E-Sign using I-Text

(Technical Codes and implementation)

*(P.S : This doc concentrates over mainly till Page : **~81/150** from the iText Documentation, before deployment on Jio locker whole doc needs to be completed. (Left part mainly concentrates on server side implementation)*

Q. What is E-Sign
An Advanced Electronic Signature is an electronic signature that is:

- Uniquely linked to the signatory,
- Capable of identifying the signatory,
- Created using means that the signatory can maintain under his sole control, and
- Linked to the data to which it relates so that any subsequent change of the data is detectable

Q. Objective behind E-Sign

- The integrity of the document— assurance that the document hasn't been changed somewhere in the workflow.
- The authenticity of the document— assurance that the author of the document is who you think it is (and not somebody else)
- Non-repudiation— assurance that the author can't deny his or her authorship.

Q.What are Approaches for E-Sign ?
CMS Advanced Electronic Signatures (CAdES)
XML Advanced Electronic Signatures (XAdES) ***(Government approach)***
PDF Advanced Electronic Signatures (PAdES) ***(our approach using iText)***

Q. Why iText ?

- Because tutorials are easily accesible
- Using basic java (bluej) so easier to understand than xml commands
- Most of the things are abstract and we just have to import pre defined libraries

Q. Basic syntaxt and view:

Every pdf starts with the %PDF and ends with %EOF and in between we insert a digital signature, which stores a has of all the data except digital signature . When we need to validate the pdf

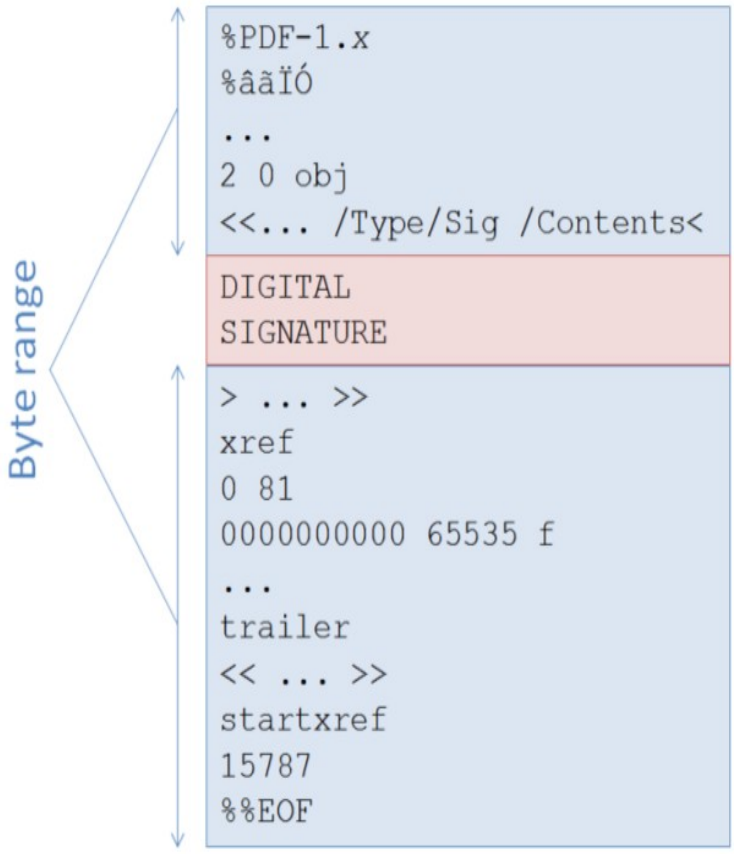

Figure 2.1: A signed PDF

just calc the hash of the pdf and match it with the hash stored in the pdf.

Q. How to achieve the above objectives ?
A. By using the concepts of hashing and encryption, we can assure that a document hasn't been modified.
Using the public/private key mechanism, we can authenticate the author and he can't deny the authorship. The authenticity of the document and author, using certifying authority

Q. Whats the need for certifying authorities ?
Imagine two real-world people who have never met: **A** and **B**. **A** needs to send a signed PDF document to **B**. For **B** to be able to validate the signature in the document, he needs **A's** public certificate. That certificate is embedded in the signed PDF document, so he could add it to his list of Trusted Identities (2.2.2), but how does **B** know he can trust that certificate? Anyone can make a self-signed certificate using **A's** name.
This problem can be solved by involving a third person who knows both **A** and **B**.

Q. How certifying authrity ensures people are they who say they are ?
A. certifying authrity is a tusted authority, 'trusted entity'; CA trusts a Intermediate party (Jio Locker in our case) which in turn validates identity of signer, and reciever. If somehow CA marks signer's public key to confirm its authenticity, reciever knows that signer is who he/she claims to be. In the context of signatures, CA does this by signing signer's certificate. Reciever can decrypt singer's signed certificate using CA's certificate, and as reciever trusts CA's public certificate, he can now also trust signer's certificate.

CA to uniquely recognise the signer : OTP, email varifiaction and account login
Govt : aadhar number

Q. How this CA works in eSign
Person is validated to Jio Cloud Services and at the backend. Earlier Jio cloud services have also registered their public key with the CA. Since CA trusts Jio cloud services they would bind the public key of Jio cloud service to their proof of indentity
Now if Jio trust the signer after few verification process and certifies the signature of the signer, using its own public key.

So Singer sign the pdf using his private key, K_S^-
Sends its Public key using CA for validity of proof of identity attached (CA certificate which binds Singer's identity to signer public key.
CA's certificate containing Jio's public key signed by CA(Using CA private key)
PUBLIC KEY of Signer SINGED BY PRIVATE KEY OF CA = $K_{CA}^- K_S^+$
EVERYONE HAS CA PUBLIC KEY K_{CA}^+
Now whoever recieves this signature all he need to see K_{CA}^{+-} matches, and he will get signer public key. It's confirmed that signer is who he say's he is.

The private key used to sign the doc at the first place would be destroyed within 30mins and validation certificate generation period would last only for 30mins to validate the signature.

After the expiray of this period it can't be validated that signer is who he says he is. But once its certified that signer can't deny that he didn't signed signature and the signature will belong to him and only him. Its the CA who is trusted by all i.e adobe truts that this signature is by the person who claims he is the one who he says he is.

Q. CA is not a free agent to recognise who all applies for the the signature,
A. Intermediate trusted body needs to do it for CA, i.e Reliance Jio intermediate CA

Q. Who are all are called CA's when validating a signature in adobe
A. https://helpx.adobe.com/acrobat/kb/approved-trust-list1.html

Q. How to stop from altering the hash stored in the signature
A. The hash value is encrypted with the signer's private key and a hex-encoded PKCS#7

Q. What this digital signature contains ?

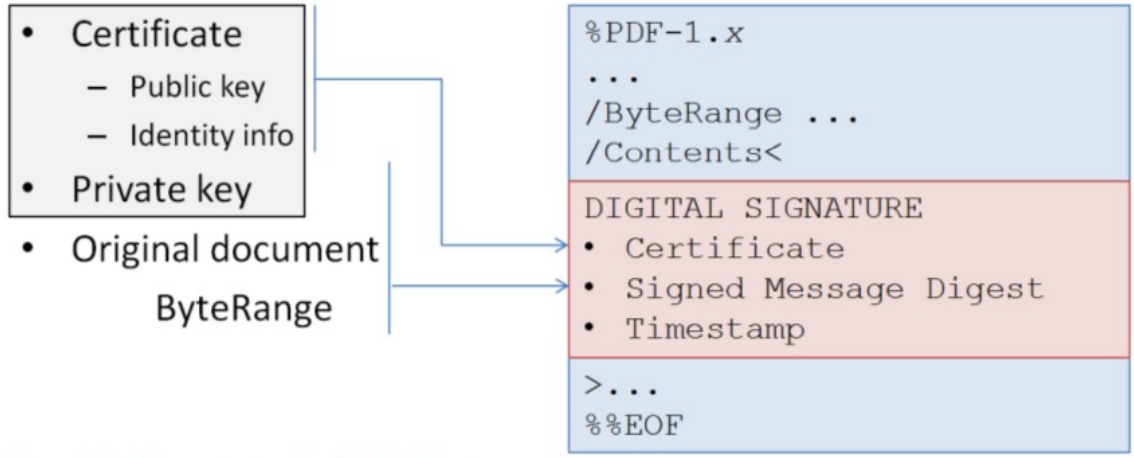

Figure 2.2: the contents of a digital signature

The signature also contains a digest of the original document that was signed using the private key. Additionally, the signature should contain a timestamp.

The encryption method ("RSA", "DSA" or "ECDSA")

Q.Type of signature that are currently fessible for us ?
A

1. One or more approval signatures : green check mark when a valid approval signature is present.
2. At most one certification signature

- NOT_CERTIFIED : normal signature, signed by :
- ***CERTIFIED_NO_CHANGES_ALLOWED*** : creates a certification signature aka an author signature. After the signature is applied, no changes to the document will be allowed.
- ***CERTIFIED_FORM_FILLING***— creates a certification signature for the author of

the document. Other people can still fill out form fields or add approval signatures without invalidating the signature.

- CERTIFIED_FORM_FILLING_AND_ANNOTATIONS

Ref : Pg50 from the referenced iText document

Q. How can we be sure that the public key we're using is signer, and not a public key from somebody pretending to be signer ?
What happens if signer private key is compromised?

CA is the solution
However, when private key is compromised, somebody stole my certificate, and is planning to sign documents pretending to be me. If that happens, I should immediately notify the Certificate Authority, and ask the CA to add my certificate to the Certificate Revocation List (CRL).

Q. What are legal requirements

In some countries, not only non-revocation of the certificate, but also existence has to be proven when verifying qualified signatures. is to demand that every signed document has a timestamp obtained from a Time Stamping Authority (TSA).

Adobe Reader will inspect the certificate and see that it has expired. If there's no embedded CRL, no embedded OCSP response, and no timestamp, Adobe Reader won't show a green check mark anymore, but it will give a warning saying: I can't trust this signature anymore because the expiration date is in the past

Apart from the fact that signer can no longer sign documents after leaving the company on, all signatures he/she created before that date can no longer be trusted if no CRL, OCSP response or timestamp was added. There's no way for the consumer of the document to verify if the document is valid. The certificate was revoked, and there's no certainty about the date the document was signed. After all, signer could have changed the clock on her computer.

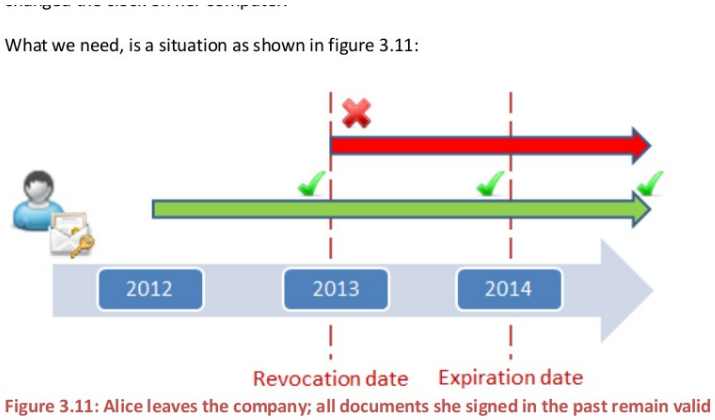

Figure 3.11: Alice leaves the company; all documents she signed in the past remain valid

Document that was signed by signer earlier will contain revocation information dating from the time of signature saying that his/her certificate wasn't revoked at that time; the timestamp will assure that the document was certainly signed in that point which it claims to be from. This signature will survive the revocation and expiration date.

Q. How to trust timestamp since the signer might have changed the computer's timing at the time of signing ?

We need to involve another third party: a Time Stamping Authority (TSA). A TSA provides an online service, signing signature bytes and concatenating a timestamp to it. This is done on a timestamp server that is contacted during the signing process. The timestamp server will return a hash and authenticated attributes that are signed using the private key of the TSA.

P.S : You need to be online to create a signature with a timestamp. Connecting to a timestamp server sending and receiving the hash takes time. If you need to sign thousands of documents in batch, you could ask the TSA to provide a time stamping certificate. This certificate will usually be stored on a Hardware Security Module.
You can subscribe to a time stamping service, in which case you get an URL and account information (a username and password). Or you can use a certificate that contains an URL of the time stamping server.
With code sample 3.12, we try to find out if CAcert also offers timestamping services.

we can extend it by adding a DSS and a timestamp as shown. In the DSS, we can store Validation Related Information (VRI). The DSS contains references to certificates, and we can add references to OCSP responses and CRLs that can be used to re-verify the certificates. To make sure that the DSS won't be altered, we sign the document including an authoritative timestamp.

The timestamp is itself signed, and so it's possible for the timestamp's own validation data to expire. If we want to extend the life of the signed document beyond the expiration date of the certificate used by the TSA, we need to add another DSS and document-level timestamp containing a CRL or OCSP response for the most recent TSA certificate
(Ref: Pg. 137)

Q. Eligibility of root certificates

https://helpx.adobe.com/acrobat/kb/approved-trust-list1.html

Appendix 1

Q. What is a digest algorithm?
When we create a digest, we're using a cryptographic hash function to turn an arbitrary block of data into a fixed-size bit string. The block of data is often called the 'message', and the hash value is referred to as the 'message digest'. In the context of PDF documents, the block of data could be the byte range of a PDF file. The result of this hash function is always identical, provided the message hasn't been altered. Any accidental or intentional change to the data will result in a different hash value. It's also a one-way algorithm.

Q. Which Hash algorithm to apply ?
MD5, SHA, and RIPEMD are implementations of different cryptographic hash algorithms.

MD5— one in a series of message digest algorithms designed by Professor Ron Rivest of MIT.
It was designed in 1991. The MD5 algorithm allows you to create a 128-bit (16-byte) digest of a message. In other words: there are 2 128 possible hashes for an infinite possible number of messages, which means that two different documents can result in the same digest. This is known as 'hash collision'. The quality of a hash function is determined by how 'easy' it is to create a collision. MD5 is still widely used, but it's no longer considered secure.

SHA— stands for Secure Hash Algorithm.
o SHA-1 was designed by the US National Security Agency (NSA). The 160-bit (20 bytes) hash was considered safer than MD5, but in 2005 a number of security flaws were identified.
o SHA-2 fixes these flaws. SHA-2 is a set of hash functions: SHA-224, SHA-256, SHA- 384, and SHA-512. The length of the message digest in SHA-2 can be 224 bits (28 bytes), 256 bits (32 bytes), 384 bits (48 bytes), or 512 bits (64 bytes).

not all of these digest algorithms can be used in the context of PDF
digital signatures

Q To detect whether or not a PDF file has been altered, we could create a message digest of those bytes and store it inside the PDF. Then when somebody changes the PDF, the message digest taken from the changed bytes in the byte range will no longer correspond with the message digest stored in the PDF. Does this solve our problems of data integrity, authenticity and non-repudiation?

Not yet. One could easily change the bytes of the PDF file, guess the digest algorithm that was used, and store a new message digest inside the PDF. To avoid this, we need to introduce the concept of encryption by means of asymmetric key algorithms.

We can choose an asymmetric key algorithm. One key cannot be derived from the other. Consumer can send their public key to Authority for authenticity.

Q. How the concept of encryption using an asymmetric key algorithm can be used for digital signing.

Consumer encrypt a message using his private key. Now when he/she share his/her public key —which is usually the intention— the whole world can decrypt his/her message. In this case, Consumer is not using the encryption mechanism to make sure a third party can't read his/her message, but to make it absolutely clear that he's/she's the author. If someone can decrypt a message using consumer's public key, its 100% certain that it was encrypted using consumer's private key. As only he/she have access to his/her private key

Q. Does this solve our problems of data integrity, authenticity and non-repudiation?

Yes, it does, because as soon as anybody alters an encrypted message sent from signer to reciever, reciever won't be able to decrypt it using signer public key. On the other hand: if reciever succeed in decrypting it, reciever is certain that sign is authentic, and author can't claim that he/she didn't sign the message while also claiming that private key isn't compromised.

Q. However, that's not how it's done in practice: encrypting and decrypting large messages requires large resources. Fails the goal to protect the message from being read by anybody.

If Signer encrypt only a digest of the message, and attach this 'signed digest' to the actual message. When reciever receive the message, reciever can decrypt the digest and hash the message for comparison. If there's a match, the message is genuine. If there isn't, the message was altered.

Appendix 2

Public-Key Cryptography Standards (PKCS) : PKCS#7

PKCS#7: Cryptographic Message Syntax Standard
This standard is published as RFC 2315. It's used to digitally sign, digest, authenticate or encrypt any form of digital data.

PKCS#11: Cryptographic Token Interface : PKCS#11 in signing a PDF

PKCS#12: Personal Information Exchange Syntax Standard
This standard defines a file format used to store private keys with accompanying public key certificates, protected with a password-based symmetric key. In practice, these files will have the extension .p12. It's usable as format for the Java key store. Note that you may also encounter .pfx files. PFX is a predecessor to PKCS#12.

Algorithms supported in PDF
RSA encryption algorithm (appendix 3)

The Digital Signature Algorithm (DSA) — this is a FIPS standard for digital signatures. RSA can be used for both encrypting and signing; DSA is used mainly for signing. DSA is faster in signing, but slower in verifying.
The Elliptic Curve Digital Signature Algorithm (ECDSA) — this is a new standard (PKCS #13). It will be introduced for signing PDFs in PDF 2.0; it's supported in iText, but it hasn't been tested yet. At the time this paper was written, it wasn't supported in Adobe Reader yet.

X.509 is a standard for a public key infrastructure (PKI) and privilege management infrastructure. It specifies, amongst other things, standard formats for public key certificates and certificate revocation lists.

Appendix 3

Generating private and public key

```
keytool -genkeypair -alias demo -keyalg RSA -keysize 2048 -keystore ks
```

'demo' name of alias
algorithm RSA – 2048 bit
keystore name : `ks`
just need to access the alias demo for this generated provate key : 2048 bit RSA key .

This generated private key will be used for encryption of the pdf byte range

-Need a root certificates (CA)
-Create a public and private key pair that is stored in a file named password.p12. Remember that the 12 refers to PKCS#12, the Personal Information Exchange Standard that defines the format of the key store.

```
Properties properties = new Properties();
properties.load(new FileInputStream("c:/home/blowagie/key.properties"));
String path = properties.getProperty("PRIVATE");
char[] pass = properties.getProperty("PASSWORD").toCharArray();
```

We get an instance of the key store using two parameters:
“pkcs12”, because our keys and certificates are stored using PKCS#12 .p12 file, and “BC” or provider.getName(); we’re using the BouncyCastleProvider.

CA Cert’s root certificate corresponds with the private key that was used to sign all the public certificates that are issued to the members of the CA Cert Community. You can trust CA Cert’s root certificate in the PDF

Certificate of the CA Cert Signing Authority, the root certificate (the final element in the chain). The CA Cert Signing Authority is the Issuer of my certificate.

all the people who have a certificate issued by this CA will be trusted. You no longer have to trust each individual separately.

Service provider has shared three certificate with us.

1. ABCD.cer
2. CA_Certificate.cer
3. CCA_Certificate.cer

I have to add them to keystore by creating a form chain for the SSL communication.I have followed below steps.

1. `keytool -keystore ks -genkey -alias demo`
   Result :- keystore npci_keystore_test.jks created.
2. `keytool -import -keystore ks -file CA_Certificate.cer -alias theCARoot`
   Result :- certificate CA_Certificate.cer is added to keystore.
3. `keytool -import -keystore ks -file CCA_Certificate.cer -alias theCCARoot`
   Result :- certificate CCA_Certificate.cer is added to keystore.

Appendix 4

```
public void sign(String reason, String location)   // attributes to be displayed in signature
throws GeneralSecurityException, IOException, DocumentException {

String src = " source pdf file (path)"
String dest = "destination file ( resultand pdf after signature) (path)"
Certificate[] chain = Use the self generated cerficate or issued certificate
PrivateKey pk = private key generated by the keytool for the encryption
String digestAlgorithm = The encryption method ("RSA", "DSA" or "ECDSA") //[ Till my
                                                                    //  knwoldge supported on pdf ]
String provider = "name of the provide: ProviderDigest class. / Bouncy Castle class"
CryptoStandard subfilter = "CryptoStandard.CADES / CryptoStandard.CMS"

/*
we need an implementation of the ExternalDigest interface to create a digest
and of the ExternalSignature interface to create the signature, a process that involves
hashing as well as encryption.

We can use Bouncy Castle as security provider for the digest by choosing an instance of
the BouncyCastleDigest class. If you want another provider, use the ProviderDigest class.

iText has only one implementation for the signing process: PrivateKeySignature;
*/

/* same steps for all signature from now on loading and creating pdf */

// Creating the reader and the stamper
PdfReader reader = new PdfReader(src);
FileOutputStream os = new FileOutputStream(dest);
PdfStamper stamper = PdfStamper.createSignature(reader, os, '\0');

/*
The zero byte means we don't want to change the version number of the PDF file.
The Boolean value indicates whether or not we want to manipulate the file in 'append
mode'. This value is false by default. The original bytes aren't preserved. By changing
this value to true, we tell iText not to change any of the original bytes.
Inserting page isn't allowed
*/

/* beautification of the signature */

// Creating the appearance
PdfSignatureAppearance appearance = stamper.getSignatureAppearance();
appearance.setReason(reason);
appearance.setLocation(location);
appearance.setVisibleSignature(new Rectangle(36, 748, 144, 780), 1, "sig");
```

/*lower left and upper right coordinates , page no. , field name in the pdf code */

// Creating the signature
ExternalDigest digest = new BouncyCastleDigest();
ExternalSignature signature = new PrivateKeySignature(pk, digestAlgorithm, provider);
MakeSignature.signDetached(appearance, digest, signature, chain, null, null, null, 0,
subfilter);
}

/*

We're using the signDetached() method, which means we're creating a detached signature, and we can choose between adbe.pkcs7.detached and ETSI.CAdES.detached. This is done with an enum named CrypoStandard in MakeSignature: use either CMS or CADES.

These null need to filled for an authentic signature
*/

appearance.setImageScale(1) original size of image
appearance.setImageScale(0.5) half the size of image
appearance.setImageScale(2) double the size of the image
appearance.setImageScale(-ve value) : no distortion

PdfTemplate n0 = appearance.getLayer(0);
float x = n0.getBoundingBox().getLeft();
float y = n0.getBoundingBox().getBottom();
float width = n0.getBoundingBox().getWidth();
float height = n0.getBoundingBox().getHeight();
n0.setColorFill(BaseColor.LIGHT_GRAY);
n0.rectangle(x, y, width, height);
n0.fill();
// Creating the appearance for layer 2
PdfTemplate n2 = appearance.getLayer(2);
ColumnText ct = new ColumnText(n2);
ct.setSimpleColumn(n2.getBoundingBox());
Paragraph p = new Paragraph("Demo signature");
ct.addElement(p);
ct.go();

/
Why are you only using layer 0 and layer 2? What happened
to layer 1? The answer is: there used to be a layer 1, 3 and 4 (and you'll find references to
them in iText), but you should no longer use them.
/

RenderingMode.DESCRIPTION—this is the default, it just shows whatever description

was defined for layer 2.
RenderingMode.NAME_AND_DESCRIPTION—this will split the signature field in two and add the name of the signer on one side, the description on the other side.
RenderingMode.GRAPHIC_AND_DESCRIPTION—this will split the signature field in two and add an image on one side, the description on the other side.
RenderingMode.GRAPHIC—the signature field will consist of an image only; no description will be shown

WARNING: support for CAdES is very new. Don't expect versions older than Acrobat/Reader X to be able to validate CAdES signatures! Acrobat 9 only supports signatures as described in the specification for PDF 1.7, and CAdES is new in PDF 2.0.

NOTE: The methods getOverContent() and getUnderContent() give you the option to write to the direct content on a layer that goes on top of or below the existing content. They don't give you access to the layer with the existing content. You can't use these methods to replace existing content, nor to complete it. It's not possible to say: "I want to add the words 'Hello people!' after the words 'Hello World'." You can only add those words to the layer above or below the existing content at an absolute position whose coordinates you know.

Suppose we switched the order. Suppose that Certifying authority certifies later and signer signs first certification signatures, ISO-32000-1 mentions the author of a document as 'the person applying the first signature' 14 '
certification signature doesn't have to be the first signature in the document. I don't know if it makes sense, but Adobe Reader validates documents that are signed by approval signatures first, followed by a certification signature as correct, provided that the certification level allows form filling.

Correct order :

Certifying authority with form silling allowance
Approval signature
other signatures ...

Reference :
Complete basis of this whole do

http://pages.itextpdf.com/rs/204-LXA-936/images/digitalsignatures20130304.pdf?mkt_tok=eyJpIjoiTVdVM05tUTRNR0k0TmpSaSIsInQiOiJFUFdXaGhkY0MxdlVUNkhXUWdmeld6ZEFJdGE5dVdEajA4UitIQWkzSmhtQ2ZHVk92MmZRcFVxcTFVK0hXMnhNV1wvdGpNRlZWb25JN3RDbFlsN3hhcnd1d1d6MnpGWjh5Z0Nja2trckhDdXBqVzZcL1dRdHJoMUcxOHpUcVwvVXNcL24ifQ%3D%3D

http://what-when-how.com/itext-5/adding-content-with-pdfstamper-part-1-itext-5/

Procedure for importing CA certificate into common alias

http://www.entrust.net/knowledge-base/technote.cfm?tn=8028

OR

https://www.sslshopper.com/article-most-common-java-keytool-keystore-commands.html