Public - jOSA

- **Javadoc**
- **Licenses**
- **Sample**

JOSA can be used to implement a signing workflow on the following platforms: Linux, Windows and MacOS. Java Version 1.8 and higher are supported. Document viewing, sign in doc, standard signing with and without hash dialog are supported.

Maven pom.xml

```
<repositories>
    <repository>
        <id>stepover-snapshots</id>

<url>http://nexus.stepover.de:8081/repository/maven-snapshots/</url>
        <releases>
            <enabled>false</enabled>
        </releases>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>stepover-releases</id>

<url>http://nexus.stepover.de:8081/repository/maven-releases/</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
<dependencies>
    <dependency>
        <groupId>com.stepover</groupId>
        <artifactId>josa</artifactId>
        <version>1.xx-SNAPSHOT</version>
    </dependency>
</dependencies>
```

You can access our maven repository to download our Java API (jOSA) with the following data

http://nexus.stepover.de:8081/#browse/browse

User: stepover

PW: letmein

Initialization

```
SignatureServiceProvider serviceProvider = new NativeServiceProvider();
```

Device search

```
SigningDevice[] devices = serviceProvider.getSigningDevices();
```

Configuration

```
SoDeviceConfiguration config = new SoDeviceConfiguration();
...
signingDevice.send( new ConfigurationMessage( config ) );
```

** only for durasign devices*

config.setPadmodeAfterSigning(int mode)

sets the pad mode after signing has completed, possible values are SoDeviceConfiguration.PADMODE_MANUFACTURER_LOGO

config.setSigningTimeout(int timeout)

sets the signature time out in milliseonds. If no buttons are shown during signing, the signature is accepted after this time span has passed without signature input.

config.setShowButtonsDuringSignature(boolean showButtons) *

sets if buttons (accept, repeat, cancel) should be shown during signing

config.setStartSigningCallback(SoDeviceConfiguration.StartSigningCallback callback)

if this callback is set, the start signing button is shown during document viewing and pressing the button calls the callback.

config.setResetDisplayRotation(int rotation) *

when the device is closed, this rotation is set. Possible values are 0, 90, 180

config.setSignatureReason(String signatureReason)

sets the reason inserted into the adobe signature

config.setDeviceDisconnectCallback(SoDeviceConfiguration.DeviceDisconnectCallback callback)

sets the callback that gets called when a device is diconnected

config.setEmulationMode(boolean state)

sets if the hash dialog should be used, false if it should be used, true otherwise

config.setSignatureFieldSizeInPercent(int percent)

defines the percentage of the display width used for the signature field. This only has an effect if sign in doc is used. On 10 inch pads in landscape mode the percentage is reduced some what internally, 100% width would be too large.

config.setSignatureHashCallback(SoDeviceConfiguration.SignatureHashCallback callback)

this callback is called delivering the hash value displayedin the hash dialog.

Document viewing

start document viewing with:

```
ApplyDocumentRenderer applyDocumentRenderer = new ApplyDocumentRenderer(
new FileInputStream( new File( "document.pdf" ) ), true );
signingDevice.send( applyDocumentRenderer );
```

sign in doc is always used if document viewing is used. To stop document viewing:

```
ApplyDocumentRenderer applyDocumentRenderer =
ApplyDocumentRenderer.stopViewing();
signingDevice.send( applyDocumentRenderer );
```

Sign in doc

To use sign in doc without document viewing:

```
ApplyDocumentRenderer applyDocumentRenderer = new ApplyDocumentRenderer(
new FileInputStream( new File( "document.pdf" ) ), false );
signingDevice.send( applyDocumentRenderer );
```

This has to be called before signing is started, it loads the document renderer. If no renderer is loaded, the standard sign mode is used.

Start signing

1. define the signature rectangle

```
Rectangle signatureRectangle = new Rectangle( resolution,
Conversion.mmToDots( resolution, x ),
                                        Conversion.mmToDots( resolution,
y ),
                                        Conversion.mmToDots( resolution,
width ),
                                        Conversion.mmToDots( resolution,
height ) );
```

2. create a signer instance

```
Signer signer = serviceProvider.createSigner( signingDevice );
```

3. define meta information (optional)

for example:

```
MetaInfoField[] metaData = { new SignatureColor( Color.RED ), new
SignatureBackgroundColor( new Color( 0, 255, 0, 100 ), 0.75f ), new
SignatureBorder( Color.BLUE, 8 ) }
```

SignatureColor(Color color)

defines the color of the signature line on the signature pad

SignatureBackgroundColor(Color color, float alpha)

defines the background color of the signature field, alpha between 0 and 1

SignatureBorder(Color color, int thickness)

defines the signature borders color and thickness in pixels

4. create signature context

```
SignatureContext signatureContext = new SignatureContext( inputstream,
metaData, pageNumber, signatureRectangle, true ) //last parameter is not
used
```

input stream is the pdf to sign, for example a FileInputStream: new FileInputStream(new File("doc-to-sign.pdf"))

5. start signing

```
signer.startSigning( signatureContext, new SignatureListener()
{

   public void updateRendering( final BufferedImage img )
   {

   }

   public void signatureFinished( SignatureResult result )
   {

   }
} );
```

updateRendering is called during signing delivering the signature image enlarged by the factor 2.

signatureFinished is called when the signature process has finished, result contains the state (done, failed or canceled), the signature data: signature image, encrypted bio data and signed document.

6. close device

```
signingDevice.close();
```

this stops all threads.