SignServer

Manual

Ver: 3.0

08-01-18

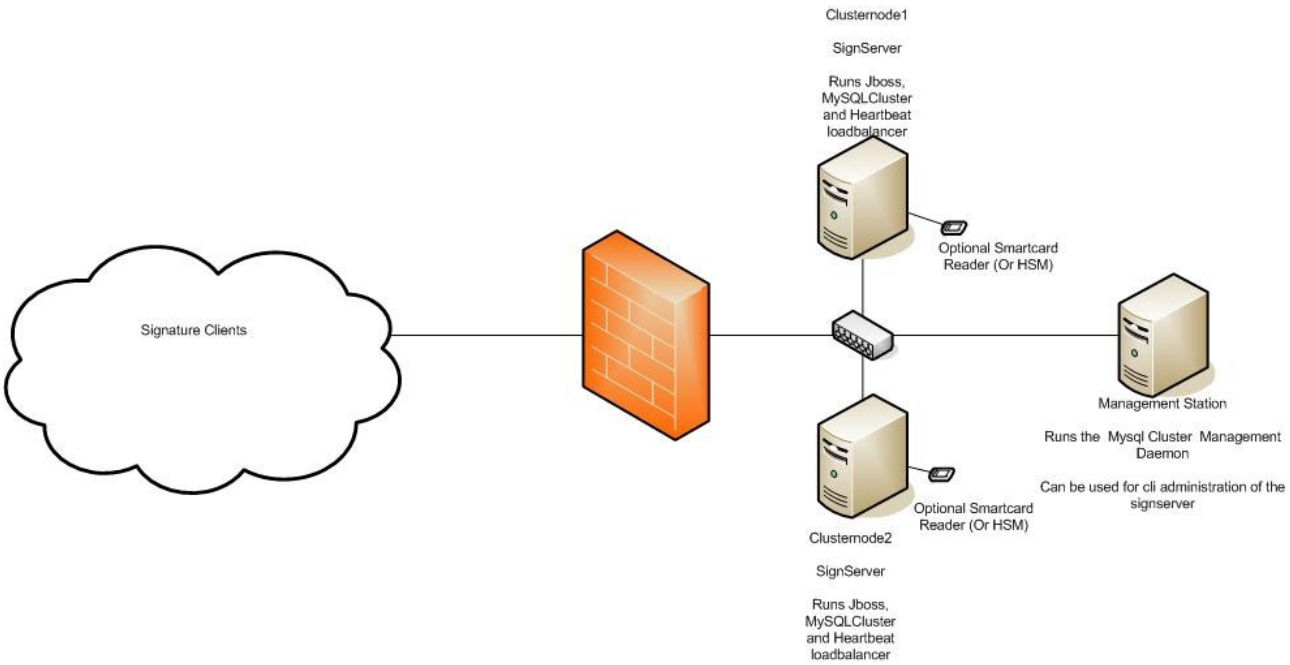
1 Introduction/Scope

The SignServer is an application framework performing cryptographic operations for other applications. It's intended to be used in environments where keys are supposed to be protected in hardware but there isn't possible to connect such hardware to existing enterprise applications or where the operations are considered extra sensitive so the hardware have to protected more carefully. Another usage is to provide a simplified method to provide signatures in different application managed from one location in the company.

The SignServer have been designed for high-availability and can be clustered for maximum reliability.

The SignServer comes with a RFC 3161 compliant Time-Stamp signer serving requests through http or client-authenticated https. A MRTD (Machine Readable Travel Document, i.e. electronic passport) signer. A PDF signer that adds a signature automatically to a uploaded document and a validation service used to lookup the validation of a given certificate.

From version 3.0 there also exists a mail signer framework that can be used to perform cryptographic operation on emails.



Drawing 1: Overview of a possible set up of a highly available SignServer solution

1.1 Changes from previous versions

1.1.1 Changes between Version 2 and Version 3

- Complete refactorisation of J2EE from EJB2 to EJB3 to simplify further development.
- Renamed component “Service” to “TimedService” since 3.0 supports other services.
- A “TimedService” can now be configured with a 'cron-like' settings to have services executed in other than just periodical intervals.
- A Validation Service API used to validate certificate from different issuers. The Validation Service API have it's own easy to use Web Service used to integrate with other platforms.
- A Group Key Service API used to generate and manage group keys, symmetric or asymmetric.
- Possibility to have customized authorization of requests, not just the built in client certificate authorization list.
- The name SignToken is changed to CryptoToken and introduced a new concept of ExtendedCryptoToken that supports symmetric operations.
- The RMI-SSL interface have been removed and replaced with a JAX-WS interface with a simple client framework supporting different load-balance or high availability policies.
- All request data have changed from serialization to externalization to be easier to translate to other platforms.
- A completely new MailSigner API based upon the JAMES SMTP server to perform automated cryptographic operations on e-mails very similar to the plug-ins for the SignServer.
- Java 1.4 is no longer supported.
- A lot of new JUnit tests in the test suite.
- A PDF Signer that can add a signature to a PDF document through a simple HTML interface.
- PKCS11 Crypto Token to connect to different PKCS11 implementations.

1.1.2 Changes between Version 1 and Version 2

- signserver_server.property file have been removed and replaced with a global configuration store.
- It is now possible to dynamically add and remove available signers
- A new type of component, “Service” that is run on a timely basis, used to perform maintenance or report generation.
- Improved cluster deployment functionality.
- New CLI tools to batch configure the SignServer, and to backup a current configuration. This makes it possible to set-up a configuration in test environment, dump the configuration and configure the same it in production.

2 Document History

Version	*Date*	*Name*	*Comment*
0.1	2006-06-04	Philip Vendil	Initial version of this document.
1.0 RC1	2006-08-22	Philip Vendil	Prerelease version
2.0	2007-08-10	Philip Vendil	Update with new features for version 2
2.1	2007-09-22	Tomas Gustavsson	Updated with PDF Signer
2.1.1	2007-09-27	Tomas Gustavsson	Added PKCS11 sign token
3.0	2007-01-06	Philip Vendil	Big update with 3.0 documentation.

Table of Contents

3 Quick start of a Simple Time-stamp Server

This section will show how to set up a quick and simple standalone time-stamp server, accepting time-stamp requests over plain HTTP.

3.1 Required Software

- Java 1.6 (or 1.5) (http://java.sun.com)
- JBoss-4.2.2.GA (http://www.jboss.org)
- Ant version 1.7 (http://ant.apache.org)
- SignServer-3.0 (http://www.signserver.org)
- 1 web-server key-store in Java key-store format (JKS), (make sure that the server certificate have the right host name in it's CN, optional (used for HTTPS))
- 1 Root certificate of the web-server in DER encoding (optional (used for HTTPS)).
- 1 Time-stamp key-store in PKCS12 format

3.2 Installation Steps

1. First make sure that ant, Java and JBoss is installed properly.

2. Set the JAVA_HOME, JBOSS_HOME and SIGNSERVER_HOME environment variables.

3. Set the SIGNSERVER_NODEID environment variable, it should be a server unique string identifying the node in a cluster. (optional for one node installations).

4. Unzip the SignServer package and go to it's home directory.

5. If you are going to protect the HTTP communication with SSL, you need a JKS SSL server key store. Rename the web server key store to tomcat.jks at put it in a 'p12' subdirectory. Also place the web server root certificate in DER encoding in the same directory, call it rootcert.cer

6. Then copy the signserver_build.properties.sample file to signserver_buld.properties and edit the file. At least configure the httpsserver.password property. If you are not using https uncomment the row "`j2ee.web-nohttps=true`".

7. Do 'ant deploy' and then start JBoss (JBOSS_HOME\bin\run.sh) in another console.

8. Edit the signserver_cli.properties and set the host name.* properties.

9. Edit the sample-configs/qs_timestamp_configuration.properties file and set the default policy id, the path to the timestamp p12 file and its password.

10. Use the signserver cli to upload the configuration file. (if it's not executive use chmod +x bin/signserver.sh)

```
bin/signserver.sh setproperties sample-configs/
qs_timestamp_configuration.properties
```

(In the path section, use '\\' for '\' in windows environment.)

Then run
```
bin/signserver.sh getconfig 1
```

Or you could use the signers name
```
bin/signserver.sh getconfig timestampSigner
```
And double check the configuration. (Important, the properties are case sensitive).

Finally run
```
bin/signserver.sh reload 1
```
To activate the configuration.

11. Run the test-client to see that everything is up.
```
cd dist-client
java -jar timeStampClient.jar "http://localhost:8080/signserver/tsa?signerId=1"
```

`

The message "TimeStampRequest Validated" should appear once a second.

Also check JBOSS_HOME/server/default/log/server.log that successful messages appear.

4 Quick start of a Simple Mail Signer

This section will do a fast and minimal configuration of a Mail Signer that will sign all authorized mails sent through it with a PKCS12 cryptographic token.

4.1 Required Software

- Java 1.6 (or 1.5) (http://java.sun.com)
- Ant version 1.7 (http://ant.apache.org)
- SignServer-3.0 (http://www.signserver.org)
- 1 mail signing key-store in PKCS12 format (The key store should have a RFC822Name that matches the sender address that will be replaces by the Simple Mail Signer plug-in). There exists one keystore in src/test/mailsigner_test1.p12 with a RFC822Name of *mailsigner@someorg.org* that can be used for testing.

4.2 Installation Steps

1. First make sure that Ant and Java is installed properly.

2. Set the JAVA_HOME and SIGNSERVER_HOME environment variables.

3. Set the SIGNSERVER_NODEID environment variable, it should be a server unique string identifying the node in a cluster. (optional for one node installations).

4. Unzip the SignServer package and go to it's home directory.

5. Then copy the signserver_build.properties.sample file to signserver_buld.properties and edit the file. First uncomment row "`build.mode=MAILSIGNER`" to instruct that this installation is a MailSigner and not a SignServer. Then go to the end of the file and fill in the required properties: `mailsigner.primarydns`, `mailsigner.secondarydns` and `mailsigner.postmaster`.

6. Then build the mail signer with the command 'ant' in the SIGNSERVER_HOME directory.

7. Edit the sample-configs/qs_simplemailsigner_configuration.properties file and set the sender and from addresses as well as the path and password to the cryptographic token that should be used.

8. Start the mail signer application with the command: `ant run`

9. Open up another console, go to SIGNSERVER_HOME and use the signserver CLI to upload the configuration file. (if it's not executive use chmod +x bin/signserver.sh)

```
bin/signserver.sh setproperties sample-configs/
qs_simplemailsigner_configuration.properties
```

Then run
```
bin/signserver.sh getconfig 1
```

Finally run
```
bin/signserver.sh reload 1
```
To activate the configuration.

19. Finally add an authorized SMTP user with the command:
bin/signserver.sh addauthorizeduser <username> <password>

20. Now is the simple mail signer ready to be used for signing outgoing emails. Configure you e-mail client to connect to the MailSigner server using the username and password you just provided and send a mail to a colleague to verify it is signed properly.

5 Terms Used in This Document

Term	*Explanation*
Signer	A Processable service performing signatures upon requests. This could be a ready made signer or a custom developed one.
Crypto Token (former Sign Token)	A Crypto Token is a name for the entity containing the private key and is responsible for its cryptographic operations. Every Processable have a Crypto Token that can be a PKCS12, Smart Card or HSM connection.
Extended Crypto Token	An enhanced Crypto Token with support for symmetric key operations.
PKCS11CryptoToken	A Crypto Token able to communicate with Hardware Security Modules through the standard PKCS11 interface.
TimedService (former Service)	A TimedService is a task that is run on a timely basis, performing maintenance tasks like changing active key or generate a report.
Worker	A common name for Processable (Signer or other type of service), Mail Processor and TimedService
Processable	A type of worker that is used to process requests, i.e. not a TimedService.
Worker Configuration	Each Worker can be configured with properties specific for that worker. There are two sets of worker configuration one “Active” that is used by the signer and one “current” which is the one configured by the administrator. The current configuration isn't used in production until the administrator issued the reload command. This makes it possible for the administrator to configure multiple properties and double-check them before they are actually used.
Global Configuration Store	Is a dynamic store used to define available Workers and their Crypto Tokens. But other data that needs to be read globally could be set there as well. The global configuration properties are activated immediately. There are two different scopes for the store data, Global Scope and Node Scope.

Term	*Explanation*
Global Scope	Data stored in the global configuration that can be read by all nodes in the cluster.
Node Scope	Data that is node specific and can only be read within the same node.
Worker Id	Unique identifier of a worker, an integer larger than 0
Worker Name	A name used as a human readable synonym for a Worker Id
Mail Processor	A plug-in used for the mail signer to perform automated cryptographic operations on e-mails sent through the mail server. This could for instance be the Simple Mail Signer for attaching a SMIME signature to all mails.
Validation Service	A Processable that checks if a certificate is valid or not. Have a Default Validation Service implementation that should work in most cases. A Validation Service should have one or more Validators configured.
Group Key Service	A Processable that can be used to manage, generate and distribute group keys to a set of clients. The service support four types of calls, fetch group key (used by clients), pre-generate group keys, switch encryption key (key used to safely store the group keys in database) and remove group keys. There exists a Default Group Key Service that should satisfy most use cases.
Validator	A Validator is responsible for checking the status of one or more issuer's certificates. This could be as a OCSP client or a CRL checker or just looking up the status in a database.
Authorizer	An interface that enables developers to integrate the authorization parts with existing authorization systems of who is authorized to perform requests to a Processable.
Time Stamp Signer	A Signer that can be used to set up a Timestamp Authority according to RFC 3161.
MRTD Signer	A Signer that performs signatures of MRTD (Machine Readable Travel Documents, i.e. Electronic Passports) blobs.

Term	*Explanation*
PDF Signer	A Signer that attaches an electronic signature signature to a PDF document.
Simple Mail Signer	A Mail Processor that envelopes authorized mails with a SMIME signature.

6 Overall Architecture

In the 3.0 version the SignServer project have two different builds, one is the classical SignServer and the other is a Mail Processing Server called the MailSigner.

6.1 SignServer

The SignServer is a framework designed to perform different kind of cryptographic operations for different applications.

In the 3.0 version there are three kind of processable services. Signers (used to sign or in other way process requested data). Validation Services used to verify the validity of a certificate against a set of backed issuers. The validation service can be used to simply the integration of PKIs into existing applications. The the third processable service is a group key service framework used to manage and to distribute group keys for different applications, these keys can be both symmetric and asymmetric. In addition to processable services there also exists another concept called Timed Service (called just 'service' in 2.0 edition) which are plug-ins run at defined intervals performing maintenance or reporting routines.

Out-of-the-box are there three Signers ready to be used. They are a MRTD Signer used for signing Machine Readable Travel Documents (also known as Electronic Passports), a Timestamp Signer that can be used to set up a Timestamp Authority and a PDF signer that can be used to automatically sign documents.

The main way of communicating with the SignServer is through a WebService interface (previous versions had a RMI-SSL interface, but that have been replaced by the WS for better platform independence.) but the Timestamp Signer is also available through HTTP communication and the PDF signer have a simple HTML page that allows users to upload documents to be signed.

For an overview of the different concepts in the SignServer see illustration 1. The base component is called Worker which is assigned an id, optionally a name and a configuration. A sub component is a Processable which receives and processes requests. A Processable (optionally) have access to a cryptographic token (CryptoToken) in charge of managing the keys of a Processable. A CryptoToken can be either software or hardware based.

The applications i administrated through a command-line interface, where the properties and access control can be configured.

One SignServer can have multiple services for different purposes.

6.2 MailSigner

The MailSigner is a different build of the SignServer, targeted to perform automated cryptographic operations on e-mails. The MailSigner is an add-on to the James SMTP project and the James SMTP binaries is shipped along with the SignServer package for simplified set-up.

The MailSigner's main component is the MailProcessor which is the base for all MailSigner plug-ins. There exists one ready to use MailSigner called the SimpleMailSigner. It generates a signed SMIME message of all mails sent to through the server.

One instance can have more than one MailProcessor configured. Then will the plug-ins be called by worker id in ascending order.

The MailSigner have similar CLI interface and is administrated much in the same way as the SignServer. The SMTP server have support for SMTP Authentication. For more information about the James SMTP server see http://james.apache.org .

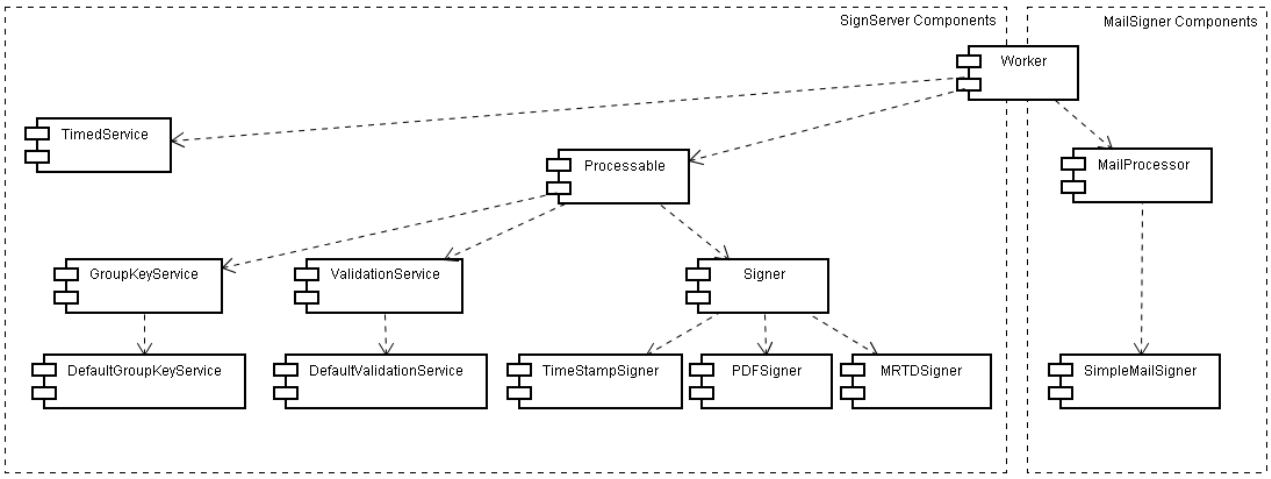


Illustration 1: Components in the SignServer project

7 Available Plug-ins

7.1 Configuring a plug-in

A worker component is configured by entering its class path (and optionally its crypto token class path) in a memory bank called the global configuration and then issuing the reload command. There exists sample configurations for most of the plug-ins in the 'sample-configs' directory.

7.2 SignServer Signers

There currently exists three types of signers. The first one is the time stamp signer generating RFC 3161 compliant timestamps using the Bouncycastle library. A MRTD signer creating 'Machine Reader Travel Document' signatures using the RSA algorithm from pre-padded data. The final signer is a PDF Signer used for automatically signed requested PDF documents.

7.2.1 Time-stamp Signer

The time-stamp signer have the class path: org.signserver.server.signers.TimeStampSigner

7.2.1.1 Overview

The time stamp server generates time stamp tokens and have the support for the following options:

- Set of accepted policies
- Set of accepted algorithms
- Set of accepted extensions
- Accuracy microseconds
- Accuracy milliseconds
- Accuracy seconds
- Included certificate chain (currently doesn't include CRLs)
- Ordering
- TSA name

The time stamp signer currently don't support:

- CRL inclusion
- Signed attributes
- Unsigned attributes

Timestamps requests are served through a http service at the URL:
'http://<host name>/signserver/tsa?signerId=<worker Id>'
If no 'worker Id' parameter is specified then will the id of 1 be used as default.

The time-stamp signer requires a time-stamp certificate with the extended key usage 'time-stamp' only. The extended key usage extension must be critical.

7.2.1.2 Available Properties

The following properties can be configured with the signer:

TIMESOURCE = property containing the class path to the ITimeSource implementation that should be used. (OPTIONAL, default LocalComputerTimeSource)

ACCEPTEDALGORITHMS = A ';' separated string containing accepted algorithms, can be null if it shouldn't be used. (OPTIONAL, Strongly recommended)

Supported Algorithms are: *GOST3411, MD5, SHA1, SHA224, SHA256, SHA384, SHA512, RIPEMD128, RIPEMD160, RIPEMD256*

ACCEPTEDPOLICIES = A ';' separated string containing accepted policies, can be null if it shouldn't be used. (OPTIONAL, Recommended)

ACCEPTEDEXTENSIONS = A ';' separated string containing accepted extensions, can be null if it shouldn't be used. (OPTIONAL)

DEFAULTTSAPOLICYOID = The default policy ID of the time stamp authority (REQUIRED, if no policy OID is specified in the request then will this value be used.)

ACCURACYMICROS = Accuracy in micro seconds, Only decimal number format, only one of the accuracy properties should be set (OPTIONAL)

ACCURACYMILLIS = Accuracy in milliseconds, Only decimal number format, only one of the accuracy properties should be set (OPTIONAL)

ACCURACYSECONDS = Accuracy in seconds. Only decimal number format, only one of the accuracy properties should be set (OPTIONAL)

ORDERING = The ordering (OPTIONAL), default false.

TSA = General name of the Time Stamp Authority. (OPTIONAL)

7.2.2 MRTD Signer

The MRTD signer have the class path: org.signserver.server.signers.MRTDSigner

7.2.2.1 Overview

The MRTD Signer performs a RSA signing operation on incoming data. The data should already be padded. This signer i used to sign 'Machine Readable Travel Documents' i.e. electronic passports.

7.2.2.2 Available Properties

No configuration properties exists.

7.2.3 PDF Signer

The PDF signer have the class path: org.signserver.server.signers.PDFSigner

7.2.3.1 Overview

The PDF signer signs PDF files and have the support for the following options:

- Reason
- Location
- Rectangle

The PDF signer currently don't support:

- Inclusion of time stamps in the signature

PDF signing requests are served through a http service at the URL:
'http://<host name>/signserver/pdf?signerId=<worker Id>'
If no 'worker Id' parameter is specified then will the id of 1 be used as default.

The PDF signer requires a signing keystore with a signing certificate.

7.2.3.2 Available Properties

The following properties can be configured with the signer:

REASON = The reason included in the PDF signature and displayed by the PDF reader. Default value is “Signed by SignServer”.

LOCATION = The location included in the PDF signature and displayed by the PDF reader. Default value is “SignServer”.

RECTANGLE = The coordinates and size of the visible signature field added to the PDF file. Has the form “llx,lly,uux,uuy”, where llx is lower left x coordinate etc.

7.3 SignServer Validation Service Framework

The validation service framework is used to validate certificates from one or more issuers. It can be used to have one central point of performing revokation statuses to simplify the integration of external PKIs within an enterprise.

The validation service framework also provides a validation cache that can be used to increase performance for those cases a application does multiple lookups of the same certificate within a short period of time.

Out-of-the-Box there exists a DefaultValidationService that should satisfy most use cases but it's possible to develop a custom ValidationService if necessary. See the developer section for more details.

All Validation Services is configured by specifying the `org.signserver.validationservice.server.ValidationServiceWorker` in the global configuration, then is the actual ValidationService configured in the worker configuration setting the class path in the property `TYPE` (Not necessary for the DefaultValidationService).

The validation service framework is mostly used with X509v3 certificates but other kinds of certificates is supported as well by design.

Another concept in the Validation Service Framework is that the client also can ask the service to check the type of certificate that the certificate might be used for. A certificate type could be IDENTIFICATION or ELECTRONIC SIGNATURE.

7.3.1 DefaultValidationService

7.3.1.1 Overview

The default validation service have a set of Validators. A validator is responsible to checking the validity against one or more issuers using for example CRL check or OCSP/XKMS lookup or just by checking some database. Currently there are no ready to use validators, these remain to be developed in future versions of the SignServer.

The Default Validation Service supports validations to be cached for some or all issuers for a specified amount of time.

If not configured otherwise will the validation service use the DefaultX509CertTypeChecker that determines the certificate type from the key usage in the certificate. Key Encipherment and Digital Signature indicates a IDENTIFICATION type and Non-reputation indicates ELECTRONIC SIGNATURE.

There exists a validation specific WebService that can be used for platform independent client calls. The WebService must be enabled during the build and isn't by default. The WebService WSDL file is located at the URL `http://<hostname>:8080/signserver/validationws/validationws?wsdl` and it contains two calls one is 'isValid' that performs the validation check and the other is a getStatus call that checks the health of the node and its underlying systems. The last calls can be used by clients for monitoring or implementing redundancy.

Important, Due to class path conflict in JBoss 4.2.x own JBoss WebService stack and the JAX-WS stack used by the SignServer must the JBoss WebService stack be removed before the WebService is used. This is done by going to JBOSS_HOME/server/default/deploy and remove the directory jbossws.sar.

7.3.1.2 Available Properties

The following properties can be configured with the default validation service:

The validation service have three types of properties, general properties (that applies for the service and all configured validators), validator properties (that only applies for a specific validator) and issuer properties (that only applies for an issuer configured in a specific validator).

General Properties:

> ***CACHEDISSUERS*** = A ';' separated list of issuer names (usually issuer DNs) (Optional, no validation is cached if unset.)
>
> ***CERTTYPECHECKER*** = Certificate type checker that should be used to determine the type of certificate (Optional, default is `org.signserver.validationservice.server.DefaultX509CertTypeChecker`)
>
> ***TIMEINCACHE*** = Time in seconds a certificates validation should be cached (Optional, default is 10 seconds)

Validator properties:
Validator properties is specified with the prefix of '`validator<validatorId>.`' or '`val<validatorId>.`' were validator Id should be an integer between 1 and 255. For instance, to specify the type of a validator with an id of 1 then specify 'val1.classpath=some.classpath.SomeClass'. This validator will be initialized with all its validator specific properties (with 'val<id>.' prefix removed) as well as the general ones.

> ***CLASSPATH*** = Class path to the validator that should be used. (Required for each configured validator)

Issuer properties: Issuer properties are specified as 'val<val id>.issuer<issuer id>.<property>' were issuer id is a positive integer between 1 and 255. All generic and validator specific properties (with the given validator id) will also be propagated to the specific issuer configuration.

> ***CERTCHAIN*** = The certificate path of the CA certificates used to verify the certificate. Should be a appended BASE64 string. (Required for each configured issuer).

Here is an example configuration of a validation service to clarify things even further

Set up the worker -> validation service wrapper

```
GLOB.WORKER1.CLASSPATH= org.signserver.validationservice
.server.ValidationServiceWorker
#Uncomment and set class path to custom validation service, othervise is default
#used.
#WORKER1.TYPE=

# Name of Service (Optional)
WORKER1.NAME=ValidationService1

# Define TestCA2 and TestCA3 as a cached for 15 seconds, TestCA1 is Not cached.
WORKER1.CACHEDISSUERS=CN=TestCA2;CN=TestCA3
WORKER1.TIMEINCACHE=15

# Define a validator in charge of issuer TestCA1 and TestCA2
WORKER1.VAL1.CLASSPATH=<Class path to some validator>
WORKER1.VAL1.ISSUER1.CERTCHAIN=EFWAASDFADFASDFKASDKFW1231.....
WORKER1.VAL1.ISSUER2.CERTCHAIN=EFWAASDFADFASDFKASDKFW1231.....

# Define a validator in charge of issuer TestCA3
WORKER1.VAL2.CLASSPATH=<Class path to some validator>
WORKER1.VAL2.ISSUER1.CERTCHAIN=EFWAASDFADFASDFKASDKFW1231.....
```

7.4 SignServer Group Key Service Framework

7.4.0.1 Overview

The group key service framework is used to manage and distribute group keys to clients in an organisation. The keys can be generated on demand or pre-generated at times when the system is not utilized a lot. The group keys can be both symmetric and asymmetric but one service can only distribute one type of key. If several kinds of keys are required should multiple services be set up within the same server.

The group keys are stored encrypted in database. The encryption key can be configured to be switched automatically after a defined number of encryptions to avoid overexposure of the cryptographic data. It is also possible to switch the encryption key manually.

The Framework requires an ExtendedCryptoToken, the difference are that the extended token have additional support for key export and symmetric key operations.

The Group Key Service have CLI commands for administration of the service such as pre-generate keys, manual switch of encryption key and removal of group keys.

The communication to the group key service is mainly done through the main Web Service interface. But other ways of communicating with the server might come in the future.

Authorization to group keys is very important and therefore should a special plug-in be developed that looks up which clients that should have access to a specific group key which fit into the organisation needs. See the authorization chapter of how to develop a customized authorization plug-in.

The basic configuration of a group key service is very similar to that of a validation service. Two entries is required in the global configuration. The first is the class path for the Worker to GroupKeyService wrapper, then a class path reference to the extended crypto token used with the service. If not the default group key service should be used it is possible to define a custom one by specifying its class path in the TYPE worker property.

7.4.0.2 Available Properties

USEPREGENERATION = Setting defining of keys should be pre-generated or generated on the fly when needed. If the pool of pre-generated keys gets empty will new keys always be generated automatically. (Optional, default is true)

ENCKEYALG = Encryption algorithm used to encrypt the group keys stored in database. (Optional, default is “AES”)

ENCKEYSPEC = Specification of the encryption key. (Optional, default is “256”)

GROUPKEYALG = Defines the type of group keys that this service should generate (Optional, default is "AES")

GROUPKEYSPEC = Specification of the generated group keys. (Optional, default is "256")

KEYSWITCHTHRESHOLD = Setting defining the number of group keys that should be encrypted by the same encryption key before it's switched. (Optional, default is 100000)

7.5 Mail Processors

This section lists the available plug-ins for the MailSigner. These plug-ins are configured in the exact same way with class path of both plug-in and its crypto token.

7.5.1 SimpleMailSigner

7.5.1.1 Overview

The SimpleMailSigner is a plug-in that generates a signed SMIME message from any authorized mail sent through the MailSigner server. It can be used prove the origin of the message to the receivers .

There exists a demo configuration in the 'sample-configs' directory.

7.5.1.2 Available Properties

EXPLAINATIONTEXT = Text attached to the e-mail describing the signature for the recipient. (Optional)

USEREBUILDFROM = Setting indicating if the from field of the SMIME should be altered. (Optional, default is true)

SIGNATUREALG = Setting configuring the signature algorithm that should be used in the SMIME message. (Optional, default is DIGEST_SHA1)

POSTMASTERSIGNS = Indicates if postmaster mail should be signed. (Optional, default is false)

FROMADDRESS = The from email address used if rebuild from is set. (Required if USEREBUILDFROM is true)

FROMNAME = Readable name used in from address field. (Optional)

CHANGEREPLYTO = Indicates if the reply-to field should be altered to the original sender. (Optional, default is false)

REPLYTOADDRESS = The reply to email address if the reply always should be changed to a default address. (Required if CHANGEREPLYTO is true)

REPLYTONAME = Readable name used in reply-to address field. (Optional)

SIGNERADDRESS = The email address that should be in the sender field. (Required)

SIGNERNAME = Readable name used in sender address field. (Optional)

REQUIRESMTPAUTH = Setting defining if SMTP AUTH should be required to sign the mail. (Default is true).

8 Available CryptoTokens

There exists three types of crypto tokens (Formerly known as sign tokens), one storing the keys in software, one general for communicating with cryptographic hardware through the PKCS11 interface and one for SmartCards. See the developer section for information about developing support for other HSMs.

8.1 P12CryptoToken

The P12CryptoToken signer have the class path:
org.signserver.server.cryptotokens.P12CryptoToken

8.1.1 Overview

A CryptoToken using a PKCS 12 key-store in the file-system. Can only contain one signing key.

In a clustered environment must the key store be at the same location at all nodes.

The P12CryptoToken, doesn't support the destroyKey() method

8.1.2 Available Properties

KEYSTOREPATH : The full path to the key-store to load. (required)

KEYSTOREPASSWORD : The password that locks the key-store. Used for automatic activation.

8.2 PrimeCardHSMCryptoToken

8.2.1 Overview

Using PrimeCardHSM it's possible to use a SmartCard to generate 2048-bit RSA signatures. The SmartCard can perform about one signature a second. PrimeCardHSM is proprietary software by PrimeKey Solutions AB.

PrimeCardHSM requires PCSCD software and SmartCard drivers. See separate documentation about installing PrimeCardHSM.

The PrimeCardHSMCryptoToken, doesn't support the destroyKey() method.

The PrimeCardHSMCryptoToken signer have the class path:
org.signserver.server.cryptotokens.PrimeCardHSMCryptoToken

8.2.2 Available Properties

DEFAULTKEY = Hash value of the signing key on the card. See PrimeCardHSM documentation for more information.(Required)

AUTHCODE = Authentication code for automatic activation (Optional).

8.3 PKCS11CryptoToken

8.3.1 Overview

Using PKCS11 it's possible to use a HSM that has a PKCS11 module, such as Utimaco, nCipher or Eracom.

The PKCS11CryptoToken have the class path:
org.signserver.server.cryptortokens.PKCS11CryptoToken

8.3.2 Available Properties

DEFAULTKEY = Hash value of the signing key on the card. (Required)

PIN = Authentication code for activation. (Required)

SHAREDLIBRARY = Full path to the library containing the PKCS11 interface. (Required)

SLOT = Slot to use (Required)

8.3.3 Example usage

Edit qs_pdfsigner_configuration.properties and choose the sign token setting for the PKCS11 sign token. Run the following command to set up a PDF signer using the PKCS11 properties configured:
bin/signserver.sh setproperties qs_pdfsigner_configuration.properties

You also need a certificate for the signer. Generate a certificate request with the command:
bin/signserver.sh generatecertreq 8 "CN=PKCS11 Signer token" SHA1WithRSA /tmp/certreq.pem

Add a user in EJBCA with a certificate profile suitable for signing, and enrol for a “Server Certificate” using the public web pages.

Create the certificate chain file with the command:
cat /tmp/cert.pem /tmp/AdminCA1.pem > /tmp/certchain.pem

The signer certificate must be first, and the root CA certificate last.

Upload the signing certificate chain to the signer using the command:
bin/signserver.sh uploadsignercertificatechain 8 GLOB /tmp/certchain.pem

After the certificate chain has been uploaded to the server, the configuration must be reloaded and the server must be restarted. It is not sufficient to only reload the configuration.

9 Setting Authorization Type

9.1 SignServer

By default is client-certificate authentication required for a signature request to be processed. This can be changed with the ***AUTHTYPE*** property.

> ***AUTHTYPE*** = NOAUTH, sets the server to not require any authentication.
>
> ***AUTHTYPE*** = CLIENTCERT (default) requires a certificate of all the clients. The certificates must be in the signers access control list and be trusted by the Java distribution, i.e imported in JAVA_HOME/jre/lib/security/cacerts. Authorized clients is configured manually using the CLI interface.

This authorization functionality doesn't work for all use cases. Then it's possible to create a customized authorizer and specify it's class path as value in the AUTHTYPE property. The Processable will then automatically instantiate and use it. How to develop such a plug-in is explained in the developers section.

9.2 MailSigner

The MailSigner support SMTP Authentication which is configured in the build properties. It's then possible to authorize users by adding them manually through the CLI interface. SMTP authorization is global for all MailProcessors.

10 Disabling a Signer

Any Processable worker in the SignServer and MailSigner can be disabled, which means that it won't be executed or be included in the health check.

To disable a worker set the property “DISABLED” to “TRUE”

11 Archiving Responses *(SignServer only)*

If there is a need to save all generated responses, then set the property “ARCHIVE” to “TRUE” and all generated responses for that signer will be saved to database.

The archived responses can later be extracted from data base using the CLI interface. See the CLI section for more information.

12 The Global Configuration Store

The available workers and its crypto tokens and services is configured in something called the global configuration store that is slightly different from a worker configuration.

Is is dynamically configured and activated immediately. I can contain any type of data (in string representation) and can be of two types, either with global scope or node scope. A Global scoped property can be accessed by all nodes in the cluster while a Node scoped property in only used within a node and cannot be accessed by the other nodes.

12.1 SignServer specific

Database failure is handled differently. If a node looses connection to the database it put itself in a state called 'unsynchronised' and will continue its operation without storing the data to database by using a cached global configuration. It is possible to later resynchronise one nodes cached global configuration data with the database with a CLI command called 'resync'. But it is only possible to sync one of the nodes global configuration to the database.

12.2 MailSigner Specific

The MailSigner doesn't store its data in database but in a regular file handled internally. It have therefore no functionality for database fail-over and cannot be resynchronized. There currently aren't any difference in global and node scoped variables.

13 Timed Services

A Timed Service (formerly called just service) is a task that is run on a timely basis, performing maintenance tasks like changing active key, or it could generate a report.

Currently isn't the SignServer shipped with any services out of the box, but read the developer section about how to write custom services.

A Timed Service framework supports a couple basic properties that is used to calculate when and how a timed service should run. These properties are:

ACTIVE = "TRUE" if the service should be run, otherwise it is disabled.

SINGLETON = "TRUE" if the service only should be run on one of the nodes in the cluster at the time. If it's not set or set to FALSE is the service run simultaneously on all nodes in the cluster. If the node running a singleton service fails will another node sense this and start up the service

INTERVAL = Property that should define the interval i seconds the service should run.

CRON= Property that should define a CRON expression of how often the service should run. It should conform to Unix CRON standard. (One of INTERVAL or CRON is required)

14 The Main WebService Interface

14.1 Overview

New to version 3.0 is the Main WebService interface. It replaces the RMI-SSL interface in version 1.0 and 2.0 for two reasons, the RMI-SSL were based on a commercial library and it only worked for Java clients.

The WebService interface have two calls, the main one is 'process' which takes a collection of process request to a processable worker and returns a collection of process responses, the second one is getStatus that performs a health check of the node and returns an OK message if the node is healthy.

The WebService stack used is the JAX-WS stack from SUN. And the actual process data is a externalized Base64 byte-arrays. The reason why the are externalized is to simply the integration towards non-Java platforms. See the source of the actual request to see how the data is structured.

The getStatus call can be used to implement high-availability towards the client. The Java client API described in the next section have built in support for different high availability policies.

The WebService WSDL file is located at the URL
`http://<hostname>:8080/signserver/signserverws/signserverws?wsdl`

It's possible to turn off the WebService interface by disabling it in the build configuration.

Important, Due to class path conflict in JBoss 4.2.x own JBoss WebService stack and the JAX-WS stack used by the SignServer must the JBoss WebService stack be removed before the WebService is used. This is done by going to JBOSS_HOME/server/default/deploy and remove the directory jbossws.sar.

14.2 Java Client API

Built along with the WebService is a Java API that can be used by clients. It's located in dist-client/signserverwscli and the file signserverws.jar and all the files in the lib directory is required to use the API. The client API have support for different high availability policies to avoid the need for load balance hardware.

The client classes is in the package org.signserver.protocol.ws.client and the main code are SignServerWSClientFactory creating a client using the specified load balance policy, it returns a ISignServerWSClient that is used to perform the actual process requests.

14.2.1 Load Balance Policies

With version 3.0 is one load balance policy defined and it's called '`CallFirstNodeWithStatusOK`' it calls the getStatus method on all the server nodes in the cluster simultaneously and the first node to respond OK it sends its process request to. This to ensure that only one node in the cluster actually performs the signing.

Other future load balance policies could be round robin or that all nodes are called with the requests simultaneously and the first response is used.

14.2.2 CLI Client

Along with the Java Client API is also a CLI utility provided, used mainly for testing but could in some cases be used for scripting as well. Its located in the dist-client/signserverwscli and all the files in the directory and 'lib' subdirectory is required. First edit the file wsclient.properties, there it's possible to define connection properties such as hosts, ports, SSL usage, load balance policy and so on. It is also possible to define the classes used for request generation, what type of request that should be generated and how the response should be analysed. Default is just dummy requests sent to the DummySigner used to test WebService connectivity.

Below is the syntax of the CLI command and can be used to perform continuous requests with a specified rate with a multiple number of concurrent requesting threads. This can be used to test performance of an implemented worker and for stress testing.

```
  Usage:
  wsclient <signer Id or Name> <number of requests> <milliseconds between reques
t> <number of threads> <random wait>
  Where:
    Signer id or name to send requests to, (required parameter)
    Number of requests is for each thread (default is '1'), use 'c' or 'continuo
us' for infinite number of requests.
    Minimum milliseconds between requests is the least time the client waits bef
ore issuing the next (default is '1000').
    Number of threads that will send concurrent requests (default is '1').
    Random wait in milliseconds, used for more random test behaviour and is adde
d to the fixed wait time (default is '0').

  The client requires a configuration file 'wsclient.properties' to exist in the
 same directory as the wscli, see the configuration file for more details.
```

15 Building and Deploying the SignServer or MailSigner

The building of the SignServer framework is configured by copying the file signserver_build.properties.sample to signserver_build.properties. In this file it's possible to set which database that is going to be used, how the web container should be configured and how many nodes that is in the cluster. After it's configured issue the command 'ant' followed by 'ant deploy' to send it to the application server. Currently is JBoss the only supported application server and it requires the environment variable JBOSS_HOME to be set prior to the deploy command.

How to set-up a highly available SignServer cluster is described in a separate document called SignServer_3_0_Installation_Guide.pdf that can be downloaded from http://www.signserver.org The document describes how to set it up in a Cent OS 4.4 environment with two service nodes and one management node, but should be rather easy to adjust to other platforms.

Building the MailSigner is very similar to the SignServer. Just set the property 'build.mode' to 'MAILSIGNER' in the top of the build configuration file and set the required settings (such as DNS servers) in the bottom of the file. Then build the MailSigner with the command 'ant'.

To start up the MailSigner application issue the command 'ant run'. If you want to debug the application in a IDE such as Eclipse issue the command 'ant debug'. This can be very helpful when developing MailProcessors. In Eclipse create a 'Remote Java Application' debug configuration and have it to connect to port 8000, then set a break point in your code and it will stop there the next time the MailSigner enters that state.

Since the MailSigner is built upon the JAMES SMTP server it is possible to configure the SMTP server more than is available in the build configuration file. This is done in the file extapps/james/apps/james/SAR-INF/config.xml, see the JAMES documentation for more information about what is possible and how to do it.

16 Administrating the SignServer

The SignServer and the MailSigner is administrated using a common CLI interface located in bin/signserver.sh/cmd. Most of the commands work in the same way for both build.

Every worker is identified by a id and optionally a name that can be used in all the CLI commands.

It is possible to do configuration of a worker while it's in production. All configuration commands are cached until a reload command is issued and the configuration becomes active.

There is a special property file for the cli interface called signserver_cli.properties defining which nodes that exists in the cluster. The properties are:

> ***hostname.masternode*** = Should only contain one of the nodes, specified as the default master node. Used by operations dealing with the database and where not all nodes in the cluster needs to be contacted. It is possible to override this setting in the CLI by using the -host <host name> parameter.
>
> ***hostname.allnodes*** = Should contain all the nodes in the cluster, separated by a ';'. Mainly used by the commands getStatus, activateCryptoToken and deactivateCryptoToken.

Its possible to customize the CLI with your own code. How to do this is described in the development section.

16.1 General Commands

These commands applies for all types of workers and works in the same way for both the SignServer and MailSigner.

Get Status Command:
Returns the status of the given worker, it says if its crypto token is active or not and the loaded 'active' configuration. It is possible to get a brief summary or a complete listing for one worker or all configured workers. If all workers are displayed will also all the global configuration parameters be displayed.

Get Configuration Command:
Returns the current worker or global configuration depending on options.

For worker configuration observe that this configuration might not have been activated yet, not until a 'reload' command is issued.

Set Property Command:
Sets a custom property used by the worker or crypto token, see reference for the given Worker and CryptoToken for available properties.

Set Properties Command
Command used to batch a set of properties, both for the global and worker configuration.
It can be used to configure a Signer in a test environment, dump all the properties and upload it into production.

It reads all the configuration properties form a property file and depending on the contents of the key it sets the given property. All properties will be set according to the following defined rule set.

Rule	*Comment*
Properties starting with id<num>.	Will set the property to the value of the given id to the worker with the given id.
Properties starting with name<name>.	Will set the property to a worker with the given name. (If the name doesn't exists a unique id will be generated and assigned).
Property keys containing GENID<NUM>, example WORKERGENID1 or GLOB. WORKERGENID1	The SignServer will find a free unique id and assign substitute all GENID<num> with this id.
Properties starting with glob.	Will set a global property with global scope.
Properties starting with node.	Will set a global property with node scope.
Properties starting with -<other prefix><value>	Will remove the property, either worker or global.

See the directory 'sample-configs' for examples.

Remove Property Command:
Removes a configured property

Dump Properties
This tool will dump all configured properties for one or all workers in the system into a property file. If the configuration for one worker is dumped it can be used to transfer the configuration from one installation to another. If all configurations is dumped, it can be used as a backup tool.

Upload Certificate Command:
Used to upload the certificate when the worker only needs the actual signing certificate and not the entire chain.

Upload Certificate Chain Command:
Used when uploading a complete certificate chain to the worker. Which command that is supposed to be used is depending on the worker and crypto token used.

Generate Certificate Request Command
Used to generate a certificate request for a worker to be signed by a certificate authority. It takes distinguished name and signature algorithm as parameters and writes the request in PEM format to file.

Activate Cryptographic Token Command:
Used to activate hard crypto tokens. Authentication code is usually the PIN used to unlock the keys on the HSM. Not used if the token is set to auto-activation.

Deactivate Cryptographic Token Command:
Brings a crypto token off-line. Not used if token is set to auto-activation.

16.2 SignServer Specific Commands

16.2.1 Authorization Related

These commands are used to configure the internal client certificate authorization when it is turned on. It controls which clients that is authorized to request a processable worker.

Add Authorized Client Command:
Adds a client certificate to a processable workers list of acceptable clients using this worker. Specify certificate serial number in hex and the Issuer DN of the client certificate.

Removes Authorized Certificate Command:
Removes added client certificate entries.

List Authorized Clients Commands:
Displays the current list of acceptable clients.

16.2.2 Database Related

Resynchronize Database Command:

The 'resync' command is used after a SignServer had a complete database failure. When this happens will the Global Configuration become in 'Off-line' mode and it's not possible for the nodes to communicate internally and the Global Configurations will not be in sync any more. After the database is up again can this command be sent to the node that have the most valid Global Configuration and write it to the database. After this will the Global Configuration be in 'On-line' mode again.

16.2.3 Archive Related

This commands can be used for processable workers that have archiving turned on. They are used to find specific archived responses. It's up to the implementation of the worker if it supports archiving or not.

Archive Find from Archive Id Command:
Command used to extract archived data from database identified by the archive Id.

The Id depends on the worker, in case of the TSA is the TimeStampInfo serial number used.
The data is stored with the same file name as the archive id in the specified path.

Archive Find from Request IP Command:
Used to extract all archived data requested from a specified IP address.

All data is stored as separate files with the archive id as file name in the specified path.

Archive Find from Request Certificate Command:
Used to extract all archived data requested from a client by specified it's certificates serial number and issuer DN.

All data is stored as separate files with the archive id as file name in the specified path.

16.2.4 Group Key Service Related

These commands only applies for group key services.

Pregenerate Group Keys Command:
Command used to pregenerate a given number of group keys for a given group key service and stores them unassigned encrypted in the database. This commands can be used to let the cluster work on CPU insensitive key generation during low business hours.

Remove Group Keys Command:
Command used to remove group keys not used any more. A time range of when created, first used and last fetched can be used as criteria.

Switch Encryption Key Commands:
Command used manually switch the encryption key used to secure the group keys in database. Usually is the encryption key switched automatically but this command can be used to override this default behaviour.

16.3 MailSigner Specific Commands

Add Authorized User Command:
Command to add a SMTP authorized user to a MailSigner, it applies for all configured MailProcessors.

Remove Authorized User Command:
Removes an authorized SMTP user.

List Authorized Users Command:
List currently authorized STMP users.

17 Making the SignServer highly-available

Here are some tips on configuration used to make the SignServer redundant. Usually is the SignServer set-up with three nodes (required minimum for MySQL cluster) where one node is a management node from were all deployment and administration is done and the other two services are service nodes processing the actual requests.

17.1 HTTP access requires a load balancer

HTTP based workers like the TSA can be clustered using a load balancer accessing a health check servlet returning the state of the SignServer. The basic settings of the health check servlet can be configured in the build configuration file but more advanced settings are done in 'src/web/healthcheck/WEB-INF/web.xml'. With the default settings will the servlet return the text 'ALLOK' when accessing the URL http://localhost:8080/signserver/healthcheck/signserverhealth. If something is wrong with the sign server will an error message be sent back instead.

The health check servlet can also be used to monitor the SignServer by creating a script that monitors the URL periodically for error messages.

Tip, heartbeat with ldirectord is a good solution for a load balancer and works well with the SignServer. KeepAlived is another open source solution.

The Main WebService using the Java client API manages the HA parts itself and then isn't a load balancer necessary.

17.2 Setting up a MySQL Cluster

The database backed of the SignServer can be made redundant using MySQL Cluster. Details on how to set-up the MySQL cluster can be found in the document SignServer_3_Installation_Guide.pdf that can be downloaded from http://www.signserver.org. More information about the MySQL Cluster can be found at http://www.mysql.com/products/database/cluster/

17.3 MailSigner

The MailSigner have no built in support for high-availability instead are the standard SMTP approach used where multiple server can be set up with different priority in the DNS MX records.

18 For Developers

This section describes the details of how to develop your own plug-ins for the SignServer API. It goes through most of the necessary interfaces to get going.

18.1 Building with Customized Code

It is possible to have your own code in a separate code tree to avoid a mix of custom code with SignServer project code. This makes it easier to maintain and update the code for future versions. This is done by configuring one or more of the 'custom.' parameters in the build configuration file. The are each described here:

custom.src.java = Should point to an external directory containing the package base of the Java code. These are then included in the compilation at build time.

custom.src.web = Should point to the base of a WAR source tree with WEB-INF/web.xml included. This will replace the default WAR deployed during the build.

custom.build.xml = This can point to a custom build.xml that will be imported from the main build.xml and lets the developer include his own ant tasks if necessary.

custom.commandfactory = Should point to a custom implementation of the interface org.signserver.cli.`ISignServerCommandFactory`. This gives to ability to extend (or replace) the default CLI with another one. The best way of extending the CLI is to look at how the `DefaultSignServerCommandFactory` is structured.

18.2 Implementing Workers

The main component in the SignServer is the Worker from which most other components inherits. To get a better overview of how the different component types relate to one and another see illustration 1 in the Overview section.

Most workers work in the same way but with different interfaces to implement but for all of them should the following steps be performed.

- Create a custom class implementing the specified interface. There usually exists a base class implementing the most basic function to simply the implementation even further. If it exists it's recommended to inherit it.
- You can define your own properties that the worker can use for its configuration.
- Make sure the custom class is available to the application server
- Redeploy the SignServer.

- Register the worker in the application by setting a property `WORKER<id>.CLASSPATH` with a global scope in the global configuration. (Also make sure to set it's crypto tokens class-path, see separate section).
- Reload the service with the CLI reload command.

18.2.1 The ISigner Interface

A Signer is a component used to perform some form of cryptographic processing of requested data and to create a custom signer class it should implement the org.signserver.server.signers.ISigner interface. There exists a BaseSigner that can be inherited taking care of some of the functionality. If the BaseSigner is inherited the only method that needs to be implemented is '`processData`() '.

There exists a DummySigner implementation that is used for demonstration purposes.

18.2.2 The ITimedService Interface

There are two kinds of timed services, singleton or non-singleton. A singleton service is only run at one of the nodes at the time while non-singleton services are run at all nodes simultaneously. If a singleton service fails to run on one of the nodes will one of the other nodes take over the service automatically.

If a service should be singleton or not is determined by a standard property SINGLETON defined in the ServiceConfig class.

Other basic properties used to configure all services are:
`ACTIVE` when set to `"TRUE"` means that the service is active and should be run.
`INTERVAL` defining the interval in seconds of how often the service should be run.
`CRON` used as a complement to `INTERVAL` to specify on a calendar basis.

To create a custom timed service class it should implement the org.signserver.server.timedservices.ITimedService interface. There exists a BaseTimedService that can be inherited taking care of most of the basic functionality. If the BaseTimedService is inherited the the only method that needs to be implemented is the 'work()' method.

The work method that needs to be implemented is described here:

```
/**
 * Method that should do the actual work and should
 * be implemented by all services. The method is run
 * at a periodical interval defined in getNextInterval.
 *
 * @throws ServiceExecutionFailedException if execution of a service failed
 */
public void work() throws ServiceExecutionFailedException;
```

There exists a DummyTimedService implementation that is used for demonstration purposes.

18.2.3 IValidationService Interface

Just as the other worker plug-ins have the validator service a base class taking care of most of the common methods and the only method that needs to be implemented is the 'validate' method below. But for most applications should the DefaultValidationService work. What is probably more interesting is to develop a custom IValidator used to integrate the default validation service against different certificate status repositories. See section called 'Other Customizations' for details of how to implement a Validator.

```
        /**
         * Method used to check the validation of a certificate
         *
         * @param validationRequest
         * @return a ValidateResponse
         * @throws IllegalRequestException if data in the request didn't conform with the
 specification.
         * @throws CryptoTokenOfflineException if the crypto token isn't online.
         * @throws SignServerException for general failure exception during validation
         * @see org.signserver.validationservice.common.ValidateRequest
         * @see org.signserver.validationservice.common.ValidateResponse
         */
        ValidateResponse validate(ValidateRequest validationRequest) throws IllegalRequestException,
CryptoTokenOfflineException, SignServerException;
```

18.2.4 IGroupKeyService Interface

To customize a group key service is slightly more work. Then need five methods be implemented: 'fetchGroupKey', 'pregenerateGroupKeys', 'swithEncryptionKey', 'removeGroupKeys' and 'getStatus'. The default implementation stores the group keys in database with a reference to the encryption key used, the encryption key is stored in the extended key store. See the JavaDoc and the code for the default group key service for more details of implementing a customized one.

```
        /**
         * Main method of a Group Key Service responsible for fetching keys from
         * the database.
         *
         * @param fetchKeyRequest
         * @return a FetchKeyReponse
         * @throws IllegalRequestException if data in the request didn't conform with the
specification.
         * @throws CryptoTokenOfflineException if the crypto token isn't online.
         * @throws SignServerException for general failure exception during key generation.
         * @see org.signserver.groupkeyservice.common.FetchKeyRequest
         * @see org.signserver.groupkeyservice.common.FetchKeyResponse
         */
        FetchKeyResponse fetchGroupKey(FetchKeyRequest fetchKeyRequest) throws
IllegalRequestException, CryptoTokenOfflineException, SignServerException;

        /**
         * Method that instructs the group key service to pregenerate keys.
         * This method is called at periods when the server is having
```

```
         * a low load. This option is optional to implement, if the
         * service doesn't support this method it should return null.
         *
         *
         * @param pregenerateKeysRequest request data
         * @return a response containing number of keys generated, etc
         * @throws IllegalRequestException if requests contain unsupported data.
         * @throws CryptoTokenOfflineException if the crypto token isn't online.
         * @throws SignServerException for general failure exception during key generation.
         * @see org.signserver.groupkeyservice.common.PregenerateKeysRequest
         * @see org.signserver.groupkeyservice.common.PregenerateKeysResponse
         */
        PregenerateKeysResponse pregenerateGroupKeys(PregenerateKeysRequest pregenerateKeysRequest)
throws IllegalRequestException, CryptoTokenOfflineException, SignServerException;


        /**
         * Method instructing the key service to switch the encryption key for
         * storing the group keys in the database. This to ensure that one encryption
         * key isn't exposed through to much data.
         *
         * This method is optional for the implementing service to implement, if
         * it's not implemented it should return null.
         *
         * @param switchEncKeyRequest request data.
         * @return a response containing the result of the operation such as new key index.
         * @throws IllegalRequestException if requests contain unsupported data.
         * @throws CryptoTokenOfflineException if the crypto token isn't online.
         * @throws SignServerException  for general failure exception during key generation.
         * @see org.signserver.groupkeyservice.common.SwitchEncKeyRequest
         * @see org.signserver.groupkeyservice.common.SwitchEncKeyResponse
         */
        SwitchEncKeyResponse switchEncryptionKey(SwitchEncKeyRequest switchEncKeyRequest) throws
IllegalRequestException, CryptoTokenOfflineException, SignServerException;

        /**
         * Method instructing the key service to remove old group keys not used anymore
         * it up to the caller to check that the implementing service supports the type
         * of IRemoveGroupKeyRequest used. The request should contain data specifying which
         * keys that should be removed.
         *
         * This method is optional for the implementing service to implement, if
         * it's not implemented it should return null.
         *
         * @param removeGroupKeyRequests request data.
         * @return a response containing the result of the operation such as number of keys actually
removed.
         * @throws IllegalRequestException if requests contain unsupported data.
         * @throws CryptoTokenOfflineException if the crypto token isn't online.
         * @throws SignServerException  for general failure exception during key generation.
         * @see org.signserver.groupkeyservice.common.RemoveGroupKeyResponse
         * @see org.signserver.groupkeyservice.common.IRemoveGroupKeyRequest
         */
        RemoveGroupKeyResponse removeGroupKeys(IRemoveGroupKeyRequest removeGroupKeyRequests) throws
IllegalRequestException, CryptoTokenOfflineException, SignServerException;


        /**
         * Should return the actual status of the worker, status could be if
         * the signer is activated or not, or equivalent for a service.
         * @return a WorkerStatus object.
         */
        public WorkerStatus getStatus();
```

18.2.5 IMailProcessor Interface

Implementing a MailProcessor for the MailSigner is almost exactly the same as for the other components but here it's the 'service' method that needs to be implemented. There also exists a utility class called SMIMEHelper that contains methods for securing emails.

```
        /**
         * Main method used when processing mails
         * @param mail the mail sent through the SMTP server
         * @throws MessagingException if error occurred during processing of mail.
         * @throws CryptoTokenOfflineException if the signing token not available at the time of the
process.
         */
        void service(Mail mail) throws MessagingException, CryptoTokenOfflineException;
```

18.3 Implementing Crypto Tokens

18.3.1 The ICryptoToken Interface

- A custom crypto token needs to implement the interface org.signserver.server.cryptotokens.ICryptoToken. See P12CryptoToken for an example implementation.
- You can define own properties for a crypto token in the same way as for workers. The properties are sent to the crypto token upon initialization.
- Make sure the custom class is available to the application server
- Redeploy the SignServer.
- Register the crypto token to a worker in the application by setting a property `WORKER<id>.CRYPTOTOKEN.CLASSPATH` with a global scope in the global configuration. (Also make sure to set it's crypto tokens class-path, see next section).
- Reload the service with the CLI reload command.

The ICryptoToken interface have the following methods that needs to be implemented:

```
public interface ICryptoToken {
        public static final int PURPOSE_SIGN = 1;
        public static final int PURPOSE_DECRYPT = 2;

        public static final int PROVIDERUSAGE_SIGN    = 1;
        public static final int PROVIDERUSAGE_DECRYPT = 2;

    /**
     * Method called after creation of instance.
     *
     */
        public abstract void init(Properties props) throws
CryptoTokenInitializationFailureException;

        /**
         *  Method that returns the current status of the crypto token.
```

```
         *
         *  Should return one of the SignerStatus.STATUS_.. values
         */
        public abstract int getCryptoTokenStatus();

    /**
     * Method used to activate SignTokens when connected after being off-line.
     *
     * @param authenticationcode used to unlock crypto token, i.e PIN for smartcard HSMs
     * @throws CryptoTokenOfflineException if SignToken is not available or connected.
     * @throws CryptoTokenAuthenticationFailureException with error message if authentication to
crypto token fail.
     */
    public abstract void activate(String authenticationcode) throws
CryptoTokenAuthenticationFailureException, CryptoTokenOfflineException;

    /**
     * Method used to deactivate crypto tokens.
     * Used to set a crypto token too off-line status and to reset the HSMs authorization code.
     *
     * @return true if deactivation was successful.
     */
    public abstract boolean deactivate();

    /** Returns the private key (if possible) of token.
    *
    * @param purpose should one of the PURPOSE_... constants
    * @throws CryptoTokenOfflineException if CryptoToken is not available or connected.
    * @return PrivateKey object
    */
    public abstract PrivateKey getPrivateKey(int purpose) throws CryptoTokenOfflineException;

    /** Returns the public key (if possible) of token.
    *
    * @param purpose should one of the PURPOSE_... constants
    * @throws CryptoTokenOfflineException if CryptoToken is not available or connected.
    * @return PublicKey object
    */
    public abstract PublicKey getPublicKey(int purpose) throws CryptoTokenOfflineException;


    /** Returns the signature Provider that should be used to sign things with
     *  the PrivateKey object returned by this crypto device implementation.
     *  @param providerUsage should be one if the ICryptoToken.PROVIDERUSAGE_ constants
     *  specifying the usage of the private key.
     * @return String the name of the Provider
     */
    public abstract String getProvider(int providerUsage);

    /**
     * Method returning the crypto tokens certificate if it's included in the token.
     * This method should only be implemented by soft crypto tokens which have the certificate
     * included in the key store.
     *
     * All other crypto tokens should return 'null' and let the signer fetch the certificate from
database.
     *
     */

    public abstract Certificate getCertificate(int purpose) throws CryptoTokenOfflineException;


    /**
     * Method returning the crypto tokens certificate chain if it's included in the token.
     * This method should only be implemented by soft crypto tokens which have the certificates
     * included in the key store.
     *
     * All other crypto tokens should return 'null' and let the signer fetch the certificate from
database.
     *
     */
```

```
    public abstract Collection<Certificate> getCertificateChain(int purpose) throws
CryptoTokenOfflineException;

        /**
         * Method used to tell the crypto token to create a certificate request using its crypto
token.
         */
        public ICertReqData genCertificateRequest(ISignerCertReqInfo info) throws
CryptoTokenOfflineException;

        /**
         * Method used to remove a key in the signer that shouldn't be used any more
         * @param purpose on of ICryptoToken.PURPOSE_ constants
         * @return true if removal was successful.
         */
        public boolean destroyKey(int purpose);
}
```

18.3.2 The Extended Crypto Token Interface

The default group key service need support for symmetric keys in addition the the functionality provided in the basic crypto token which mainly focuses on asymmetric key functionality.

The extended crypto token adds four more methods that need implementation used to generate exportable keys (symmetric or asymmetric) and to encrypt/decrypt data using symmetric keys.

```
public interface IExtendedCryptoToken extends ICryptoToken {

        /**
         * Method instructing the crypto token to generate a key that is returned
         *
         * @param keyAlg the key algorithm to generate, it's up to the caller to check that the
crypto token
         * used supports the given value.
         * @param keySpec specification of the key, it's up to the caller to check that the crypto
token
         * used supports the given value.
         * @return either a java.security.Key or a java.security.KeyPair depending on type of keyAlg
sent to the the crypto token.
         * @throws IllegalRequestException if the token doesn't support the given key alg or key
spec.
         * @throws CryptoTokenOfflineException if the token isn't online.
         */
        Serializable genExportableKey(String keyAlg, String keySpec) throws IllegalRequestException,
CryptoTokenOfflineException;

        /**
         * Instructs the crypto token to generate a key stored in the device returning only
         * a alias reference to the key.
         *
         * @param keyAlg the key algorithm to generate, it's up to the caller to check that the
crypto token
         * @param keySpec keySpec specification of the key, it's up to the caller to check that the
crypto token
         * used supports the given value.
         * @return a reference to the key in that can be used later for encryption/decryption.
         *
         * @throws IllegalRequestException if the token doesn't support the given key alg or key
spec.
         * @throws CryptoTokenOfflineException if the token isn't online.
         */
```

```
        String genNonExportableKey(String keyAlg, String keySpec) throws IllegalRequestException,
CryptoTokenOfflineException;

        /**
         * Method used to encrypt data using a key stored in the crypto token. This
         * method should mainly be used for symmetric encryption.
         * @param keyRef a alias reference to the key that should be used.
         * @param data the data to encrypt.
         * @return the encrypted data.
         * @throws CryptoTokenOfflineException if the token isn't online.
         */
        byte[] encryptData(String keyRef, byte[] data) throws CryptoTokenOfflineException;

        /**
         * Method used to decrypt data using a key stored in the crypto token. This
         * method should mainly be used for symmetric encryption.
         * @param keyRef a alias reference to the key that should be used.
         * @param data the data to decrypt.
         * @return the encrypted data.
         * @throws CryptoTokenOfflineException if the token isn't online.
         */
        byte[] decryptData(String keyRef, byte[] data) throws CryptoTokenOfflineException;


}
```

18.4 Other Customizations

18.4.1 The IValidator Interface

A Validator is used in the DefaultValidationService to connect to different kinds of certificate status repositories, such as CRL, OCSP, XKMS, database etc. It contains two methods 'validate' used for the actual certificate validation and 'testConnection' used by health check related functionality to check that the connection to the underlying validator resource is alright.

```
        /**
         * Main method of a Validation Service responsible for validating certificates.
         *
         * Important a validator also have to support to check the revocation status of the
         * involved CA certificates and should only return Validation object with status REVOKED or
VALID
         * If the validator doesn't support the given issuer it must return null.
         *
         *
         * @param cert the certificate to validate.
         * @return a Validation object or null if the certificate couldn't be looked up in this
validator.
         * @throws IllegalRequestException if data in the request didn't conform with the
specification.
         * @throws CryptoTokenOfflineException if the crypto token isn't online.
         * @throws SignServerException for general failure exception during validation.
         */
        Validation validate(ICertificate cert) throws IllegalRequestException,
CryptoTokenOfflineException, SignServerException;
```

```
        /**
         * Optional method used to test the connection to a specific underlying validator
implementation.
         *
         * @throws ConnectException if connection to underlying validator implementation failed.
         * @throws SignServerException for general failure exception during validation.
         */
        void testConnection() throws ConnectException, SignServerException;
```

18.4.2 The IAuthorizer Interface

It's possible to integrate the authorization of processable requests with external authorizations applications. All that is needed is a class implementing the IAuthorizer interface containing two methods, 'init' and 'isAuthorized'.

To register that the customized authorizer should be used by a worker, all that's needed to be done is to set the property AUTHTYPE to the class path of the authorizer implementation.

```
public interface IAuthorizer {

        /**
         * Method called by the worker upon first call to the authenticator after instantiation.
         *
         * @param workerId id of worker.
         * @param config active worker configuration of worker
         * @param em the SignServer EntityManager
         * @throws SignServerException if unexpected error occurred during initialization.
         */
        void init(int workerId, WorkerConfig config, EntityManager em) throws SignServerException;

        /**
         *
         * Main method determining if the requester is authorized to process the data in the
request.
         *
         * @param request the request data sent to the worker to process.
         * @param requestContext containing the optional clientCert client certificate or remote IP
of the user, may also contain customly defined data.
         * @throws SignServerException if unexpected error occurred during authorization.
         * @throws IllegalRequestException if the requester isn't authorized or couldn't be
authenticated for some other reason.
         */
        void isAuthorized(ProcessRequest request, RequestContext requestContext) throws
IllegalRequestException, SignServerException;
}
```

18.5 Using the Global Configuration Store

The global configuration store is a memory bank that workers can use to store data used in ongoing operations. The data can be either node (i.e. only read by the current node) or global scoped.

To access the global configuration store use the `getGlobalConfigurationSession()` method from the BaseWorker (inherited by most of the base component implementations). The returned GlobalConfigurationSession have the following methods that can be used (the other ones should be avoided)

```
/**
* Method setting a global configuration property. For node. prefix will the
node id be appended.
* @param scope, one of the GlobalConfiguration.SCOPE_ constants
* @param key of the property should not have any scope prefix, never null
* @param value the value, never null.
*/
public void setProperty( java.lang.String scope,java.lang.String
key,java.lang.String value ) ;
/**
* Method used to remove a property from the global configuration.
* @param scope, one of the GlobalConfiguration.SCOPE_ constants
* @param key of the property should start with either glob. or node., never
null
* @return true if removal was successful, othervise false.
*/
public boolean removeProperty( java.lang.String scope,java.lang.String key )
;
/**
* Method that returns all the global properties with Global Scope and Node
scopes properties for this node.
* @return A GlobalConfiguration Object, nevel null
*/
public org.signserver.common.GlobalConfiguration getGlobalConfiguration( ) ;
```

The getGlobalConfiguration returns a GlobalConfiguration and have a method `String getProperty(String scope, String property)` that can be used.

The value of the property can be user-defined as long as it is guaranteed to be unique over the entire application.

Reserved values are all property keys starting with "`WORKER`".

19 Testing

There exists some test scripts used to test that the SignServer functions correctly. They are described here.

19.1 Automatic Junit Tests

Automatic Junit tests lies in the directory '`src/tests`'. There are two different test suites, one for the SignServer build and the other for the MailSigner. The same command applies for both.

Important: For the SignServer test suite to run successful through all the tests must both the Main WebService and validation service WebService API be enabled in the build configuration.

To run the test suite do the following:

- Set the environment variable SIGNSERVER_HOME
- Make sure the sign server is deployed and JBoss is running
- do 'ant test:run'
- A protocol is generated in the directory 'tmp/bin/junit'

19.2 Testing the TimeStamp Authority

19.2.1 The TSA Test Client

There exists a Time Stamp Authority test client that is built with the main distribution. For other workers it's recommended to test it with the WS CLI client described in the 'Main WebService' section.

It only works without client authentication requirements and through HTTP.

To run the client do

```
ant
cd dist-client
java -jar timeStampClient.jar "http://<hostname>:8080/signserver/tsa?signerId=1"
```

It will continuously make one request per second.

19.2.2 Manual Tests

The time stamp signer have been tested with the OpenTSA client with both HTTP and HTTPS.

20 References

Java (http://java.sun.com)

Jboss (http://www.jboss.org)

Apache James mail server (http://james.apache.org/)

Apache Ant (http://ant.apache.org)

Bouncycastle (http://www.bouncycastle.org)

RFC3161, Time-Stamp Protocol (TSP) (http://www.ietf.org)