JDigiDoc Programmers Guide

Veiko Sinivee
S|E|B IT Partner Estonia

Contents

Overview

JDigiDoc is a library of Java classes. It offers the functionality for creating digitally signed files in DIGIDOC-XML 1.4, 1.3, 1.2 and 1.1 formats, adding new signatures, verifying signatures , timestamps and adding confirmations in OCSP format.

In addition to digital siging the JDigiDoc library offers also digital encryption and decryption according to the XML-ENC standard. This standard describes encrypting and decrypting XML documents or parts of them but it also allowes one to encrypt any binary data in Base64 encoding. JDigiDoc enables one also to compress the data with ZLIB algorithm before encryption. JDigiDoc encrypts data with a 128 bit AES transportkey whitch is in turn encrypted with the recipients certificate.

JDigiDoc classes are modeled after the XML-DSIG, ETSI and XML-ENC standards. This document describes the JDigiDoc architecture, configuring possibilities and how to use it in Java programs.

One of the design targets of this library is usability in various environments and possibility of the user to exchange parts of the library to other modules offering the similar functionality by other means. For example the library does not assume the existence of a J2EE container. One can use it also in standalone Java programs. Library encapsulates certain key functionality in separate modules and communicates with those modules over a fixed interface. One can always create a new module offering similar functionality and change the configuration to make library use it. For example the base library uses a separate module for creating new digital signatures with a smart card. This module uses a PKCS#11 driver to access the smartcard. But third-party implementations exist (not part of JDigiDoc distribution) that implement the same interface but use a cryptographic accelerator device (HSM) for creating new digital signatures. The basic part of the library can still be used for creating and parsing digidoc files no matter if the RSA signature is created by a smartcard, HSM or perhaps soft-keys. The exchangeable modules have been implemented mostly as singleton – factory classes that implement the fixed Java interface for this functionality.

DigiDoc documents are XML files based on the international standards XML-DSIG and ETSI TS 101 903. DigiDoc documents and the JDigiDoc library implement a subset of XML-DSIG and ETSI TS 101 903. The document format version 1.3 is fully conforming to XAdES standard but the library doesn't offer support for every single detail that is allowed in XAdES standard. The library supports XadES-X-L standard with SignatureTimeStamp and SignAndRefsTimeStamp but only provides verification of such signatures.

References

- RFC2560 - Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, C., X.509 Internet Public Key Infrastructure: Online Certificate Status Protocol - OCSP. June 1999.
- RFC3275 - Eastlake 3rd D., Reagle J., Solo D., (Extensible Markup Language) XML-Signature Syntax and Processing. (XML-DSIG) March 2002.
- ETSI TS 101 903 - XML Advanced Electronic Signatures (XAdES). February 2002.
- XML Schema 2 - XML Schema Part 2: Datatypes. W3C Recommendation 02 May 2001

  http://www.w3.org/TR/xmlschema-2/
- DAS - Estonian Digital Signature Act
- DigiDoc format - DigiDoc file format. http://www.id.ee/files/digidoc_vorming_1_3.pdf
- XML-ENC -

Dependencies

JDigiDoc library depends on the following modules/libraries:

- Java2 – JDK/JRE 1.3.1 or newer
- Apache XML Security – used for XML canonicalization
- XML parser – Apache Xerces. Required for the Apache XML Security library. Unfortunately one cannot replace Xerces with JAXB or another parser because it's been hard coded in Apache XML Security library. One could use xerces also for normal xml parsing and thereby use only one xml parser, but we don't want to make this a requirement. One can use the system default XML parser for normal digidoc parsing as long as it supports SAX interface and then one still needs xerces for canonicalization.
- Xalan – Version 2.2D13 or newer. Required for the Apache XML Security library.
- IAIK JCE cryptographic library – Required only if one uses IAIKNotaryFactory class. The latter is one possible implementation of a module handling OCSP confirmations.
- Bouncy-Castle cryptographic library – Used in all other cryptographic operations. This library was chosen as it is a freeware module. Please note that you can replace IAIKNotaryFactory module with BouncyCastleNotaryFactory class that offers the equivalent OCSP functionality but requires only BouncyCastle library (freeware) and thus you need no proprietary librarys for using JDigiDoc. This only works with BouncyCastle 1.20 or newer and that's the reason IAIK JCE was a requirement earlier.
- Jakarta Log4j - Required for the Apache XML Security library and by JDigiDoc itself for logging purposes.

Environment

One needs to add the following libraries to the CLASSPATH environment variable in order to use JDigiDoc:

- JDigiDoc.jar – JDigiDoc library itself
- jce-jdk13-114.jar – Bouncy-Castle Java cryptographic library. One can use a a newer version of this library if available.
- iaik_jce.jar – IAIK Java cryptographic library. Ainult if you are using also IAIKNotaryFactory module.
- jakarta-log4j-1.2.6.jar - Jakarta Log4j library
- xmlsec.jar – Apache XML Security library
- xalan.jar, xercesImpl.jar, xml-apis.jar, xmlParserAPIs.jar – Xalan and Xerces

If you want to create RSA-SHA1 digital signatures using a smartcard and card reader device and access the latter using a PKCS#11 driver, then add to CLASSPATH the following:

- iaikPkcs11Wrapper.jar

and copy to a directory listed in PATH environment variable (for example c:\windows\system32 on win32) the following DLL-s:

- esteid-pkcs11.dll
- Pkcs11Wrapper.dll

In Linux environment one has to copy the following shared object libraries:

- libesteid-pkcs11.so or opesc-pkcs11.so
- libpkcs11wrapper.so

to the directory {JAVA_HOME}\jre\lib\i386.

Configuring JDigiDoc

JDigiDoc uses the class ee.sk.utils.ConfigManager for reading configuration data from a Java property file – JDigiDoc.cfg. This file can be inside the JDigiDoc.jar library or in any other location referenced by CLASSPATH. One can store it also in another location, not referenced by CLASSPATH and pass the full filename to the ConfigManager.init() method.

Configuration file – JDigiDoc.cfg has the following entires:

- Exchangeable modules. If you wish to replace one of the standard modules with your implementation then place the module in CLASSPATH and register it's class name here.

```
# module used to create signatures – default = use smartcard over PKCS#11
DIGIDOC_SIGN_IMPL=ee.sk.digidoc.factory.PKCS11SignatureFactory
# module used to get OCSP confirmations – now no longer required
#DIGIDOC_NOTARY_IMPL=ee.sk.digidoc.factory.IAIKNotaryFactory
# module used to verify timestamps
DIGIDOC_TIMESTAMP_IMPL=ee.sk.digidoc.factory.BouncyCastleTimestampFactory
# module used to get OCSP confirmations – free ware module
DIGIDOC_NOTARY_IMPL=ee.sk.digidoc.factory.BouncyCatsleNotaryFactory
# module used to parse digidoc files – default = use SAX parser
DIGIDOC_FACTORY_IMPL=ee.sk.digidoc.factory.SAXDigiDocFactory
# module used to canonicalize XML – default = use Apache XML Security library
CANONICALIZATION_FACTORY_IMPL=ee.sk.digidoc.factory.DOMCanonicalizationFactory
# optional module used to check signatures with CRL – not required
CRL_FACTORY_IMPL=ee.sk.digidoc.factory.CRLCheckerFactory
# XML-ENC parsers
ENCRYPTED_DATA_PARSER_IMPL=ee.sk.xmlenc.factory.EncryptedDataSAXParser
ENCRYPTED_STREAM_PARSER_IMPL=ee.sk.xmlenc.factory.EncryptedStreamSAXParser
```

- Java cryptographic libraries.

```
# default security library used in JDigiDoc – required even if you use
# also IAIK for other stuff like OCSP
DIGIDOC_SECURITY_PROVIDER=org.bouncycastle.jce.provider.BouncyCastleProvider
DIGIDOC_SECURITY_PROVIDER_NAME=BC
# used to be required for OCSP handling but newer BouncyCastle offers
# equivalent functionality, so you don't need this any more
IAIK_SECURITY_PROVIDER=iaik.security.provider.IAIK
```

- PKCS#11 settings

```
# EstID smartcard driver (used by private persons)
DIGIDOC_SIGN_PKCS11_DRIVER=esteid-pkcs11
# AID smartcard driver (used for example by companies)
#DIGIDOC_SIGN_PKCS11_DRIVER=pk2priv
# JNI shared object / dynamic link library to access PKCS#11
DIGIDOC_SIGN_PKCS11_WRAPPER=pkcs11rapper
# verification algorithm used in JDigiDoc
DIGIDOC_VERIFY_ALGORITHM=RSA/NONE/PKCS1Padding
```

- OCSP confirmation settings

```
# OCSP responders URL
DIGIDOC_OCSP_RESPONDER_URL=http://ocsp.sk.ee
# your HTTP proxy server settings if applicable
DIGIDOC_PROXY_HOST=<your proxy server name>
DIGIDOC_PROXY_PORT=<your proxy server port>
```

```
# Flag: sign your OCSP confirmations or not. Please not that most private
# persons have to sign the OCSP requests to get access to the server.
SIGN_OCSP_REQUESTS=true (or “false” for sending unsigned OCSP requests)

# OCSP confirmation server access token
DIGIDOC_PKCS12_CONTAINER=<full path to the PKCS#12 token file>
DIGIDOC_PKCS12_PASSWD=<PKCS#12 tokens password>
DIGIDOC_OCSP_SIGN_CERT_SERIAL=<serial number of the PKCS#12 tokens certificate
issued to you>

# OCSP responders certificate files and CN attributes
DIGIDOC_OCSP_COUNT=2
DIGIDOC_OCSP1_CN=ESTEID-SK OCSP RESPONDER
DIGIDOC_OCSP1_CERT=<path>\\esteid-ocsp.pem
DIGIDOC_OCSP1_CA_CERT=<path>\\esteid.pem
DIGIDOC_OCSP1_CA_CN=ESTEID-SK
DIGIDOC_OCSP2_CN=KLASS3-SK OCSP RESPONDER
DIGIDOC_OCSP2_CERT=<path>\\KLASS3-SK-OCSP.pem
DIGIDOC_OCSP2_CA_CERT=<path>\\KLASS3-SK.pem
DIGIDOC_OCSP2_CA_CN=KLASS3-SK

# OCSP or CRL selectors
DIGIDOC_CERT_VERIFIER=OCSP
DIGIDOC_SIGNATURE_VERIFIER=OCSP
```

- Logging settings

```
# log4j configurations file path. CHANGE THIS !
DIGIDOC_LOG4J_CONFIG=<path>\\SignatureLogging.properties
```

- CA certificate locations

```
DIGIDOC_CA_CERTS=3
DIGIDOC_CA_CERT1=<path>\\juur.pem
DIGIDOC_CA_CERT2=<path>\\esteid.pem
DIGIDOC_CA_CERT3=<path>\\KLASS3-SK.pem
```

- CRL settings if required

```
CRL_USE_LDAP=false
CRL_FILE=esteid.crl
CRL_URL=http://www.sk.ee/crls/esteid/esteid.crl
CRL_SEARCH_BASE=cn=ESTEID-SK,ou=ESTEID,o=AS Sertifitseerimiskeskus,c=EE
CRL_FILTER=(certificaterevocationlist;binary=*)
CLR_LDAP_DRIVER=com.ibm.jndi.LDAPCtxFactory
CRL_LDAP_URL=ldap://194.126.99.76:389
CRL_LDAP_ATTR=certificaterevocationlist;binary
CRL_PROXY_HOST=<your proxy server name>
CRL_PROXY_PORT=<your proxy server port>
```

- Encryption settings

```
DIGDOC_ENCRYPT_KEY_ALG=AES
DIGIDOC_ENCRYPTION_ALOGORITHM=AES/CBC/PKCS7Padding
DIGIDOC_SECRANDOM_ALGORITHM=SHA1PRNG
DIGIDOC_KEY_ALOGORITHM=RSA/NONE/PKCS1Padding
```

- Timestamping settings

```
DIGIDOC_TSA_COUNT=1
DIGIDOC_TSA1_CERT=jar://certs/tsa_ns.crt
DIGIDOC_TSA1_CA_CERT=jar://certs/ts_cacert.crt
DIGIDOC_TSA1_USE_NONCE=true
DIGIDOC_TSA1_ASK_CERT=true
DIGIDOC_TSA1_URL=http://ns.szikszi.hu:8080/tsa
DIGIDOC_TSA1_CN=OpenTSA demo
DIGIDOC_TSA1_CA_CN=OpenTSA Root
DIGIDOC_TSA1_SN=12
```

```
MAX_TSA_TIME_ERR_SECS=1
```

JDigiDoc uses only a part of Apache XML Security library for XML canonicalization. Unfortunately this library requires one the put references to DTD in one's XML documents and outputs lots of warnings if it doesn't find such references. One way of discarding those warnings is to set the main logger in Log4j config file very restrictive and then selectively enable logging only for those components that you wish. For example:

```
# root logger properties
log4j.rootLogger=FATAL, output

# JDigiDoc loggers
log4j.logger.ee.sk.utils.ConfigManager=DEBUG, output
log4j.logger.ee.sk.digidoc.DigiDocException=DEBUG, output
log4j.logger.ee.sk.digidoc.factory.IAIKNotaryFactory=DEBUG, output
log4j.logger.ee.sk.digidoc.factory.SAXDigiDocFactory=DEBUG, output
log4j.logger.ee.sk.digidoc.factory.PKCS11SignatureFactory=INFO, output
log4j.logger.ee.sk.xmlenc.factory.EncryptedDataSAXParser=INFO, output
log4j.logger.ee.sk.xmlenc.factory.EncryptedStreamSAXParser=INFO, output
log4j.logger.ee.sk.digidoc.DataFile=INFO, output
log4j.logger.ee.sk.xmlenc.EncryptedData=INFO, output
log4j.logger.ee.sk.xmlenc.EncryptedKey=INFO, output
log4j.logger.ee.sk.digidoc.Base64Util=INFO, output

#setup output appender
log4j.appender.output =org.apache.log4j.ConsoleAppender
log4j.appender.output.layout=org.apache.log4j.PatternLayout
log4j.appender.output.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss}
[%c{1},%p] %M; %m%n
```

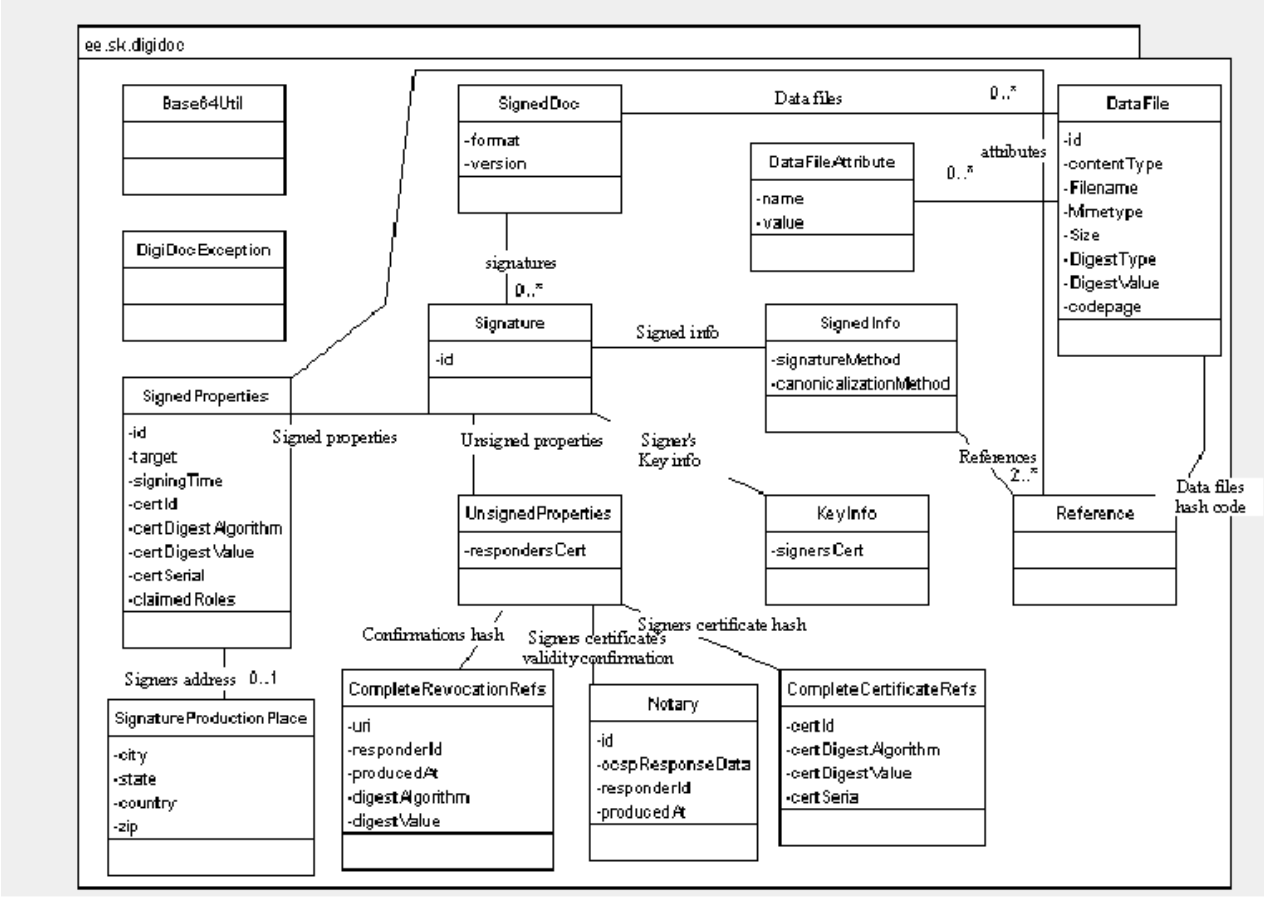
JDigiDoc architecture

JDigiDoc library consists of the following packages:

- ee.sk.digidoc – core classes of JDigiDoc modeling the structure of various XML-DSIG and XAdES entities.
- ee.sk.digidoc.factory – Exchangeable modules implementing various functionality that you might wish to modify and interfaces to those modules.
- ee.sk.utils – Configuration and other utility classes
- ee.sk.test – Sample programs.

Package: ee.sk.digidoc

This package contains the core JDigiDoc classes modeling the structure of various XML-DSIG and XAdES entities.

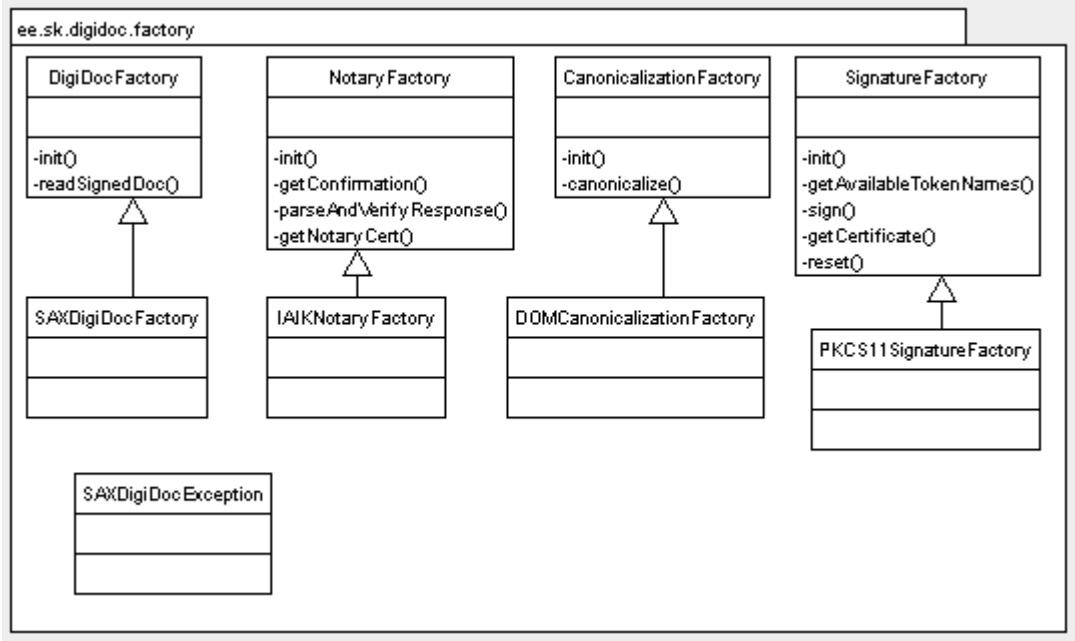


- ee.sk.digidoc.SignedDoc – Models a digitally signed document in DIGIDOC-XML 1.3, 1.2 or 1.1 format. One document can contain one or more <DataFile> elements (payload data) and zero or more signatures (<Signature> element). Please use only the latest format version 1.3 for full XAdES conformity.
- ee.sk.digidoc.DataFile – Payload data. Contains data from an external file that is being signed. Files can be added in EMBEDDED_BASE64 mode (binary files), EMBEDDED mode (pure XML or text) and DEATCHED mode (no actual data, just a reference to an external file and it's hash code).
- ee.sk.digidoc.DataFileAttribute – optional user defined attributes for <DataFile> element. This way one can add some attributes to the actual data that are not part of the original file but also get signed. Required attributes are: Id, Filename, Contenttype, MimeType, Size. In case of using DETATCHED mode the following attributes are also required: DigestType, DigestValue, Codepage. User can add any attributes that don't collide with the names of required attributes.
- ee.sk.digidoc.Signature – Digital signature (XML-DSIG)
- ee.sk.digidoc.SignedInfo – actually signed content containing references and hash codes of other signed data objects. (XML-DSIG)
- ee.sk.digidoc.Reference – a reference to a signed data object and it's hash code (XML-DSIG).
- ee.sk.digidoc.KeyInfo – Signers certificate data. (XML-DSIG). Please not that even if XML-DSIG allows various other possibilities of establishing the signers identity like sending the signers distinguished name etc. we still require the whole signers certificate to be included in this element.
- ee.sk.digidoc.SignedProperties – additional signed properties (XAdES)

- ee.sk.digidoc.SignatureProductionPlace – optional signers address (XAdES).
- ee.sk.digidoc.UnsignedProperties – unsigned signature properties (XAdES)
- ee.sk.digidoc.Notary – OCSP confirmation
- ee.sk.digidoc.CompleteCertificateRefs – signers certificate info and hash code (XAdES).
- ee.sk.digidoc.CompleteRevocationRefs – OCSP confirmation info and hash code (XAdES).
- ee.sk.digidoc.DigiDocException – specific Exception class for JDigiDoc exceptions.
- ee.sk.digidoc.Base64Util – Base64 decoder and encoder.
- ee.sk.digidoc.CertID – contains certificate info (not certificate itself). Such info is stored in XAdES documents in <SignersCertificate> and <Cert> elements. It contains certificates serial number, issuer DN and hash code.
- ee.sk.digidoc.CertValue – contains certificate itself. Certficates are added to XAdES signatures <CertificateValues> section and they are used when verifying the signature.
- ee.sk.digidoc.TimestampInfo – contains timestamp info. This version of JDigiDoc supports XadES-X-L format with <SignatureTimeStamp> and <SigAndRefsTimeStamp>, but provides only verification of such documents.

Package: ee.sk.digidoc.factory

This package contains interfaces for exchangeable modules and their implementations.



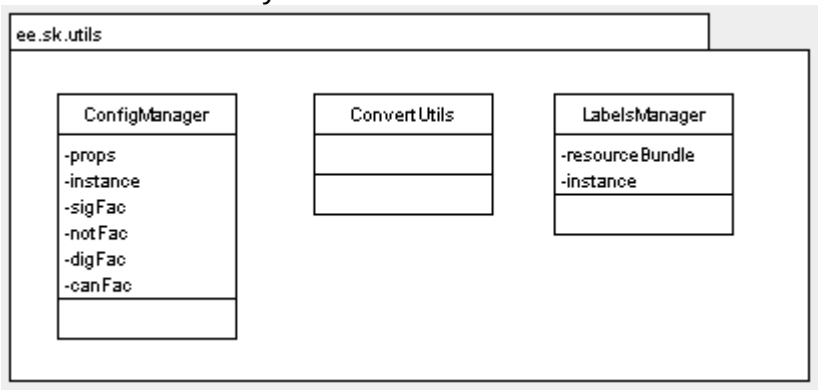
- ee.sk.digidoc.factory.DigiDocFactory – interface for digidoc file parser
- ee.sk.digidoc.factory.SAXDigiDocFactory – implementation parsing digidoc documents using a SAX parser.
- ee.sk.digidoc.factory.SAXDigiDocException – DigiDoc specific parsing exception class that is a subclass of SAXException.
- ee.sk.digidoc.factory.NotaryFactory – interface for getting OCSP confirmations.
- ee.sk.digidoc.factory.IAIKNotaryFactory - Implementation module for getting OCSP confirmations. This modules uses IAIK JCE library. You have to get a license of IAIK-JCE to use this.
- ee.sk.digidoc.factory.BouncyCastleNotaryFactory - Implementation module for getting

OCSP confirmations. This modules uses BouncyCastle library and requires no license.

- ee.sk.digidoc.factory.CanonicalizationFactory – interface for XML canonicalization.
- ee.sk.digidoc.factory.DOMCanonicalizationFactory – Implementation for XML canonicalization using the Apache XML Security library.
- ee.sk.digidoc.factory.SigantureFactory – Interface for signing.
- ee.sk.digidoc.factory.PKCS11SigantureFactory – Implementation for signature module using smartcards and a PKCS#11 driver. This module uses IAIK PKCS#11 wrapper library to access external native language PKCS#11 drivers. IAIK PKCS#11 wrapper is free ware.
- ee.sk.digidoc.factory.TimestampFactory – interface for timestamp operations.
- ee.sk.digidoc.factory.BouncyCastleTimestampFactory – implementation for timestamp operations using BouncyCastle library. In this version provides only verification of timestamps.

Package: ee.sk.util

This package contains various utility classes.



- ee.sk.util.ConfigManager – configuration data utility.
- ee.sk.util.ConvertUtils – contains data conversion methods.
- ee.sk.util.LabelsManager – manages labels for GUI, not used at the moment.

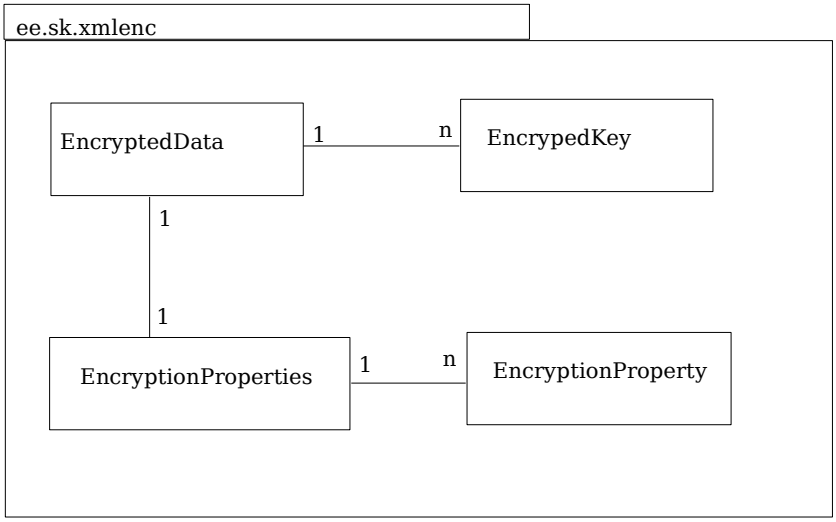
Package: ee.sk.test

This package contains sample programs

- ee.sk.test.Test1 – creates a digidoc document in format: DIGIDOC-XML, version: 1.3 adds three data files, one in each possible mode, signs the document using Estonian ID card, card reader and a PKCS#11 driver, gets an OCSP confirmation to this signature, writes everything in a digidoc file, reads the file in again and verifies signatures and OCSP confirmations.
- ee.sk.test.Test2 – Reads in a digidoc document and verifies signatures and OCSP confirmations.
- ee.sk.test.jdigidoc – commandline utility program that allowes one to invoke most of the functionality offered by the library.

Package: ee.sk.xmlenc

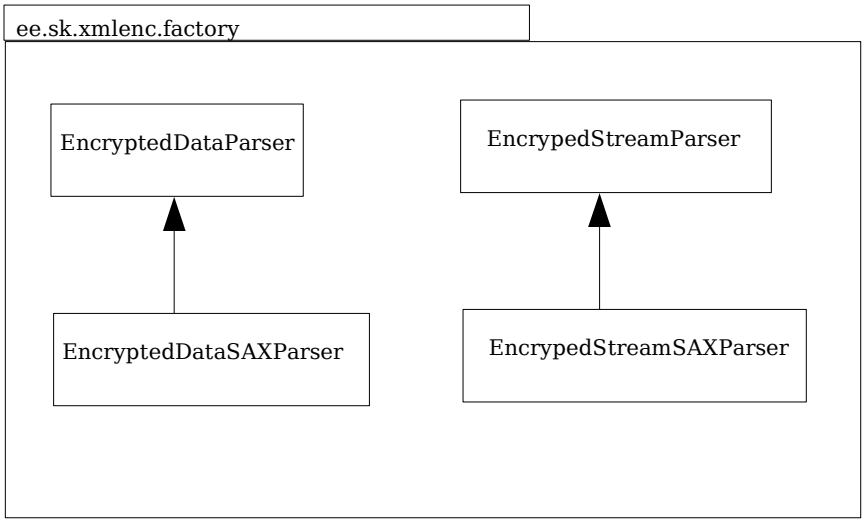
This package contains classes modelling the XML entitys specified in XML-ENC standard.



- ee.sk.xmlenc.EncryptedData – Corresponds to the <EncryptedData> element in XML-ENC standard. JDigiDoc doesn't support many encrypted data objects or a mix of encrypted and unencrypted data in one xml document. One encrypted document can have only one <EncryptedData> element, which is also the documents root element and it contains one <EncryptedKey> element for every recipient (e.g. Possible decrypter) of the document and a set of <EncryptionProperty> elements to store any meta data.
- ee.sk.xmlenc.EncryptedKey – Corresponds to the <EncryptedKey> element of XML-ENC standard. One encrypted document contains on <EncryptedKey> element per every recipient that contains the 128 bit AES transport key encrypted by the recipients certificate and any other data used to identify the key.
- ee.sk.xmlenc.EncryptionProperties - Corresponds to the <EncryptionProperties> element of XML-ENC standard. It contains one or more EncryptionPropety objekts.
- ee.sk.xmlenc.EncryptionProperty - Corresponds to the <EncryptionProperty> element of XML-ENC standard. XML-ENC standard defines optional atributes “Id” and “Target” and allowes to add any custom atributes. JDigiDoc uses the atribute “Name” to identify special elements that the library uses to store metadata.

Package: ee.sk.xmlenc.factory

This package contains parsers of encrypted files.



- ee.sk.xmlenc.factory.EncryptedDataParser – defines a general interface for parsing encrypted files. Suitable for only smaller files because collects all data in memory
- ee.sk.xmlenc.factory.EncryptedDataSAXParser – implements the EncryptedDataParser interface with a SAX parser.
- ee.sk.xmlenc.factory.EncryptedStreamParser – defines an interface for decrypting encrypted files. Suitable for larger files but doesn't keep any document info in memory. Decrypts data directly from inputstream and writes to output stream.
- ee.sk.xmlenc.factory.EncryptedStreamSAXParser- implements the EncryptedStreamParser interface with a SAX parser.

Digital signing

JDigiDoc teek pakub digiallkirjastatud dokumentide koostamist, allkirjastamist, allkirjade kontrolli ja lugemist vastavalt XAdES (ETSI TS101903) ja XML-DSIG standarditele. Järgnevates peatükkides kirjeldatakse digiallkirjastase põhilisi operatsioone JDigiDoc teegiga.

Initialization

Before you can use JDigiDoc, you must initialize it be reading in configuration data. This is necessary because the library needs to know the location of CA certificates and other parameters in order to fulfill your requests. Pass the full path and name of the configuration file to library like that:

```
ConfigManager.init("jar://JDigiDoc.cfg");
```

The configuration file can be embedded in JDigiDoc jar file which is indicated by the “jar://” prefix. Otherwise just pass the normal full filename in your platform like “/etc/jdigidoc.conf”.

Creating a digidoc document

Before you can add payload data to digidoc document you must create a SignedDoc object to receive this data:

```
SignedDoc sdoc = new SignedDoc(SignedDoc.FORMAT_DIGIDOC_XML,
SignedDoc.VERSION_1_3);
```

One can use also the older format versions 1.1 and 1.2 but those are not fully XAdES conform and are maintained only for historical reasons and backward compatibility.

(`SignedDoc.VERSION_1_2, SignedDoc.VERSION_1_1`). The new digidoc object is kept in memory and not immediately written to a file.

Adding data files

Payload data can be added in EMBEDDED_BASE64 mode (binary files), EMBEDDED mode (pure XML or text) or DEATCHED mode (reference to an external file).

- EMBEDDED_BASE64 – use this to embed binary data in base64 encoding:

```
DataFile df = sdoc.addDataFile(new File(<full-filename-with-path>),
          <mime-type>, DataFile.CONTENT_EMBEDDED_BASE64);
```

- EMBEDDED – use this to embed pure text or XML:

```
DataFile df = sdoc.addDataFile(new File(<full-filename-with-path>),
          <mime-type>, DataFile.CONTENT_EMBEDDED);
```

- DETATCHED – use this to add a reference to an external file (file to big etc.):

```
DataFile df = sdoc.addDataFile(new File(<full-filename-with-path>),
          <mime-type>, DataFile.CONTENT_DETATCHED);
```

The new objects are created in memory but the references files are not read from the disk yet. The library reads all files when you write the digidoc document to a file or start adding signatures (because one has to know the hash codes for signing the data)

If you don't want JDigiDoc to read the files because you hold the data in a database, generate it dynamically etc., then read it yourself and assign to the DataFile object using the setBody() method. Once you have assigned the data to this object it no longer will read the file from disk:

```
String myBody = "My sample data with umlauts äüõ";
df.setBody(myBody.getBytes("ISO-8859-1"), "ISO-8859-1");
```

It is your task to know what code page is used by your data. In this example we use the ISO-8859-1 code page. Now JDigiDoc stores the original code page in the Codepage="ISO-8859-1" attribute of <DataFile> to be able to convert the data back to original code page later. All data in digidoc documents is kept in UTF-8, so we had to convert the data in the previous sample. If your data is already in UTF-8 the do the following:

```
byte[] u8b = ConvertUtils.str2data(myBody);
df.setBody(u8b, "UTF-8");
```

If the DataFile contains pure XML or text the you can get the data using:

```
String body = df.getBodyAsString();
```

Adding signatures

One has to use the SignatureFactory interface for signing. One can sign using an Estonian ID card or any other smartcard provided that you have the external native language PKCS#11 driver for it or calculate the signature in some external program (web-application?) and the add the signature value to digidoc document. Before signing you have to get the signers certificate that is being referenced by the signature. If you use PKCS#11 driver to access smartcard then do:

```
String pin = "<smartcard-PIN>";
SignatureFactory sigFac = ConfigManager.
                instance().getSignatureFactory();
# Index of the key pair used for signing. In this sample we use 0 as
# it's used on Estonian ID cards. Please not that this index starts with 0
# and counts ONLY the key pairs usable for digital signature.
# e.g. no authentication key pairs!
X509Certificate cert = sigFac.getCertificate(0, pin);
```

Now sign compute the payload data hash codes and create a Signature object:

```
Signature sig = sdoc.prepareSignature(cert,
                    null, // String[] claimedRoles,
                    null); // SignatureProductionPlace (address)
byte[] sidigest = sig.calculateSignedInfoDigest();
```
The signature is not complete yet as it's missing the actual RSA signature data, but we have now calculated the final 20 byte SHA1 hash that serves as input for RSA signature. You can use an external program to get the RSA signature value or compute it with a smartcard if you have the required PKCS#11 driver:

```
byte[] sigval = sigFac.sign(sidigest, 0, pin);
```
Now add signature value to the Signature object.

```
sig.setSignatureValue(sigval);
```

Adding an OCSP confirmation

One has to use the NotaryFactory interface to get an OCSP confirmation, but the required method is in the Signature object itself:

```
sig.getConfirmation();
```
After adding an OCSP confirmation the signature is now complete and provides long-time proof of the signed data.

If you use an external application to get OCSP confirmations, you don't use digidoc documents or want to implement confirmation functionality yourself, then do:

```
NotaryFactory notFac = ConfigManager.
                instance().getNotaryFactory();
Notary not = notFac.getConfirmation(byte[] nonce,
        X509Certificate signersCert, String notId)
        throws DigiDocException;
```
If OCSP confirmation couldn't be acquired (no internet access etc.) then an exception is thrown.

If you just want to verify the validity of a certificate, not sign a document (for example when authentication users to your application) then do:

```
public void NotaryFactory.checkCertificate(X509Certificate cert)
        throws DigiDocException;
```

Reading and writing digidoc documents

Write a SignedDoc object to a digidoc file as follows:

```
sdoc.writeToFile(new File("<full-path-and-filename>"));
```
Read a digidoc document as follows:

```
DigiDocFactory digFac = ConfigManager.instance().getDigiDocFactory();
SignedDoc sdoc = digFac.readSignedDoc("<full-path-and-filename>");
```
If you want to store the digidoc document in database not in a file, then use the method:

```
SignedDoc.writeToStream(OutptStream os).
```

Verifying signatures and OCSP confirmations

After having read a digidoc document verify the signatures as follows:

```
ArrayList errs = sdoc.verify(true, true);
if(errs.size() == 0)
  System.out.println("OK");
for(int j = 0; j < errs.size(); j++)
  System.out.println((DigiDocException)errs.get(j));
```
This method verifies all signatures one by one. If the signature has an OCSP confirmation then this too is being verified. The first parameter to verify() method is a boolean flag determining if the signers certificates have to be verified using the certificates start and end

date (stored on the certificate itself). This is a less accurate check than verifying the OCSP signature (which is enable by the second parameter to verify() method). But if you have no OCSP confirmations then it can be useful. It doesn't hurt to do both if you have an OCSP confirmation as well.

The second parameter determines if OCSP confirmations are required and an exception should be generated if it's missing.

In case of signature verification errors no exceptions are actually thrown but they are returned to the user in an ArrayList container. This way you can get all errors and not just the first. If the returned container was empty then the document verified ok.

Validating digidoc documents

JDigiDoc classes contain methods that validate the contents of their fields and atributes. It is always usefull to validate the document after reading it from a file or after adding or changing some content. This helps to identify problems at later phases. Use the method:

```
ArrayList SignedDoc.validate(boolean bStrong);
```

This method returns an array of DigiDocException objects which could be displayed in some userinterface. If the array is empty then the document is ok. Verifying signatures calls also this method using bStrong=false, so it might still accept some smaller errors if this doesn't invalidate the signatures.

Encryption and decryption

In addition to digital siging the JDigiDoc library offers also digital encryption and decryption according to the XML-ENC standard. This standard describes encrypting and decrypting XML documents or parts of them but it also allowes one to encrypt any binary data in Base64 encoding. JDigiDoc enables one also to compress the data with ZLIB algorithm before encryption. JDigiDoc encrypts data with a 128 bit AES transportkey whitch is in turn encrypted with the recipients certificate. JDigiDoc doesn't support many encrypted data objects or a mix of encrypted and unencrypted data in one xml document. One encrypted document can have only one <EncryptedData> element, which is also the documents root element and it contains one <EncryptedKey> element for every recipient (e.g. Possible decrypter) of the document and a set of <EncryptionProperty> elements to store any meta data. In the following chapters we review most common encryption and decryption operations with JDigiDoc library.

Composing encrypted documents

In order to compose an encrypted document one has to create the EncryptedData object first, then add all recipients certificates and other data, then add the unencrypted data, encrypt and possibly compress it and finally store it in a file or other medium.

```
EncryptedData cdoc = new EncryptedData(
          null, // optional Id atribute value
          null, // optional Type atribute value
          null, // optional Mime atribute value
          EncryptedData.DENC_XMLNS_XMLENC,  // fixed xml namespace
          EncryptedData.DENC_ENC_METHOD_AES128); // fixed cryptographik
algorithm
```

Optional atribute values have to be passed in as nulls in case you don't need them. Passing in for example an empty string will case this to be considered a valid atribute value. If encrypting a digidoc document one should assign the “Type” atribute the value: "http://www.sk.ee/DigiDoc/v1.3.0/digidoc.xsd" which has also been defined as a constant: EncryptedData.DENC_ENCDATA_TYPE_DDOC. If encrypting pure XML documents then

one could assign to atribute “MimeType” the value “text/xml” which has also been defined as a constant: EncryptedData.DENC_ENCDATA_MIME_XML. JDigiDoc library uses the atribute “MimeType” to store the fact that the data has been packed with ZLIB algorithm before encryption. The library assigns to MimeType atribute the value "http://www.isi.edu/in-noes/iana/assignments/media-types/application/zip", which has also been defined as a constant: EncryptedData.DENC_ENCDATA_MIME_ZLIB. You don't have to do this yourself. JDigiDoc assigns this value when packing the data and if the MimeType atribute was not empty before that then the old value is stored in an <EncryptionProperty Name=”OriginalMimeType”> subelement. If JDigiDoc reads a document with this specific MimeType then it decompresses the decrypted data and restores the original mime type if one is found.

Adding recipient info

Every encrypted document should have at least one or many recipient blocks, otherwise nobody can decrypt it. For every recipient the library stores the AES transport key encrypted with the recipients certificate, the certificate itself and possibly some other data used to identify the key. One must use a certificate that is usable for data encryption. In case of Estonian ID cards it's the authentication certificate. For adding an EncryptedKey object one has to have the recipients certificate in PEM format. On can add optional atributes “Id” and “Recipient” and/or sublements <KeyName> and <CerriedKeyName> to identify the key object. All of the abovementioned atributes and sublements are optional but can be used to search for the right recipients key or display it's data in an application. EncryptedKey object is added as follows:

```
X509Certificate recvCert = SignedDoc.readCertificate(new File(certFile));
EncryptedKey ekey = new EncryptedKey(
          null,     // optional Id atribute value
          null,     // optional Recipient atribute value
          EncryptedData.DENC_ENC_METHOD_RSA1_5, // fikseeritud krüptoalgoritm
          null,     // optional KeyName subelement value
          null,     // optional CarriedKeyName subelement value
          recvCert); // recipients certificate. Required!
cdoc.addEncryptedKey(ekey);
```

The commandline utility program ee.sk.test.jdigidoc assigns a unique value to every EncryptedKey objects “Recipient” atribute. It could be the recipients forname or something more complicate like “<last-name>,<first-name>,<personal-code>”. This is later used as a commandline option to identify the recipient who's key and smartcard is used to decrypt the data. As the recipients certificate is the only required data, it would be wise no to demand encrypted documents to contain other atributes for an applications proper functioning. One should use something from the certificate like it's CN atribute to identify the recipient.

Encryption and data storage

There are two possible methods for encrypting data:

- small data objects – does all operations in memory. Faster and more flexible but requires more memory. One can use the compression option “BEST EFFORT” which means that data will be compressed and if this resulted in reduction of data size then it's used , otherwise it will be discarded and original not compressed data is encrypted.
- big data obejcts – reads and handles all data in blocks of fixed size. Capable of encrypting large sets of data but less flexible. One must specify compression or no compression for this operation. Doesn't offer encrypting in memory. One has to provide

input and output streams.

For encrypting small data objects one must first construct the EncryptedData objekt, then add all recipient info, then add the unencrypted data, encrypt it and finally store in a file or another medium. For example:

```
// read unencrypted data
byte[] inData = SignedDoc.readFile(new File(inFile));
cdoc.setData(inData);
cdoc.setDataStatus(EncryptedData.
DENC_DATA_STATUS_UNENCRYPTED_AND_NOT_COMPRESSED);
// store the original filename and or mime type if applicable
cdoc.addProperty(EncryptedData.ENCPROP_FILENAME, inFile);
// Encryption. Options: EncryptedData.DENC_COMPRESS_ALLWAYS,
// EncryptedData.DENC_COMPRESS_NEVER and
EncryptedData.DENC_COMPRESS_BEST_EFFORT
cdoc.encrypt(EncryptedData.DENC_COMPRESS_BEST_EFFORT);
FileOutputStream fos = new FileOutputStream(outFile);
fos.write(m_cdoc.toXML());
fos.close();
```

For encrypting bigger data sets one also has to first construct EncryptedData object and register all recipients. Then add any metadata and encrypt the data by reading input stream, encrypting and possibly compressing the data and writing to output stream.

```
// store metadata such as the original file name.
cdoc.addProperty(EncryptedData.ENCPROP_FILENAME, inFile);
// Encryp. Compression options are only EncryptedData.DENC_COMPRESS_ALLWAYS
// and EncryptedData.DENC_COMPRESS_NEVER
cdoc.encryptStream(new FileInputStream(inFile),
          new FileOutputStream(outFile), EncryptedData.DENC_COMPRESS_ALLWAYS);
```

In both cases one doesn't necessarily have to use files to store encrypted data. One can write it to any output stream and use as required.

Parsing and decrypting

There are also two options for decrypting and parsing encrypted documents:

- EncryptedDataParser – suitable for parsing smaller encrypted objects. After parsing data is in memory and can be decrypted or displayed on screen. Doesn't automatically decrypt data during parsing. Decryption is a separate operation.
- EncryptedStreamParser – suitable for parsing and decrypting large encrypted objects. Doesn't keep any data in memory. One has to provide input and outputstreams. Decryption and decompression is done during parsing. One has to provide also a way to identify the EncryptedKey object to use for decryption. The current implementation uses the Recipient atribute for this purpose.

Parsing small encrypted files is done as follows:

```
EncryptedDataParser dencFac =  ConfigManager.instance().
          getEncryptedDataParser();
cdoc = dencFac.readEncryptedData(inFile);
```

Now all data is in memory in encrypted and possibly in compressed form. One can use the methods of EncryptedData, EncryptedKey and EncryptionProperty objects to display and

decrypt data as follows:

```
m_cdoc.decrypt(0,  // index of EncryptedKey object
          0,      // smartcards Tokeni index. For Eestoinin ID cards allways 0
          pin);   // smartcards PIN code. For Eestoinin ID card PIN1
FileOutputStream fos = new FileOutputStream(outFile);
fos.write(m_cdoc.getData());
fos.close();
```

For decrypting big encrypted documents use:

```
// provide input and outputstreams
FileInputStream fis = new FileInputStream(inFile);
FileOutputStream fos = new FileOutputStream(outFile);
EncryptedStreamParser streamParser = ConfigManager.
          instance().getEncryptedStreamParser();
// decrypt and possibly decompress the data
streamParser.decryptStreamUsingRecipientName(fis, fos,
          0,  // smartcards Tokeni index. For Eestoinin ID cards allways 0
          pin, // smartcards PIN code. For Eestoinin ID card PIN1
          recvName); // selected EncryptedKey objects Recipient atribute
fos.close();
fis.close();
```

Data is read from input stream decrypted, possibly decompressed and written to output stream.

JDigiDoc utility

There is a commandline utility programm that one can use to test the library or simply use it directly to encrypt, decrypt and digitally sign documents.

General commands

- **-? or -help** – displays help about command syntax
- **-config <configuration-file>** - enables one to specify the JDigiDoc configfile name. Default is “jar://jdigidoc.cfg”.
- **-check-cert <certficate-file-in-pem-format>** - checks the certificate status by OCSP.

Digital signature commands

- **-ddoc-in <input-digidoc-file>** - specifying the input digidoc document name.
- **-ddoc-new [format] [version]** – Creates a new digidoc document in the specified format and version. Default formaat is 1.3 (newest).
- **-ddoc-add <input-file> <mime-type> [content-type]** – adds a new data file to a digidoc document. If digidoc doesn't exist then creates one in the default format.
- **-ddoc-sign <pin-code> [manifest] [country] [state] [city] [zip]** – adds a digital signature to the digidoc document.
- **-ddoc-out <ouput-file>** - stores the newly created or modified digidoc document in a file.
- **-ddoc-list** – displays the data file and signature info of a digidoc just read in. Verifies all signatures.

Encryption commands

- **-cdoc-in <input-encrypted-file>** - specifying the input encrypted document name
- **-cdoc-list** – displays the encrypted data info and recipients info of an encrypted document just read in.
- **-cdoc-recipient <certificate-file> [recipient] [KeyName] [CarriedKeyName]** – adds a new recipient certificate and other metadata to an encrypted document.
- **-cdoc-encrypt <input-file> <output-file>** - encrypts the data from the given input file and writes the completed encrypted document in a file (small document interface)
- **-cdoc-encrypt-stream <input-file> <output-file>** - encrypts the input file and writes to output file (large document interface)
- **-cdoc-decrypt <pin> <output-file>** - decrypts and possibly decompresses the encrypted file just read in and writes to output file (small document interface).
- **-cdoc-decrypt-stream <input-file> <recipient> <pin> <output-file>** - decrypts and possibly decompresses the inputfile and writes to outputfile (large document interface).

ChangeLog

This chapter contains a list of changes applied in various releases

JDigiDoc 2.1.6

- Fixed a bug in displaying UTF-8 format filenames on win32 platforms
- Fixed a bug in selecting the correct OCSP responders certificate if there were many possibilities. Now the algorithm is to check the OCSP response with all listed responders certificates until the signature verifies with one of them, which is then selected and added to the digidoc document.
- Added new SK-EID CA certificate and new responder certificates: "KLASS3-SK OCSP RESPONDER" and "EID-SK OCSP RESPONDER" in the config file. Old OCSP responder certficates will still be necessary to verify old documents. Configuring more than one OCSP responder certificate can be done like in the following sample:

File: jdigidoc.cfg

```
...
DIGIDOC_OCSP2_CN=KLASS3-SK OCSP RESPONDER
# first alternative to check (newest?)
DIGIDOC_OCSP2_CERT=jar://certs/sk-klass3-ocsp-responder-2006.cer
# second certificate to check....
DIGIDOC_OCSP2_CERT_1=jar://certs/sk-klass3-ocsp-responder.pem
DIGIDOC_OCSP2_CA_CERT=jar://certs/sk-klass3.pem
DIGIDOC_OCSP2_CA_CN=TEST-SK
...
```

JDigiDoc 2.3.2

- Added timetsamping functionality. This version now supports XadES-X-L format with <SignatureTimeStamp> and <SigAndRefsTimeStamp>, but provides only verification of such documents. Added new classes ee.sk.digidoc.CertID, ee.sk.digidoc.CertValue, ee.sk.digidoc.TimestampInfo, ee.sk.digidoc.factory.TimestampFactory and ee.sk.digidoc.factory.BouncyCastleTimestampFactory.

- Signing and verification operations haven't changed. The library just supports a new format 1.4 for the timestams mentioned above.

File: jdigidoc.cfg

Add the following new config entries:

```
...
DIGIDOC_TIMESTAMP_IMPL=ee.sk.digidoc.factory.BouncyCastleTimestampFactory
...
DIGIDOC_TSA_COUNT=1
DIGIDOC_TSA1_CERT=jar://certs/tsa_ns.crt
DIGIDOC_TSA1_CA_CERT=jar://certs/ts_cacert.crt
DIGIDOC_TSA1_USE_NONCE=true
DIGIDOC_TSA1_ASK_CERT=true
DIGIDOC_TSA1_URL=http://ns.szikszi.hu:8080/tsa
DIGIDOC_TSA1_CN=OpenTSA demo
DIGIDOC_TSA1_CA_CN=OpenTSA Root
DIGIDOC_TSA1_SN=12
MAX_TSA_TIME_ERR_SECS=1
```

The last entry – MAX_TSA_TIME_ERR_SECS – specifies the permitted difference of OCSP reponder servers clock and timestamping servers clock in seconds. Library will verify, that SignatureTimeStamp -s time is before SigAndRefsTimeStamp -s time and that OCSP confirmations time is between the two timestamps times.

JDigiDoc 2.3.15

- Fixed a bug in handling DataFile of type EMBEDDED on win32 platform
- update 2007 CA certificates and corrected sample config files
- fixed bug in handling padding during decryption
- added method for decrypting input stream and passing in decrypted transport key:

```
int EncryptedStreamParser.decryptStreamUsingRecipientNameAndKey(InputStream
dencStream, OutputStream outs, byte[] deckey, String recipientName) throws
DigiDocException;
```

- added new testprogram: ee.sk.test.DecryptTest1 to test the above mentioned method.
- Added new method for reading signatures from stream. This can be used for accepting signatures constructed by DigiDocService during mobile signing.

```
/**
 * Reads in only one <Signature>
 * @param sigStream opened stream with Signature data
 * The user must open and close it.
 * @return signed document object if successfully parsed
 */
public Signature DigiDocFactory.readSignature(InputStream sigStream)
                    throws DigiDocException;
```

- Added new method for accessing DataFile body data as byte array. This method decodes base64 form of data if necessary.

```
/**
 * Accessor for body attribute.
 * Returns the body as a byte array. If body contains
 * embedded base64 data then this is decoded first
 * and decoded actual payload data returned.
 * @return body as a byte array
 */
```

```
public byte[] DataFile.getBodyAsData();
```

JDigiDoc 2.3.22

- Reimplemented Base64Utils.decode(byte[]) to improve memory usage. Added a testprogram ee.sk.test.DecodeBase64Test1 to test it.
- Added config file entry: DIGIDOC_MAX_DATAFILE_CACHED=<max-bytes-cached> to specify maximum number of bytes per <DataFile> object to be cached in memory. Default value is Long.MAX_VALUE. If object size exceeds the limit then the data is stored in temporary file. Changed DataFile.getBody() and DataFile.getBodyAsString() to read temp file if necessary. Added testprogram ee.sk.test.ReadBigDdocTest1 to test using temp files. Changed ee.sk.digidoc.factory.SAXDigiDocFactory to use temp files if necessary.
- Added config file entry: DIGIDOC_DF_CACHE_DIR=<temp-dir> to specify directry for storing temporary files. Default value is read from system param java.io.tmpdir.
- Added “File DataFile.getDfCacheFile()“ to provide read access to temp file name. Either temp file name is not null and body atribute is null or vice versa.
- Added “DataFile.cleanupDfCache()” to remove temporary file for this DataFile. Added “SignedDoc.cleanupDfCache()” to remove all temporary files of this digidoc.
- Added “DataFile.setBodyFromStream(InputStream is)” to set DataFile contents from a stream. This method allways uses temp files and makes a copy of data from stream because the size of data is unknown in beforehand. Caller can close stream immediately after invoking this method. Please note that input data is stored in original format and not converted to base64 at this time. Thus getBody() and getBodyAsString() will not work during composing the digidoc. Only during SignedDoc.write() the conversion is made and then one has to read digidoc in again to be able to use getBody().
- Added “InputStream DataFile.getBodyAsStream()” to read object contents from stream.
- Added “**boolean** DataFile.schouldUseTempFile()” to check wether a temp file will be used. Caller mustuse setSize() before invoking it or pass correct size in constructor.
- Added “**public** File DataFile.createCacheFile()” to have the object generate a temp file for storing temp data. Used from SAX parser.
- Added testprogramm ee.sk.test.WriteBigDdocTest1 to test creating digidoc by passing input file object in constructor.
- Added testprogramm ee.sk.test.WriteBigDdocTest2 to test creating digidoc by passing DataFile contents using setBodyFromStream().