the ResponseStatus attribute is SUCCESS then this contains the following items.
		CERTIFICATE (Base64) The X509 certificate **PKCS12 (Base64)** The PKCS#12 private key file **PKCS7 (Base64)** The PKCS#7 certificate chain

**** End of Document ****