INTESI GROUP

PkBox
Java Software Development Kit

Developer Guideline

Revision 1.3.1

27 November 2020

History

Rev.	Date	Author	Description
1.0	18/01/2016	Daniele Ribaudo	First release.
1.1	22/09/2016	Daniele Ribaudo	Added XAdES signature section.
1.2	03/02/2017	Daniele Ribaudo	Added ASiC signature section.
1.3	07/05/2019	Daniele Ribaudo	Added OTP pushing section.
1.3.1	27/11/2020	Daniele Ribaudo	Fixed XAdES sample code.

Index

Acronyms

Name	Description
CMS	Cryptographic Message Syntax
PDF	Portable Document Format
XML	Extensible Markup Language
CAdES	CMS Advanced Electronic Signatures
PAdES	PDF Advanced Electronic Signatures
XAdES	XML Advanced Electronic Signatures
ASiC	Associated Signature Containers
URI	Uniform Resource Identifier
MAC	Message Authentication Code
CSC	Cloud Signature Consortium
SDK	Software Development Kit
API	Application Programming Interface
PKCS	Public Key Cryptography Standards
MIME	Multipurpose Internet Mail Extensions
SMIME	Secure/Multipurpose Internet Mail Extensions

1 Document scope

This document describes how to use the PkBox SDK functionalities for the creation and validation of electronic signature compliant to CAdES, PAdES, XAdES and ASiC specification.

1.1 Audience

Primary requisites of target audience are familiarity with Java programming language, electronic signature creation, base notions of CAdES, PAdES, XAdES and ASiC specification.

2 SDK architecture

The PkBox SDK is mainly composed by a java class that handles the configuration and connections aspects, and by a set of java classes each one performing a subset of functionalities provided by PkBox Server. These classes are described in the following sections.

The approach adopted in the definition of APIs is to offer as simple as possible interfaces using a high level of abstraction (ideally a call for each function) and minimizing the number of parameters to be used in each call. For more details you may refer to the javadoc.

2.1 PkBox class

`PKBox` class is the "entry" class of the SDK. The calling application has to create, at initialization time, a single `PKBox` class instance for the same pkbox server group. This class is thread-safe.

The user may set one or more pkbox server url. If more than one server is specified, the class automatically manages fault tolerance and load balancing of the calls.

In case a load balancer is already used in the customer infrastructure, the user may set only the load balancer endpoint of the pkbox server.

The PKBox instance can be configured using its instance methods or via `pkbox.properties` configuration file (the first one is preferred). The configuration file must be placed in a JRE classpath directory (see section 8 for a sample).

Although several parameters may be set, the only mandatory parameter is the pkbox server endpoint, that has a format like the follow (refer to the pkbox server configuration):

```
https://<host:port>/pkserver/servlet/defaulthandler
```

By default the SDK writes log messages using the `System.out` java native object. A custom logging system can be set extending the `PkBoxLog` interface and passing it to the PKBox class constructor. Using the `PkBoxLog` interface, the user is free to use any own logging system (e.g log4j, slf4j, etc.).

```
PkBoxLog pklog = new PkBoxLog() {
        public void warn(String msg) {
            logger.warn(msg);
        }

        public void info(String msg) {
            logger.info(msg);
        }

        public void error(String msg, Throwable e) {
            logger.error(msg, e);
        }

        public void error(String msg) {
            logger.error(msg);
        }

        public void debug(String msg) {
            logger.debug(msg);
        }
```

```
};

PKBox pkbox = new PKBox(pklog);
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
```

2.1.1 Secure PIN

The Secure PIN mechanism allows to enforce protection and secure managing of PIN and OTP values. It can be activated in order to raise the level of security for access to the signature credentials.

This mechanism is natively implemented by PkBox SDK and may be enabled by the user through the set of the SecurePIN certificate in the PKBox instance (see `setSecurePINCert` method in the javadoc).

The user can specify the X.509 certificate as a binary data or as a URI to the resource (e.g. the file system path, the http url, etc).

Depending on the pkbox server version you are using, the SecurePIN certificate can be directly retrieved with the following url (refer to the pkbox server configuration):

`https://<host:port>/pkserver/servlet/defaulthandler?action=getsecurepin`

Anyway for performance reason could be useful download the certificate on the file system or use a temporary cache in order to avoid to call pkbox server frequently to retrieve the certificate.

```
pkbox.setSecurePINCert("/Users/developer/temp/SecurePin.cer");
```

NOTE: SecurePIN is mandatory if you are using multiple signatures with transaction (see chapter 5) or if you are using a Certified PkBox server version. In the other cases you may not specify the SecurePIN but for security and privacy reasons is highly advised.

2.2 Envelope class

`Envelope` class is the class implementing pkcs#7, CAdES (p7m), PAdES (pdf) and ASiC signature features. The calling application has to create, at initialization time, a single `Envelope` class instance for each `PKBox` class instance. The class is thread safe, the same class instance can be shared by different threads.

The main supported features are:

- pkcs#7 signature creation
- CAdES signature creation
- CAdES signature validation
- CAdES timestamping
- PAdES signature creation
- PAdES signature validation
- PAdES timestamping
- ASiC signature creation
- ASiC signature validation
- Asymmetric cipher
- Asymmetric decipher
- S/MIME signature creation
- OTP pushing
- Transaction management

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Envelope env = new Envelope(pkbox);
```

2.3. XMLEnvelope class

`XMLEnvelope` class is the class implementing XMLDISG and XAdES signature features. The calling application has to create, at initialization time, a single `XMLEnvelope` class instance for each `PKBox` class instance. The class is thread safe, the same class instance can be shared by different threads.

The main supported features are:

- XAdES signature creation
- XAdES signature validation
- XAdES timestamping

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
XMLEnvelope xmlEnv = new XMLEnvelope(pkbox);
```

2.4. Cipher class

`Cipher` class is the class implementing the symmetrical encryption features. The calling application has to create, at initialization time, a single `Cipher` class instance for each `PKBox` class instance. The class is thread safe, the same class instance can be shared by different threads.

The main supported features are:

- Symmetrical encryption
- Symmetrical decryption
- MAC computation

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Cipher cipher = new Cipher(pkbox);
```

2.5. Utils class

`Utils` class is the class implementing several utility functionalities. The calling application has to create, at initialization time, a single `Utils` class instance for each `PKBox` class instance. The class is thread safe, the same class instance can be shared by different threads.

The main supported features are:

- Raw signature creation
- Timestamp creation
- Timestamp validation
- Certificate validation
- Certificate export
- Retrieving credential info
- Digest computation

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Utils utils = new Utils(pkbox);
```

3. Quick Start

This section describes some very easy samples that allows the user to quickly acquire familiarity with the PkBox SDK, postponing to the javadoc references for a deeper study taking into account the user specific scenarios.

3.1. CAdES signature sample

The following code allows to apply an attached CAdES signature to a file using the method `sign` provided by the `Envelope` class (pay attention: there are several versions of this method with the same behavior, here has been used the one accepting File variables). This method produces the so called p7m file:

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Envelope env = new Envelope(pkbox);
File doc_to_sign = new File("/temp/document_to_sign.pdf");
File signed_doc = new File("/temp/signed_doc.p7m");
String alias = "signer_test";
String pin = "12345678";
String otp = "275855"; // if required by the credential otherwise set to null

/* applying an attached signature to a file */
env.sign(doc_to_sign, "App Tester", alias, pin, otp, null,
         Envelope.implicitMode, Envelope.derEncoding, null, signed_doc);
```

Changing the values passed to the `sign` method, a detached CAdES signature can be produced from the input file, as in the following code (pay attention: if the credential requires OTP, you have to generate a new value if you ran the previous signature sample):

```
/* applying a detached signature to a file */
env.sign(doc_to_sign, "App Tester", alias, pin, otp, null,
         Envelope.explicitMode, Envelope.derEncoding, null, signed_doc);
```

If several signers need to apply a signature to the same large size file, in order to avoid the sending of the file at every sign operation you can create for each signer a `pkcs7` detached envelope starting from the `DigestInfo` (that is considerably smaller) of the file to sign and after merge the file and pkcs7s together to produce the p7m envelope. `Envelope` class provides the method `signdigest` that allows to create such `pkcs7` detached envelope, as shown in the following code (pay attention: if the credential requires OTP, you have to generate a new value if you ran the previous signature sample):

```
Utils utils = new Utils(pkbox);
byte[] digestInfo = utils.digest("test".getBytes(), Utils.sha256,
                                 Utils.derEncoding);

/* create a pkcs#7 detached envelope */
byte[] pkcs7 = env.signdigest(digestInfo, "App Tester", alias, pin, otp, null,
                              Envelope.derEncoding, null);
```

The full source code sample can be found in 7.1, refer to javadoc for parameter details of the `sign` and `signdigest` methods.

3.2. PAdES signature sample

The following code allows to apply an invisible digital signature to a PDF file using the method `pdfsign` provided by the `Envelope` class (pay attention: there are several versions of this method with the same behavior, here has been used the one accepting File variables):

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Envelope env = new Envelope(pkbox);
File doc_to_sign = new File("/Users/developer/temp/doc_to_sign.pdf");
File signed_doc = new File("/Users/developer/temp/signed_doc.pdf");
String alias = "signer_test";
String pin = "12345678";
String otp = "144569"; // if required by the credential otherwise set to null

/* applying an invisible signature to the PDF */
env.pdfsign(doc_to_sign, 0, "<invisible>", null, null, null, null,
            "App Tester", alias, pin, otp, null, null, null, 0, 0, 0, 0, 0, 0,
            signed_doc);
```

Changing the values passed to the `pdfsign` method, a visible digital signature can be applied to the PDF in a specific position, as in the following code (pay attention: if the credential requires OTP, you have to generate a new value if you ran the previous signature sample):

```
/* applying a visible signature to the PDF */
env.pdfsign(doc_to_sign, 0, "<new>", "<Text>Signed by %cn%\non %date%</Text>",
            null, null, null, "App Tester", alias, pin, otp, null, null, null,
            -1, 0, 100, 300, 200, 60, signed_doc);
```

The full source code sample can be found in 7.2, refer to javadoc for parameter details of the `pdfsign` method.

3.3. XAdES signature sample

The following code allows to apply an *enveloped* XAdES signature to an XML file using the method `xmlsign` provided by the `XMLEnvelope` class (pay attention: there are several versions of this method with the same behavior, here has been used the one accepting Stream variables):

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
XMLEnvelope xmlEnv = new XMLEnvelope(pkbox);
String alias = "signer_test";
String pin = "12345678";
String otp = "454876"; // if required by the credential otherwise set to null
FileOutputStream fop = null;
File doc_to_sign = new File("/temp/doc_to_sign.xml");
FileInputStream fip = new FileInputStream(doc_to_sign);
fop = new FileOutputStream("/temp/signed_doc_enveloped.xml");

/* applying an XML enveloped signature */
xmlEnv.xmlsign(fip, (int)doc_to_sign.length(), null, null, null, alias, pin,
               otp, XMLEnvelope.envelopedMode, null, fop);

fip.close();
fop.close();
```

Changing the values passed to the `xmlsign` method as in the following code, produces an *enveloping* XAdES signature (pay attention: if the credential requires OTP, you have to generate a new value if you ran the previous signature sample):

```
XMLReferenceData[] refsData = new XMLReferenceData[1];
XMLReference[] references = new XMLReference[1];
refsData[0] = new XMLReferenceData("/temp/doc_to_sign.xml", null,
                                   "application/xml", null);
references[0] = new XMLReference(refsData[0], true, null, null, null, null,
                                 false, true);
fop = new FileOutputStream("/temp/signed_doc_enveloping.xml");

/* applying an XML enveloping signature */
xmlEnv.xmlsign(null, 0, null, references, null, alias, pin, otp,
               XMLEnvelope.envelopingMode, null, fop);

fop.close();
```

The following code instead produces a *detached* XAdES signature (pay attention: if the credential requires OTP, you have to generate a new value if you ran the previous signature sample):

```
refsData[0] = new XMLReferenceData("/temp/doc_to_sign.xml", "doc_to_sign.xml",
                                   "application/xml", null);
references[0] = new XMLReference(refsData[0], false, null, null,
                                 "doc_to_sign.xml", null, false, true);
fop = new FileOutputStream("/temp/signed_doc_detached.xml");

/* applying an XML detached signature */
xmlEnv.xmlsign(null, 0, null, references, null, alias, pin, otp,
               XMLEnvelope.detachedMode, null, fop);

fop.close();
```

The full source code sample can be found in 7.3, refer to javadoc for parameter details of the `xmlsign` method.

3.4. ASiC signature sample

The following code allows to apply an ASiC-S signature to a file using the method `asicsign` provided by the `Envelope` class (pay attention: there are several versions of this method with the same behavior, here has been used the one accepting Stream variables):

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Envelope env = new Envelope(pkbox);
String alias = "signer_test";
String pin = "12345678";
String otp = "774675"; // if required by the credential otherwise set to null
FileOutputStream fop = null;
AsicDataObject[] dataObjects = new AsicDataObject[1];
dataObjects[0] = new AsicDataObject(new File("/temp/document_to_sign.pdf"),
                          null, "document_to_sign.pdf", "application/pdf");
fop = new FileOutputStream("/Users/ribba/temp/signed_doc.asics");

/* applying an ASiC-S signature to a file */
env.asicsign(dataObjects, null, "App Tester", alias, pin, otp, null,
             Envelope.ASiC_S_Format, null, fop);

fop.close();
```

Changing the values passed to the `asicsign` method, an ASiC-E signature can be applied to one or more files, as in the following code (pay attention: if the credential requires OTP, you have to generate a new value if you ran the previous signature sample):

```
dataObjects = new AsicDataObject[2];
dataObjects[0] = new AsicDataObject(new File("/temp/document_to_sign.pdf"),
                          null, "document_to_sign.pdf", "application/pdf");
```

```
dataObjects[1] = new AsicDataObject(new File("/temp/signed_doc.asics"),
                          null, "signed_doc.asics", "application/zip");
fop = new FileOutputStream("/Users/ribba/temp/signed_doc.asice");

/* applying an ASiC-E signature to a file */
env.asicsign(dataObjects, null, "App Tester", alias, pin, otp, null,
             Envelope.ASiC_E_Format, null, fop);
fop.close();
```

The full source code sample can be found in 7.4, refer to javadoc for parameter details of the `asicsign` method.

3.5. Raw signature sample

The following code allows to apply a raw sign to a hash using the method `rawsign` provided by the `Utils` class (pay attention: there are several versions of this method with the same behavior, here has been used the one accepting File variables):

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Utils utils = new Utils(pkbox);
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] data_to_sign = md.digest("test".getBytes());
byte[] signed_data = null;
String alias = "signer_test";
String pin = "12345678";
String otp = "746694"; // if required by the credential otherwise set to null

/* applying a raw sign */
signed_data = utils.rawsign(data_to_sign, Utils.sha256, "App Tester",
                            null, alias, pin, otp, null, Utils.rsaPkcs1_15,
```

```
                                Utils.rawBinaryEncoding);
```

The full source code sample can be found in 7.5, refer to javadoc for parameter details of the `rawsign` method.

4. OTP pushing

The OTP used by a credential to apply a signature (if required) may be generated through an offline device (e.g. software token or hardware token) or using an online service like SMS or email. In the latter case the OTP generation is subordinated to a trigger done by the signing application to PkBox: as response, PkBox generates the OTP and sends it to the user by the SMS or email (pay attention: no OTP is replied to the caller, only the end user is the owner of the OTP).

The OTP pushing is done through the method `pushOTP` provided by the `Envelope` class. In the following is shown a sample to use it.

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Envelope env = new Envelope(pkbox);
String alias = "signer_test";

/* pushing an OTP to the user */
env.pushOTP(alias, null, null, null, null);
```

The default message sent to the user is configured on server-side. This message can be customized on client-side in the following way:

```
String alias = "signer_test";
String message = "Hi, here you are your verification code: $otp";

/* pushing an OTP to the user */
env.pushOTP(alias, null, message, null, null);
```

where the placeholder `$otp` is replaced with the real OTP.

If the OTP is sent via SMS, the default "sender" of the message is also configured on server-side. The sender may be customized on client-side (if the SMS service provider allows it) using the following JSON structure as template value:

```
{
  "sender": "Intesi Group",
  "body": "Hi, here you are your verification code: $otp"
}
```

Refer to javadoc for parameter details of the `pushOTP` method.

5. Transaction Management

In some business domains a lot of documents need to be signed by the same user, or it is necessary to apply many signatures on the same document (e.g. contract signature in PAdES format). In those cases, the users often request to be able to sign all these documents inserting PIN and OTP in one go.

The PkBox SDK makes available a specific method that, starting from PIN and OTP inserted, creates a transaction that allows to apply at most the specified number of signatures declared on transaction creation phase.

The transaction management is done through the following methods provided by the `Envelope` class:

- `startTransaction`
  This method creates a transaction for a specified number of signatures. A transaction has a validity of 5 minutes.
- `continueTransaction`
  If a signing process needs more time than the validity of a transaction, this method allows to extend the validity for other 5 minutes. The output of this method has to be used in the following calls as the new authentication data.
- `endTransaction`
  If for any reason the transaction needs to be invalidated, this method ends the transaction validity. Anyway when the allowed number of signatures is reached, the transaction is automatically closed.

After the creation, the authorized number of signatures cannot be changed: you may only extend or end the validity time.

NOTE 1: if you have to sign N documents and each document has M signature, the number of signatures to specify to the startTransaction will be N x M.

NOTE 2: as described in 2.1.1, SecurePIN is mandatory in order to use signatures with transactions.

5.1. startTransaction sample

The following code allows to create a transaction to sign two document each one with one signature.

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
pkbox.setSecurePINCert("/temp/SecurePin.cer");
Envelope env = new Envelope(pkbox);
String alias = "signer_test";
String pin = "12345678";
String otp = "685234";

/* start a transaction for a specific number of signatures */
String trans = env.startTransaction("App Tester", alias, pin, otp, null, 3);

/* signing the first document */
File doc_to_sign1 = new File("/temp/document_to_sign1.pdf");
File signed_doc1 = new File("/temp/signed_doc1.pdf");
env.pdfsign(doc_to_sign1, 0, "<invisible>", null, null, null, null,
            "App Tester", alias, pin, trans, null, null, 0, 0, 0, 0, 0, 0,
            signed_doc1);

/* signing the second document */
File doc_to_sign2 = new File("/temp/document_to_sign2.pdf");
File signed_doc2 = new File("/temp/signed_doc2.pdf");
env.pdfsign(doc_to_sign2, 0, "<new>", "<Text>Signed by %cn%\non %date%</Text>",
            null, null, null, "App Tester", alias, pin, trans, null, null, -1,
            0, 100, 300, 200, 60, signed_doc2);
```

```
/* continue a transaction for other 5 minutes */
String newTrans = env.continueTransaction("App Tester", alias, pin, trans);

/* signing the third document */
File doc_to_sign3 = new File("/temp/document_to_sign3.pdf");
File signed_doc3 = new File("/temp/signed_doc3.p7m");
env.sign(doc_to_sign3, "App Tester", alias, pin, newTrans,
         Envelope.implicitMode, Envelope.derEncoding, null, signed_doc3);
```

6. Multiple Signatures

Although you may ask to the user the PIN and OTP only one time and use the transaction mechanism to sign several documents, every sign operation requires sending the file to the server to apply the signature. If you are applying more than one signature in the same document, sometimes this approach could not be efficient.

In this case the SDK makes available a set of methods that allows to apply more than one signature in the same document or digest with one call. In the following sections are described these multiple signature methods.

6.1. CAdES signatures

For multiple CAdES signatures the `Envelope` class provides the method `multisigndigest` that allows to create an array of `pkcs7` detached envelope starting from an array of `DigestInfo` datas.

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Envelope env = new Envelope(pkbox);
Utils utils = new Utils(pkbox);
String alias = "signer_test";
String pin = "12345678";
String otp = "275855"; // if required by the credential otherwise set to null
byte[][] digestInfos = {
     utils.digest("test1".getBytes(), Utils.sha256, Utils.derEncoding),
     utils.digest("test2".getBytes(), Utils.sha256, Utils.derEncoding) };

/* create a multiple pkcs#7 detached envelope */
byte[][] pkcs7s = env.multisigndigest(digestInfos, "App Tester", alias, pin,
                                      otp, null, Envelope.derEncoding, null);
```

The full source code sample can be found in 7.1, refer to javadoc for parameter details of the `multisigndigest` method.

6.2. PAdES signatures

For multiple PAdES signatures the `Envelope` class provides the method `multipdfsignEx` (pay attention: there are several versions of this method with the same behavior, here has been used the one accepting File variables). The signature parameters are specified using the array variables: the same index of each array identifies a parameter for the same signature.

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Envelope env = new Envelope(pkbox);
File doc_to_sign = new File("/temp/document_to_sign.pdf");
File signed_doc = new File("/temp/signed_doc.pdf");
String alias = "signer_test";
String pin = "12345678";
String otp = "685234"; // if required by the credential otherwise set to null

int[] accessPermissions = {0, 0, 0};
String[] fieldNames =     {"<new>", "<new>", "<new>"};
String[] sigLayout =      {"<Text>Signed by %cn%\non %date%</Text>",
                           "<Text>%dn%</Text>", ""};
int[] page =              {0, -1, -1};
int[] position =          {0, 0, 0};
float[] x =               {100, 150, 200};
float[] y =               {500, 300, 100};
float[] cx =              {200, 200, 200};
float[] cy =              {60, 60, 60};
```

```
/* applying three visible signatures in one time to the PDF */
env.multipdfsignEx(doc_to_sign, accessPermissions, fieldNames, sigLayout, null,
                   null, null, "App Tester", alias, pin, otp, null, null,
                   null, page, position, x, y, cx, cy, signed_doc);
```

The full source code sample can be found in 7.2, refer to javadoc for parameter details of the `multipdfsignEx` method.

6.3. Raw signatures

For multiple raw signatures the `Utils` class provides the method `multirawsign` that allows to sign an array of hashes.

```
PKBox pkbox = new PKBox();
pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
                null, null, null);
Utils utils = new Utils(pkbox);
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[][] datas_to_sign = { md.digest("test1".getBytes()),
                           md.digest("test2".getBytes()) };
byte[][] signed_datas = null;
String alias = "signer_test";
String pin = "12345678";
String otp = "922922"; // if required by the credential otherwise set to null

/* applying a multi raw sign */
signed_datas = utils.multirawsign(datas_to_sign, Utils.sha256,
                                  "App Tester", null, alias, pin, otp, null,
                                  Utils.rsaPkcs1_15, Utils.rawBinaryEncoding);
```

The full source code sample can be found in 7.5, refer to javadoc for parameter details of the `multirawsign` method.

7. Source Code Samples

In the following sections you may find the full source code of the sample previously discussed.

7.1. CAdES

```
import java.io.File;
import it.pkbox.client.Base64;
import it.pkbox.client.Envelope;
import it.pkbox.client.PKBox;
import it.pkbox.client.PKBoxException;
import it.pkbox.client.PkBoxLog;
import it.pkbox.client.Utils;

/**
 * This sample class performs an attached CAdES signature,
 * a detached CAdES signature, a pkcs7 detached envelope and a multiple
 * pkcs7 detached envelope.
 */
public class CAdES
{
    public static void main(String[] args)
    {
        try
        {
            PkBoxLog pklog = new PkBoxLog()
            {
                public void warn(String msg) {
                    logger.warn(msg);
                }

                public void info(String msg) {
                    logger.info(msg);
                }

                public void error(String msg, Throwable e) {
                    logger.error(msg, e);
```

```
                }

                public void error(String msg) {
                    logger.error(msg);
                }

                public void debug(String msg) {
                    logger.debug(msg);
                }
            };

            PKBox pkbox = new PKBox(pklog);
            pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
null, null, null);
            pkbox.setSecurePINCert("/temp/SecurePin.cer");

            Envelope env = new Envelope(pkbox);
            Utils utils = new Utils(pkbox);

            File doc_to_sign = new File("/temp/document_to_sign.pdf");
            File signed_doc = new File("/temp/signed_doc.p7m");

            String alias = "signer_test";
            String pin = "12345678";
            String otp = "275855"; // if required by the credential otherwise set to null

            if (otp != null && otp.length() > 0)
            {
                /* start a transaction for a specific number of signatures */
                int numSignatures = 5;
                otp = env.startTransaction("App Tester", alias, pin, otp, null,
numSignatures);
            }

            /* applying an attached signature to a file */
            env.sign(doc_to_sign, "App Tester", alias, pin, otp, null,
Envelope.implicitMode, Envelope.derEncoding, null, signed_doc);
            System.out.println("Attached signature successfully applied!");

            signed_doc = new File("/temp/signed_doc.p7s");
```

```
            /* applying a detached signature to a file */
            env.sign(doc_to_sign, "App Tester", alias, pin, otp, null,
Envelope.explicitMode, Envelope.derEncoding, null, signed_doc);
            System.out.println("Detached signature successfully applied!");

            byte[] digestInfo = utils.digest("test".getBytes(), Utils.sha256,
Utils.derEncoding);

            /* create a pkcs#7 detached envelope */
            byte[] pkcs7 = env.signdigest(digestInfo, "App Tester", alias, pin, otp,
null, Envelope.derEncoding, null);
            System.out.println("pkcs7 successfully create: " + new
String(Base64.encode(pkcs7)));

            byte[][] digestInfos = { utils.digest("test1".getBytes(), Utils.sha256,
Utils.derEncoding),
                                     utils.digest("test2".getBytes(), Utils.sha256,
Utils.derEncoding) };

            /* create a multiple pkcs#7 detached envelope */
            byte[][] pkcs7s = env.multisigndigest(digestInfos, "App Tester", alias, pin,
otp, null, Envelope.derEncoding, null);
            System.out.println("pkcs7 successfully create:");
            System.out.println("pkcs7 1: " + new String(Base64.encode(pkcs7s[0])));
            System.out.println("pkcs7 2: " + new String(Base64.encode(pkcs7s[1])));
        }
        catch (PKBoxException e)
        {
            System.out.println("error code: " + e.GetErrorCode() + ", error msg: " +
e.getMessage());
            System.out.println();
            e.printStackTrace();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
```

```
}
```

7.2. PAdES

```
import java.io.File;
import it.pkbox.client.Envelope;
import it.pkbox.client.PKBox;
import it.pkbox.client.PKBoxException;
import it.pkbox.client.PkBoxLog;

/**
 * This sample class performs an invisible, a visible and a multiple
 * visible PAdES signature.
 */
public class PAdES
{
    public static void main(String[] args)
    {
        try
        {
            PkBoxLog pklog = new PkBoxLog()
            {
                public void warn(String msg) {
                    logger.warn(msg);
                }

                public void info(String msg) {
                    logger.info(msg);
                }

                public void error(String msg, Throwable e) {
                    logger.error(msg, e);
                }

                public void error(String msg) {
                    logger.error(msg);
                }
```

```
                public void debug(String msg) {
                    logger.debug(msg);
                }
            };

            PKBox pkbox = new PKBox(pklog);
            pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
null, null, null);
            pkbox.setSecurePINCert("/temp/SecurePin-int.cer");

            Envelope env = new Envelope(pkbox);

            File doc_to_sign = new File("/temp/document_to_sign.pdf");
            File signed_doc = new File("/temp/signed_doc.pdf");

            String alias = "signer_test";
            String pin = "12345678";
            String otp = "685234"; // if required by the credential otherwise set to null

            if (otp != null && otp.length() > 0)
            {
                /* start a transaction for a specific number of signatures */
                int numSignatures = 5;
                otp = env.startTransaction("App Tester", alias, pin, otp, null,
numSignatures);
            }

            /* applying an invisible signature to the PDF */
            env.pdfsign(doc_to_sign, 0, "<invisible>", null, null, null, null, "App
Tester", alias, pin, otp, null, null, null, 0, 0, 0, 0, 0, 0, signed_doc);
            System.out.println("Invisible signature successfully applied!");

            signed_doc = new File("/temp/signed_doc2.pdf");

            /* applying a visible signature to the PDF */
            env.pdfsign(doc_to_sign, 0, "<new>", "<Text>Signed by %cn%\non
%date%</Text>", null, null, null, "App Tester", alias, pin, otp, null, null, null, -1, 0,
100, 300, 200, 60, signed_doc);
            System.out.println("Visible signature successfully applied!");
```

```
            int[] accessPermissions = {0, 0, 0};
            String[] fieldNames =     {"<new>", "<new>", "<new>"};
            String[] sigLayout =      {"<Text>Signed by %cn%\non %date%</Text>",
"<Text>%dn%</Text>", ""};
            int[] page =              {0, -1, -1};
            int[] position =          {0, 0, 0};
            float[] x =               {100, 150, 200};
            float[] y =               {500, 300, 100};
            float[] cx =              {200, 200, 200};
            float[] cy =              {60, 60, 60};

            signed_doc = new File("/temp/signed_doc3.pdf");

            /* applying three visible signatures in one time to the PDF */
            env.multipdfsignEx(doc_to_sign, accessPermissions, fieldNames, sigLayout,
null, null, null, "App Tester", alias, pin, otp, null, null, null, page, position, x, y,
cx, cy, signed_doc);
            System.out.println("Multiple visible signature successfully applied!");
        }
        catch (PKBoxException e)
        {
            System.out.println("error code: " + e.GetErrorCode() + ", error msg: " +
e.getMessage());
            System.out.println();
            e.printStackTrace();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

7.3. XAdES

```
import java.io.File;
```

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import it.pkbox.client.Envelope;
import it.pkbox.client.PKBox;
import it.pkbox.client.PKBoxException;
import it.pkbox.client.PkBoxLog;
import it.pkbox.client.XMLEnvelope;
import it.pkbox.client.XMLReference;
import it.pkbox.client.XMLReferenceData;

/**
 * This sample class performs an XML enveloped, enveloping
 * and detached XAdES signature.
 */
public class XAdES
{
    public static void main(String[] args)
    {
        try
        {
            PkBoxLog pklog = new PkBoxLog()
            {
                public void warn(String msg) {
                    logger.warn(msg);
                }

                public void info(String msg) {
                    logger.info(msg);
                }

                public void error(String msg, Throwable e) {
                    logger.error(msg, e);
                }

                public void error(String msg) {
                    logger.error(msg);
                }

                public void debug(String msg) {
                    logger.debug(msg);
```

```
                }
            };

            PKBox pkbox = new PKBox(pklog);
            pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
null, null, null);
            pkbox.setSecurePINCert("/temp/SecurePin.cer");

            Envelope env = new Envelope(pkbox);
            XMLEnvelope xmlEnv = new XMLEnvelope(pkbox);

            String alias = "signer_test";
            String pin = "12345678";
            String otp = "454876"; // if required by the credential otherwise set to null

            if (otp != null && otp.length() > 0)
            {
                /* start a transaction for a specific number of signatures */
                int numSignatures = 3;
                otp = env.startTransaction("App Tester", alias, pin, otp, null,
numSignatures);
            }

            FileOutputStream fop = null;
            File doc_to_sign = new File("/temp/doc_to_sign.xml");
            FileInputStream fip = new FileInputStream(doc_to_sign);

            fop = new FileOutputStream("/temp/signed_doc_enveloped.xml");

            /* applying an XML enveloped signature */
            xmlEnv.xmlsign(fip, (int)doc_to_sign.length(), null, null, null, alias, pin,
otp, XMLEnvelope.envelopedMode, null, fop);
            fip.close();
            fop.close();
            System.out.println("XML enveloped signature successfully applied!");

            XMLReferenceData[] refsData = new XMLReferenceData[1];
            XMLReference[] references = new XMLReference[1];
```

```
            refsData[0] = new XMLReferenceData("/temp/doc_to_sign.xml", null,
"application/xml", null);
            references[0] = new XMLReference(refsData[0], true, null, null, null, null,
false, true);

            fop = new FileOutputStream("/temp/signed_doc_enveloping.xml");

            /* applying an XML enveloping signature */
            xmlEnv.xmlsign(null, 0, null, references, null, alias, pin, otp,
XMLEnvelope.envelopingMode, null, fop);
            fop.close();
            System.out.println("XML enveloping signature successfully applied!");

            refsData[0] = new XMLReferenceData("/temp/doc_to_sign.xml",
"doc_to_sign.xml", "application/xml", null);
            references[0] = new XMLReference(refsData[0], false, null, null,
"doc_to_sign.xml", null, false, true);

            fop = new FileOutputStream("/temp/signed_doc_detached.xml");

            /* applying an XML detached signature */
            xmlEnv.xmlsign(null, 0, null, references, null, alias, pin, otp,
XMLEnvelope.detachedMode, null, fop);
            fop.close();
            System.out.println("XML detached signature successfully applied!");
        }
        catch (PKBoxException e)
        {
            System.out.println("error code: " + e.GetErrorCode() + ", error msg: " +
e.getMessage());
            System.out.println();
            e.printStackTrace();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

7.4. ASiC

```
import java.io.File;
import java.io.FileOutputStream;
import it.pkbox.client.AsicDataObject;
import it.pkbox.client.Envelope;
import it.pkbox.client.PKBox;
import it.pkbox.client.PKBoxException;
import it.pkbox.client.PkBoxLog;

/**
 * This sample class performs an ASiC-S signature
 * and an ASiC-E signature.
 */
public class ASiC
{
    public static void main(String[] args)
    {
        try
        {
            PkBoxLog pklog = new PkBoxLog()
            {
                public void warn(String msg) {
                    logger.warn(msg);
                }

                public void info(String msg) {
                    logger.info(msg);
                }

                public void error(String msg, Throwable e) {
                    logger.error(msg, e);
                }

                public void error(String msg) {
                    logger.error(msg);
                }

                public void debug(String msg) {
```

```
                    logger.debug(msg);
                }
            };

            PKBox pkbox = new PKBox(pklog);
            pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
null, null, null);
            pkbox.setSecurePINCert("/temp/SecurePin.cer");

            Envelope env = new Envelope(pkbox);

            String alias = "signer_test";
            String pin = "12345678";
            String otp = "774675"; // if required by the credential otherwise set to null

            if (otp != null && otp.length() > 0)
            {
                /* start a transaction for a specific number of signatures */
                int numSignatures = 2;
                otp = env.startTransaction("App Tester", alias, pin, otp, null,
numSignatures);
            }

            FileOutputStream fop = null;
            AsicDataObject[] dataObjects = new AsicDataObject[1];
            dataObjects[0] = new AsicDataObject(new File("/temp/document_to_sign.pdf"),
null, "document_to_sign.pdf", "application/pdf");

            fop = new FileOutputStream("/temp/signed_doc.asics");

            /* applying an ASiC-S signature to a file */
            env.asicsign(dataObjects, null, "App Tester", alias, pin, otp, null,
Envelope.ASiC_S_Format, null, fop);
            fop.close();
            System.out.println("ASiC-S signature successfully applied!");

            dataObjects = new AsicDataObject[2];
            dataObjects[0] = new AsicDataObject(new File("/temp/document_to_sign.pdf"),
null, "document_to_sign.pdf", "application/pdf");
```

```
            dataObjects[1] = new AsicDataObject(new File("/temp/signed_doc.asics"), null,
"signed_doc.asics", "application/zip");

            fop = new FileOutputStream("/temp/signed_doc.asice");

            /* applying an ASiC-E signature to a file */
            env.asicsign(dataObjects, null, "App Tester", alias, pin, otp, null,
Envelope.ASiC_E_Format, null, fop);
            fop.close();
            System.out.println("ASiC-E signature successfully applied!");
        }
        catch (PKBoxException e)
        {
            System.out.println("error code: " + e.GetErrorCode() + ", error msg: " +
e.getMessage());
            System.out.println();
            e.printStackTrace();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

7.5. Raw signature

```
import java.security.MessageDigest;
import it.pkbox.client.Base64;
import it.pkbox.client.Envelope;
import it.pkbox.client.PKBox;
import it.pkbox.client.PKBoxException;
import it.pkbox.client.PkBoxLog;
import it.pkbox.client.Utils;

/**
 * This sample class performs a raw signature and a multiple raw signature.
```

```
 */
public class Rawsign
{
    public static void main(String[] args)
    {
        try
        {
            PkBoxLog pklog = new PkBoxLog()
            {
                public void warn(String msg) {
                    logger.warn(msg);
                }

                public void info(String msg) {
                    logger.info(msg);
                }

                public void error(String msg, Throwable e) {
                    logger.error(msg, e);
                }

                public void error(String msg) {
                    logger.error(msg);
                }

                public void debug(String msg) {
                    logger.debug(msg);
                }
            };

            PKBox pkbox = new PKBox(pklog);
            pkbox.addServer("https://192.168.0.39:8443/pkserver/servlet/defaulthandler",
null, null, null);
            pkbox.setSecurePINCert("/temp/SecurePin.cer");

            Utils utils = new Utils(pkbox);
            Envelope env = new Envelope(pkbox);

            MessageDigest md = MessageDigest.getInstance("SHA-256");
            byte[] data_to_sign = md.digest("test".getBytes());
```

```
            byte[] signed_data = null;

            String alias = "signer_test";
            String pin = "12345678";
            String otp = "922922"; // if required by the credential otherwise set to null

            if (otp != null && otp.length() > 0)
            {
                /* start a transaction for a specific number of signatures */
                int numSignatures = 3;
                otp = env.startTransaction("App Tester", alias, pin, otp, null,
numSignatures);
            }

            /* applying a raw sign */
            signed_data = utils.rawsign(data_to_sign, Utils.sha256, "App Tester", null,
alias, pin, otp, null, Utils.rsaPkcs1_15, Utils.rawBinaryEncoding);
            System.out.println("Raw sign successfully applied: " + new
String(Base64.encode(signed_data)));

            byte[][] datas_to_sign = { md.digest("test1".getBytes()),
md.digest("test2".getBytes()) };
            byte[][] signed_datas = null;
            /* applying a multi raw sign */
            signed_datas = utils.multirawsign(datas_to_sign, Utils.sha256, "App Tester",
null, alias, pin, otp, null, Utils.rsaPkcs1_15, Utils.rawBinaryEncoding);

            System.out.println("Multi raw sign successfully applied:");
            System.out.println("rawsign 1: " + new
String(Base64.encode(signed_datas[0])));
            System.out.println("rawsign 2: " + new
String(Base64.encode(signed_datas[1])));
        }
        catch (PKBoxException e)
        {
            System.out.println("error code: " + e.GetErrorCode() + ", error msg: " +
e.getMessage());
            System.out.println();
            e.printStackTrace();
        }
```

```
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

8. Configuration Sample

In the following section is shown a sample of configuration file `pkbox.properties`.

```
# To be copied into classpath

#Proxy parameters

#Proxy Host URL (leave blank when if no proxy is used)
ProxyHost=

#Proxy Host Port (default value = 80)
ProxyPort=

#PkBox connection timeout  (default value = 60 seconds)
Timeout=60

#SSL Client Authentication parameters

#Flag which defines if SSL client authentication has to be managed to connect to PkBox
servers (default value is false). The next parameters are used only if the SSL client
authentication is enabled
SSLClientAuth=

#Flag which specify if SSL client certificate path validation has to be enabled (default
value is true). In case of true value, the issuer certificate (CA certificate) of SSL
client certificate has to be installed in the JDK cacerts store (usually installed in
PKBox/<jdk dir>/jre/lib/security directory)
ClientCertValidation=

#Defines the SSL key algorithm (default value is SunX509)
KeyManagerFactoryAlgorithm=

#Defines the SSL key store file path. The parameter is mandatory
KeyStoreFile=

#defines the SSL key store type. The default value is pkcs12 if the KeyStoreFile
parameters ends with .p12 or .pfx, otherwise is JKS
```

```
KeyStoreType=

#Defines the SSL key store provider. The default value is null, the first installed JCE
provider is used
KeyStoreProvider=

#Defines the SSL key store password
KeyStorePassword=

#defines the SSL key password. If no value is specified, it is used the same value
specified with KeyStorePassword parameter
KeyStoreKeyPassword=

#Defines the SSL protocol (default value is TLS)
SSLProtocol=

#The next parameters are repeated for each PkBox server (from 1 to N), the mandatory
parameter is PkBoxUrlX (where X is the server index)

#PkBox URL
PkBoxUrl1=https://192.168.0.39:8443/pkserver/servlet/defaulthandler

#PkBox Realm for HTTP Basic Authentication (default value = pkserver, leave blank if HTTP
Basic Authentication is not required)
PkBoxRealm1=

#PkBox User for HTTP Basic Authentication (leave blank if HTTP Basic Authentication is
not required)
PkBoxUser1=

#PkBox Password for HTTP Basic Authentication (leave blank if HTTP Basic Authentication
is not required)
PkBoxPassword1=
```