

Sign Live! CC Security Applications Developers Guide

Februar 2022

intarsys GmbH

Sign Live! CC Security Applications Developers Guide

Version 7.1

Developing applications and workflows using Sign Live! CC

intarsys GmbH
Sign Live! CC Security Applications Developers Guide
Version 7.1

All rights reserved
© 2021 intarsys GmbH
www.intarsys.de

Preface

- Author and company

This book has been provided by different authors from the development staff of intarsys GmbH.

- Trademarks

Wherever possible and where the authors were aware of a trademark claim, such designations are marked as trademarks in this book.

CABAReT is a registered trademark of intarsys (Schweiz) AG.

EForm is a registered trademark of intarsys GmbH.

jPod is a trademark of intarsys GmbH.

Sun, Java and JavaScript are trademarks of Sun Microsystems

Microsoft and Windows are trademarks of Microsoft Corporation.

Adobe and Acrobat are trademarks of Adobe Systems Incorporated

- Code examples

Most of the examples contained in this book are given as JavaScript code. You should be familiar with this programming language and its integration in Sign Live! CC.

Source code for this examples are contained in the standard application installation subdirectory “demo”.

In this book for most examples, scripting code is separated from the CodeExit calling it. While you may add the scripting code literally in the CodeExit declaration, this has the following advantages:

One day you will wake up and can no longer recognize what you have done. This embedded code is extremely hard to read and maintain.

Embedding JavaScript in XML is error prone because of the nested string delimiter quotes

An external script is “hot replaced” upon a change, while upon an instrument declaration change the application must be restarted.

So, in these examples we will separate the script code in a file of its own. You can address the script relative to the instrument or web application directory.

In many examples you need a certificate or an identity (private key) - in most cases the examples are using the Sign Live! CC demo certificate, deployed in the standard key store. Here we repeat for your reference the serial number and password for this certificate:

Serial number 8139571262270123122

Password password

■ Who should read this book

This book is intended for application scripters or programmers. The concepts described in the Sign Live! CC Developer’s Guide are prerequisite knowledge.

Basic knowledge of PPK based security applications is assumed, this book will not explain PPK concepts and algorithms.

■ Organization

This book has a single chapter for each security application.

Here you will learn the concepts and syntax and get a lot of examples for interfacing these features using the Sign Live! CC API’s.

The last chapter is dedicated to smartcard management tools.

■ Reviews and comments

We make constant efforts to improve our documentation and meet your requirements. Your comments are welcome and are a valuable resource for us.

Email support@intarsys.de

Website www.intarsys.de

■ Disclaimer

Every effort has been made to make this book as complete and accurate as possible, but no warranty is implied.

The information is provided “as is”. The authors shall are in no way liable to any person or entity with respect to any loss or damages

Preface

arising from the information contained in this book, or from the use of the disks or programs that may accompany it.

Contents

Preface	5
▪ Author and company	5
▪ Trademarks	5
▪ Code examples	5
▪ Who should read this book	6
▪ Organization	6
▪ Reviews and comments	6
▪ Disclaimer	6
Contents	9
Introduction	15
1. Signature	17
1.1 Overview	17
1.2 The Document	18
1.3 The Digester	18
1.3.1 Overview	18
1.3.2 Supported algorithms	18
1.4 The Device	19
1.4.1 Overview	19
1.4.2 Local keystore signing device	19
1.4.3 Smartcard signing device	21
1.4.4 Swisscom AIS static signing device	22
1.4.5 Swisscom AIS on-demand signing device	23
1.4.6 Remote security device	24
1.4.7 cloud suite gears signing device	31
1.5 The Method	33
1.5.1 The signing method	33
1.5.2 Generic signing method	33
1.5.3 PDF signing method	34
1.5.4 PKCS#7 signing method	44
1.5.5 Internal XML Signature	47
1.5.6 External XML Signature	50

Content

1.6	The Wizard	55
1.6.1	The signing wizard	55
1.6.2	Examples	56
1.7	Timestamp Devices	56
1.7.1	Overview	56
1.7.2	Timestamp device lookup	56
1.7.3	HTTP Timestamp Device	57
1.8	Explicit Policy Electronic Signatures (EPES)	58
1.8.1	Overview	58
1.8.2	Defining signature policy references	59
1.8.3	Defining commitment types	60
1.8.4	Examples	61
1.9	Examples	62
1.9.1	Overview	62
1.9.2	PDF signature on an HTTP uploaded document using a local certificate	62
1.9.3	PDF signature using a local certificate via XMLRPC	64
1.9.4	PKCS#7 signature using a smartcard via ActiveX	65
1.9.5	PKCS#7 signature, Commandline version	66
1.9.6	PDF signature for use in Script Center	67
1.9.7	Signature wizard activation via ActiveX	68
2.	Validation	71
2.1	Overview	71
2.2	The Document	71
2.3	The Method	72
2.3.1	Generic document validation method	72
2.3.2	Certificate validation method	73
2.4	Examples	74
2.4.1	Overview	74
2.4.2	Validation of an HTTP uploaded document	74
2.4.3	Validation triggered via XMLRPC	76
2.4.4	Validation triggered via ActiveX	77
2.4.5	Document validation, Commandline version	78
2.4.6	Validation for use in Script Center	79
3.	Encryption	81
3.1	Overview	81
3.2	The Document	82
3.3	The Device	82
3.3.1	The symmetric encryption algorithm	82
3.3.2	The asymmetric encryption device	83
3.3.3	Local keystore encryption device	83
3.3.4	Smartcard encryption device	84
3.4	The Method	85
3.4.1	The encryption method	85
3.4.2	Generic encryption method	86
3.4.3	PKCS#7 encryption method	87
3.5	The Wizard	89

3.5.1	The encryption wizard	89
3.5.2	Examples	90
3.6	Examples	90
3.6.1	Overview	90
3.6.2	PKCS#7 encryption using a local keystore via ActiveX	90
3.6.3	Encryption wizard activation via ActiveX	91
4.	Decryption	93
4.1	Overview	93
4.2	The Device	94
4.2.1	Overview	94
4.2.2	Local keystore decryption device	94
4.2.3	Smartcard decryption device	95
4.3	The Method	96
4.3.1	The decryption method	96
4.3.2	Generic decryption method	97
4.3.3	PKCS#7 decryption method	97
4.4	The Wizard	99
4.4.1	The decryption wizard	99
4.4.2	Examples	100
4.5	Examples	100
4.5.1	Overview	100
4.5.2	Simple Decryption wizard activation via ActiveX	100
4.5.3	Headless Decryption wizard via ActiveX	100
5.	Workflow integration	103
5.1	Overview	103
5.2	Cosima	103
5.2.1	Conversion	104
5.2.2	Tagging (keywords)	105
5.2.3	Signature	106
5.2.4	Mail	106
5.2.5	Archive	108
5.2.6	Notification	108
5.2.7	Installation	109
5.2.8	Configuration	109
5.2.9	Tipps and Tricks	112
5.2.10	Arguments	112
6.	Workflow Processors	116
6.1	Postscript Conversion	116
6.1.1	Overview	116
6.1.2	Synopsis	116
6.2	PDF “@@” tag extraction	117
6.2.1	Overview	117
6.2.2	Caveats	118
6.2.3	Synopsis	118
6.3	Tags in property file format	118
6.3.1	Overview	118

Content

6.3.2	Synopsis	119
6.3.3	Example	119
6.4	Tags in XML file format	119
6.4.1	Overview	119
6.4.2	Synopsis	119
6.4.3	Example	120
6.5	All Mail processors	121
6.5.1	Overview	121
6.5.2	Mail Content	121
6.5.3	Synopsis	121
6.6	Mail via MAPI	122
6.6.1	Overview	122
6.6.2	Synopsis	122
6.7	Mail via SMTP	122
6.7.1	Overview	122
6.7.2	Synopsis	122
6.8	Mail via AppleScript	123
6.8.1	Overview	123
6.8.2	Synopsis	123
6.9	File system archiving	123
6.9.1	Overview	123
6.9.2	Synopsis	123
7.	Advanced Signature Features	125
7.1	Pools	125
7.1.1	Pools	125
7.1.2	Signing with Pools	125
7.2	Session	130
7.2.1	Overview	130
7.2.2	Opening a session	131
7.2.3	Signing on a session	132
7.2.4	Closing a session	133
7.2.5	Expiration	134
8.	Monitoring	135
8.1	Overview	135
8.2	Pool MBean	135
8.3	Notification String expansion	137
9.	Smartcard Tools	139
9.1	Overview	139
9.2	Credential Changer	139
9.2.1	Overview	139
9.2.2	Examples	141
9.3	Credential changer wizard	142
9.3.1	Examples	142
9.4	Credential Resetter	142
9.4.1	Overview	142
9.4.2	Examples	143

9.5	Credential reset wizard	143
9.5.1	Examples	144

Introduction

Sign Live! CC comes along with a powerful security application framework. You can sign, validate, encrypt and decrypt any document format with a broad range of settings.

Best of all - these features are published, as always, using the standard API's of Sign Live! CC. For those who missed it until now – these API's are for example

- JavaScript
- Commandline
- ActiveX
- HTTP

More on the interoperability of Sign Live! CC you will find in the “Developer’s Guide”. In this book we will concentrate on syntax and semantics of the security specific components and special workflows of the application.

1. Signature

1.1 Overview

This chapter presents the signature features for Sign Live! CC. You will learn how to sign a document using different methods like “PKCS#7” or “PDF internal”, on different devices like local key stores or smartcards.

You will see how to do this completely programmatically or by calling the interactive wizard of Sign Live! CC itself.

The basic steps for creating signatures using PPK techniques are

- Select a document

- Create a digest from a document

- Sign the digest using an appropriate device

- Combine the signature with the document in some way

Return the signature container created. The concrete implementation depends on the signing method.

The security framework provides some abstractions for the steps and objects involved.

Document

The data to be signed. The document object is important to identify document specific or proprietary ways of creating signatures. This is for example the case with PDF internal signatures.

Digester

The algorithm used to create the documents digest. The selection of a digester may be interdependent with the choice of the digest signer and signature method. An example for a digester is “SHA1” or “RIPEMD160”.

Digest Signer

The algorithm and technical device used to create the signature. “Device” is an abstraction that can be implemented locally, using

keystores or the windows certificate repository or a security token like a smartcard. The result of this step is a signed digest.

Signature Method

The method used to create a signature. This describes the process applied on the source document and the syntax and semantics of the serialized signature object. A signature method may be a vendor defined one (like a PDF internal signature) or a de facto standard like PKCS#7. The result of this step is a signature container.

Most of these you can freely assemble to create the signature application of choice.

1.2 The Document

A document is represented using the *de.intarsys.document.model.IDocument* abstraction from the application platform. More information on this topic you can find in “Developers Guide”.

You will find at least basic support for these document types:

- PDF
- HTML
- Text flavors
- Image flavors (BMP, PNG, Tiff, GIF, JPG)
- Generic transparent document (binary data)

Some signature methods support all document types, some are restricted to specific types. You will find this information along with the signature method.

1.3 The Digester

1.3.1 Overview

The digester is the reference to the algorithm to be used for creating the document's digest.

Only thing to take into account is that not all digest signer devices will support all digester algorithms. For a list of compatible assignments, see the online help or user manual.

1.3.2 Supported algorithms

All message digest algorithms available with Java (JCA) are supported. This is a list of common digesters for use with signing (in ascending strength of cryptographic security)

- SHA1
- RIPEMD160
- RIPEMD256
- SHA256

- SHA384
- SHA512

1.4 The Device

1.4.1 Overview

As the digest signer device is needed for the signature method, we will introduce that first.

The digest signer is an abstraction from both the algorithm used (for example RSA or elliptic curves) as well as the device (local CPU, security token, remote implementation, . . .).

The digest signer is implemented using the *de.intarsys.processor.model.IProcessor* abstraction from the application platform. More information on this topic you can find in “Developers Guide”.

The following digest signer devices are supported:

- Local keystore
- Smartcard
- Remote security device
- HSM modules

Some of these implementations may not be available with your installation, depending upon the license you achieved.

More implementations exist, for example the combination of biometrical information via handwritten signatures into the signed document using tablet devices.

1.4.2 Local keystore signing device

1.4.2.1 Overview

Signing using an identity (private key) from a local keystore is implemented with the local keystore signing device.

The local keystore digest signer supports identities (private keys) stored locally on the machine. It uses by default the Sign Live! CC key store implementation. Here you can access identities from the windows certificate repository or import from all popular formats like Java keystore or PKCS#12.

More about this key store can be found in the documentation about the certificate administration.

PROCESSOR FACTORY

`com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory`

ARGUMENTS

Name	Description
signerIdentifier	<p>Select the identity (private key) used for signing the digest data.</p> <p>This is a polymorphic argument that can accept a</p> <ul style="list-style-type: none"> • de.intarsys.security.cert.IX509Certificate • de.intarsys.security.cert.IX509CertificateSelector String identifying a certificate. In this string you can use the SerialNumber, Subject and Issuer to select the certificate from the store.
signerPassword	The password to open the signers private key.
attributeCertificates	<p>A single or a list of attribute certificates, which target the signer certificate and refine its usage in this situation. The usage of this parameter is optional.</p> <p>This is a polymorphic argument that can accept a single or a list of</p> <ul style="list-style-type: none"> • de.intarsys.security.cert.IX509Certificate • de.intarsys.security.cert.IX509CertificateSelector String identifying a certificate. In this string you can use the SerialNumber, Subject and Issuer to select the certificate from the store.

RESULT

The processor returns an object of type byte[].

EXCEPTIONS

The processing may raise exceptions.

1.4.2.2 Example

You will not call a digest signer directly, so here no example is given. A digest signer is used as a parameter to a signer method.

For reference purposes we will give you here the argument values needed to sign with the demo certificate in the local key store.

```
signerIdentifier: 'SerialNumber:8139571262270123122;'
signerPassword: 'password'
```

1.4.3 Smartcard signing device

1.4.3.1 Overview

This signature implementation uses a smartcard device. The implementation supports access to advanced and qualified certificates.

For a list of supported smartcards and smartcard readers see the operating documentation or online help.

PROCESSOR FACTORY

de.intarsys.stage.security.device.smartcard.signature.SmartcardDigestSignerFactory

ARGUMENTS

Name	Description
signerIdentifier	<p>Select the identity (private key) used for signing the digest data.</p> <p>This is a polymorphic argument that can accept a</p> <ul style="list-style-type: none"> • de.intarsys.security.cert.IX509Certificate • de.intarsys.security.cert.IX509CertificateSelector <p>String identifying a certificate. In this string you can use the SerialNumber, Subject and Issuer to select the certificate from the store.</p>
signerPassword	<p>The password to open the signers private key (PIN) .</p> <p>This is not available for qualified signatures.</p>
attributeCertificates	<p>Select a single or a list of attribute certificates, which target the signer certificate and refine its usage in this situation. The usage of this parameter is optional.</p> <p>This is a polymorphic argument that can accept a single or a list of</p> <ul style="list-style-type: none"> • de.intarsys.security.cert.IX509Certificate • de.intarsys.security.cert.IX509CertificateSelector <p>String identifying a certificate. In this string you can use the</p>

SerialNumber, **Subject** and **Issuer** to select the certificate from the store.

RESULT

The processor returns an object of type `byte[]`.

EXCEPTIONS

The processing may raise exceptions.

1.4.3.2 Example

You will not call a digest signer directly, so here no example is given. A digest signer is used as a parameter to a signer method.

1.4.4 Swisscom AIS static signing device

1.4.4.1 Overview

This device implements signing using the AIS web service offered by Swisscom. The signature is performed using a static signing certificate installed at Swisscom.

PROCESSOR FACTORY

`de.intarsys.stage.security.device.ais.signer.AisStaticDigestSignerFactory`

ARGUMENTS

Name	Description
<code>addRevocationInformation</code>	true if you want to include revocation information for all certificates related to the signature. The default is "false" .
<code>addTimestamp</code>	true if you want to include a signature timestamp into the signature. The default is "false" .
<code>keyEntity</code>	The key entity name identifying the signing key pair to use.

RESULT

The processor returns an object of type `byte[]`.

EXCEPTIONS

The processing may raise exceptions.

1.4.4.2 Example

You will not call a digest signer directly, so here no example is given. A digest signer is used as a parameter to a signer method.

1.4.5 Swisscom AIS on-demand signing device**1.4.5.1 Overview**

This device implements signing using the AIS web service offered by Swisscom. The signature is performed using a dynamically created certificate issued by Swisscom. These on-demand certificates usually are subject to a very short validity period. Before issuance, the announced certificate owner's identity is confirmed using his Mobile ID.

PROCESSOR FACTORY

de.intarsys.stage.security.device.ais.signer.AisOnDemandDigestSignerFactory

ARGUMENTS

Name	Description
addRevocationInformation	true if you want to include revocation information for all certificates related to the signature. The default is " false ".
addTimestamp	true if you want to include a signature timestamp into the signature. The default is " false ".
keyEntity	The key entity name identifying the signing key pair to use.
distinguishedName	The subject distinguished name to be certified.
mobileIDMSISDN	The Mobile ID phone number.
mobileIDLanguage	The language used in the Mobile ID notification.

mobileIDMessage

The message displayed on the mobile device.

RESULT

The processor returns an object of type byte[].

EXCEPTIONS

The processing may raise exceptions.

1.4.5.2 Example

You will not call a digest signer directly, so here no example is given. A digest signer is used as a parameter to a signer method.

1.4.6 Remote security device**1.4.6.1 Overview**

Using the remote security device you can forward local requests to another device hosted by another installation on a networked host.

Together with the session device, one can establish a secure and reliable infrastructure for a comfort signature, where the client needs to be authenticated only once with the physical device within a session lifecycle.

This is for example a very useful solution with mobile agent scenarios (like an applet), where the agent now can reattach to the already existing session.

Physically the remote device consists of a client part – the “Remote device” and the server part – the “Remote device service”.

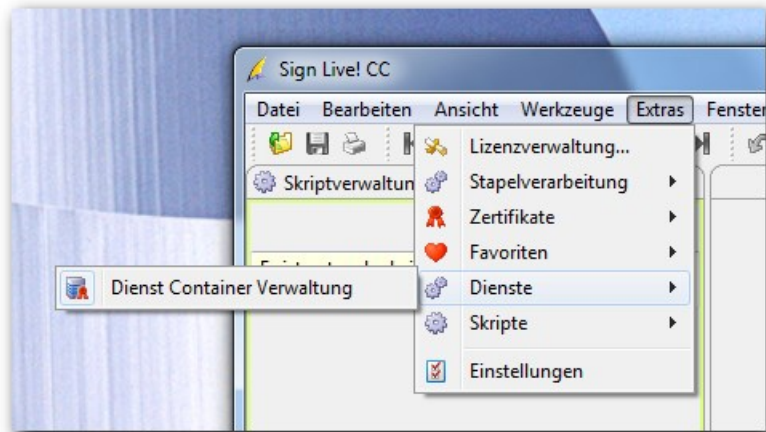
1.4.6.2 Sign Live! CC server configuration

The remote device is configured in Sign Live! CC itself.

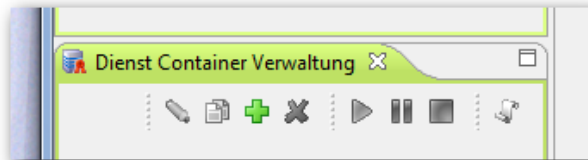
To provide a quickstart, a complete configuration instrument is included in the deployment in the directory "<install>/demo/Service/RemoteDevice/SmartcardSession". You can copy the instrument in the "instrument" directory of your local Sign Live! CC installation to get a working remote device.

Alternatively you can set up a remote device from scratch using the service container console in Sign Live! CC.

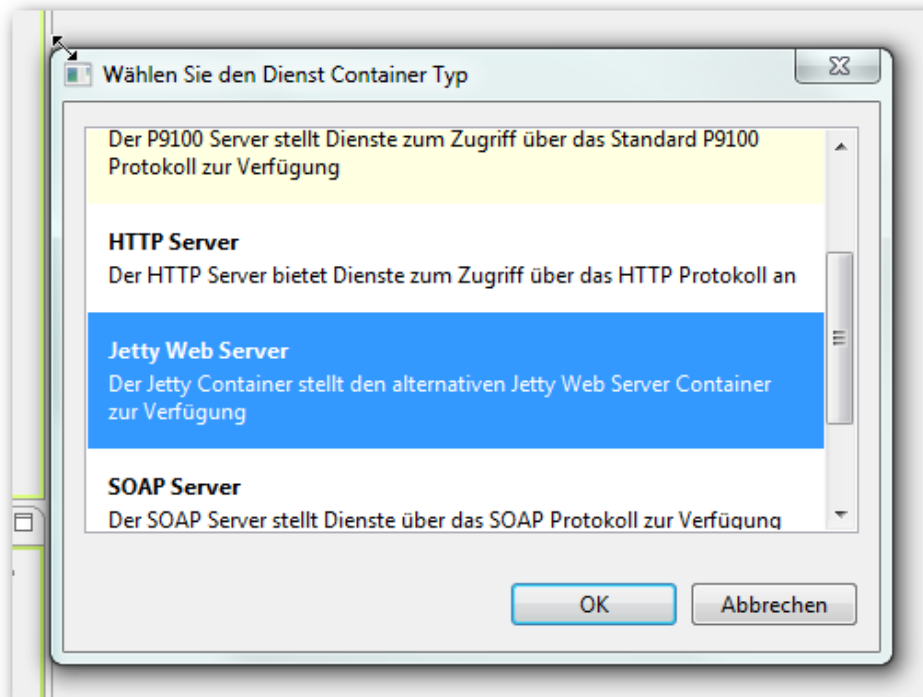
If not already visible in the lower left of your Sign Live! CC window, first open the service console (Menu → Extras → Services → Service Container Console).



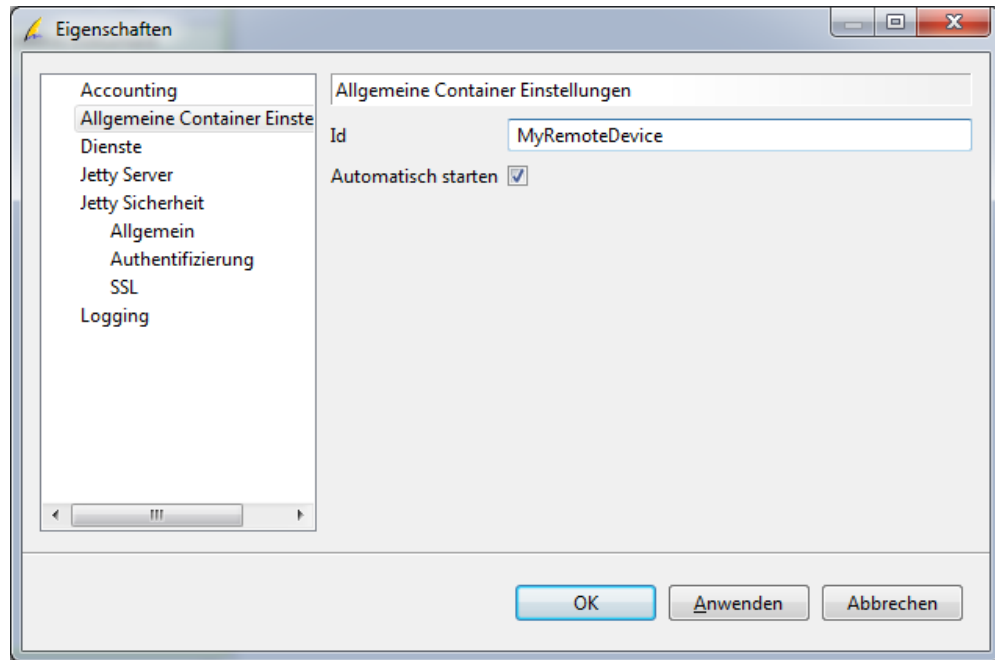
In the service container console you click the plus icon to add another service container.



The next dialog queries the type of service container - we need a "Jetty Web Server" here.



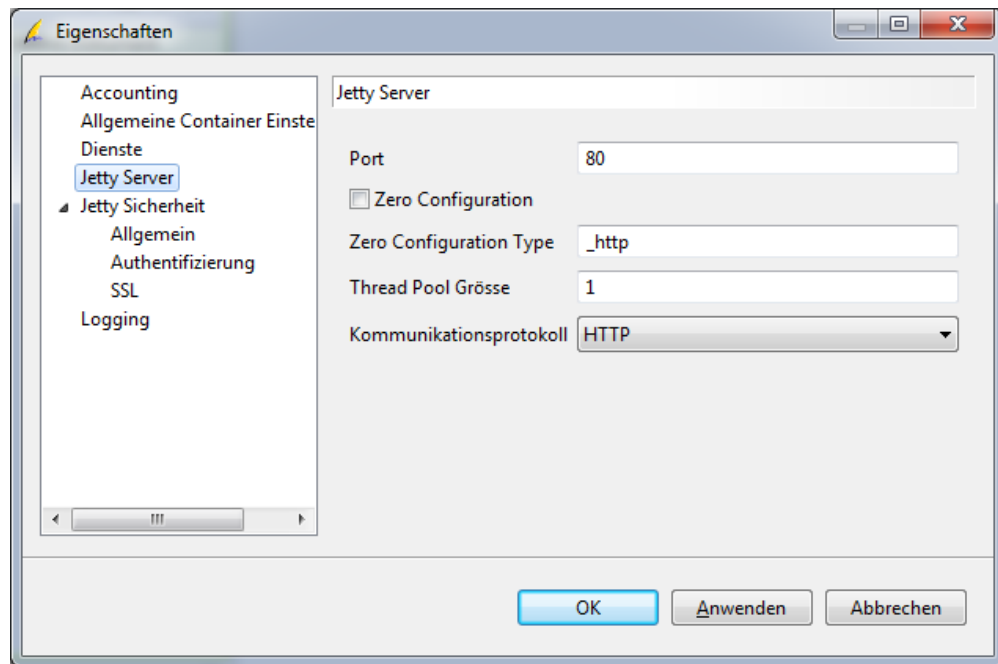
You should give the container a descriptive name and check the "autostart" check box to ensure the service will be available every time you start Sign Live! CC.



In the settings for the web server itself, be sure to select a port that is not already used by another application and that is not restricted by some security constraints on your system. If you set the port to "0", the application selects a useable port on its own. This is a very useful technique in combination with the "Zero Configuration" settings.

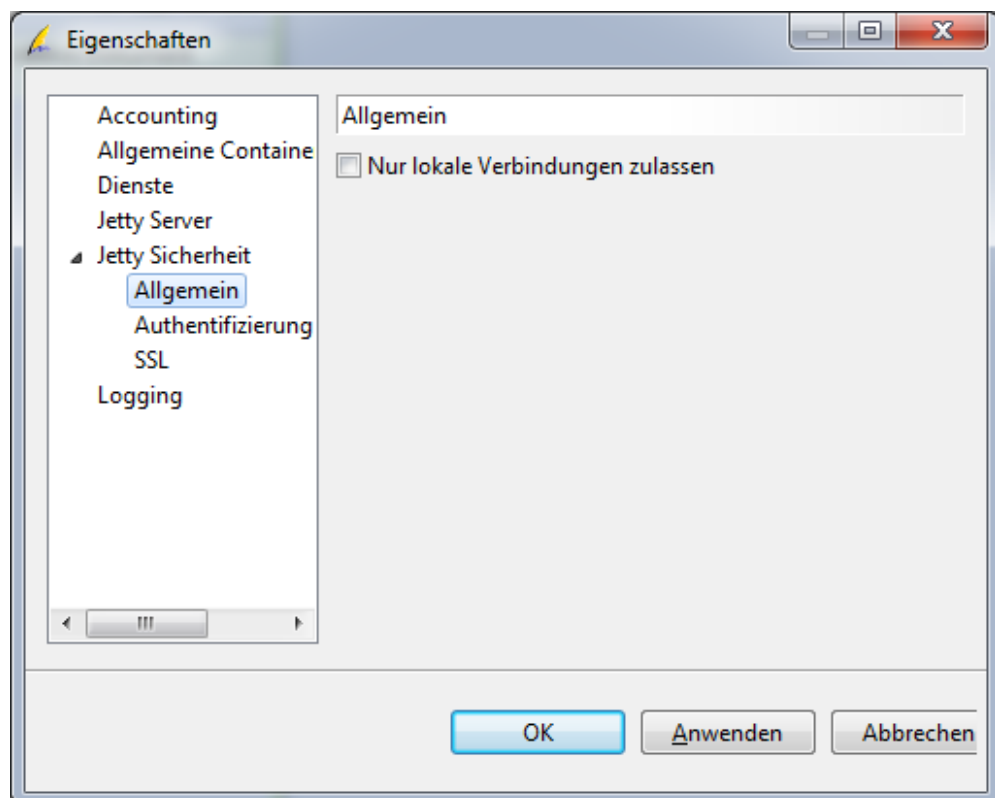
If you check "Zero Configuration" (recommended), your new service will be advertised using the "Zero Configuration networking" protocol, a de facto standard for automatic service discovery. This way your client does not need to know the port number for the service in advance. It simply looks up the service and its current parameters before connecting.

If you use this feature, we recommend setting the "Zero Configuration Type" to "_httprpc" for the remote device. In any case, you must use this prefix for configuring the lookup on the client side.



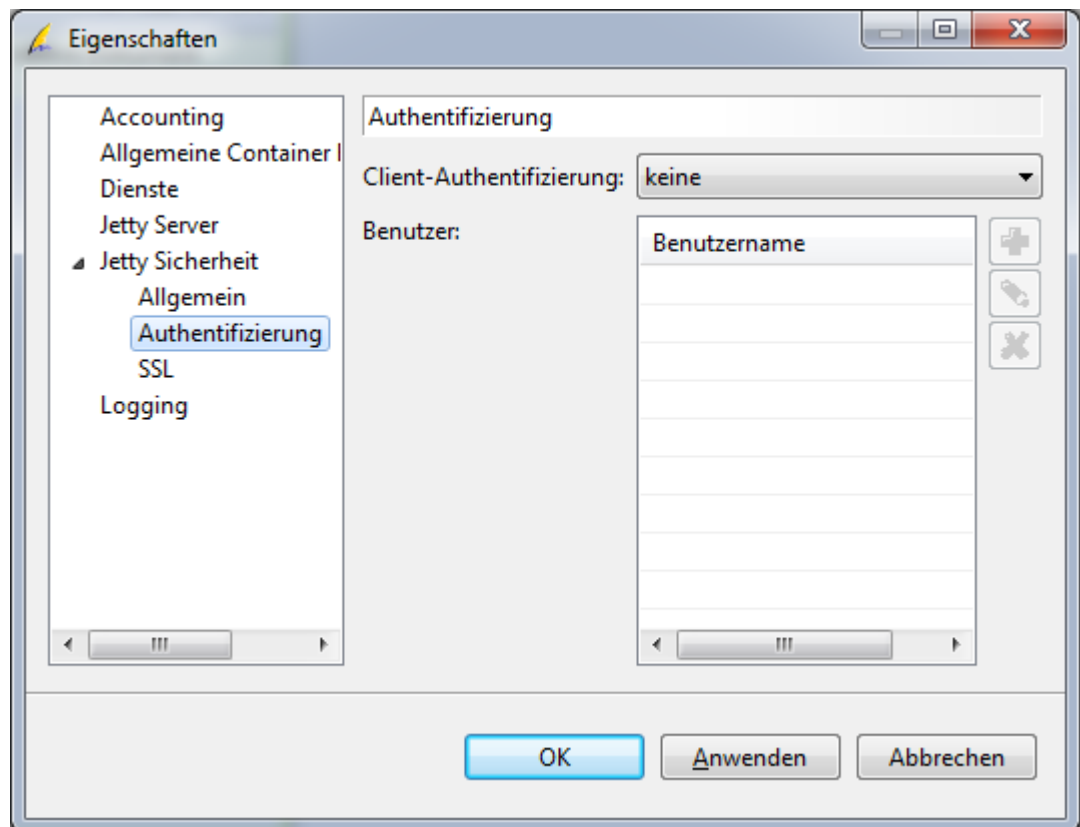
Leave "Thread Pool" and "Protocol" alone.

The security settings are quite important for the remote device: First, you choose common settings.



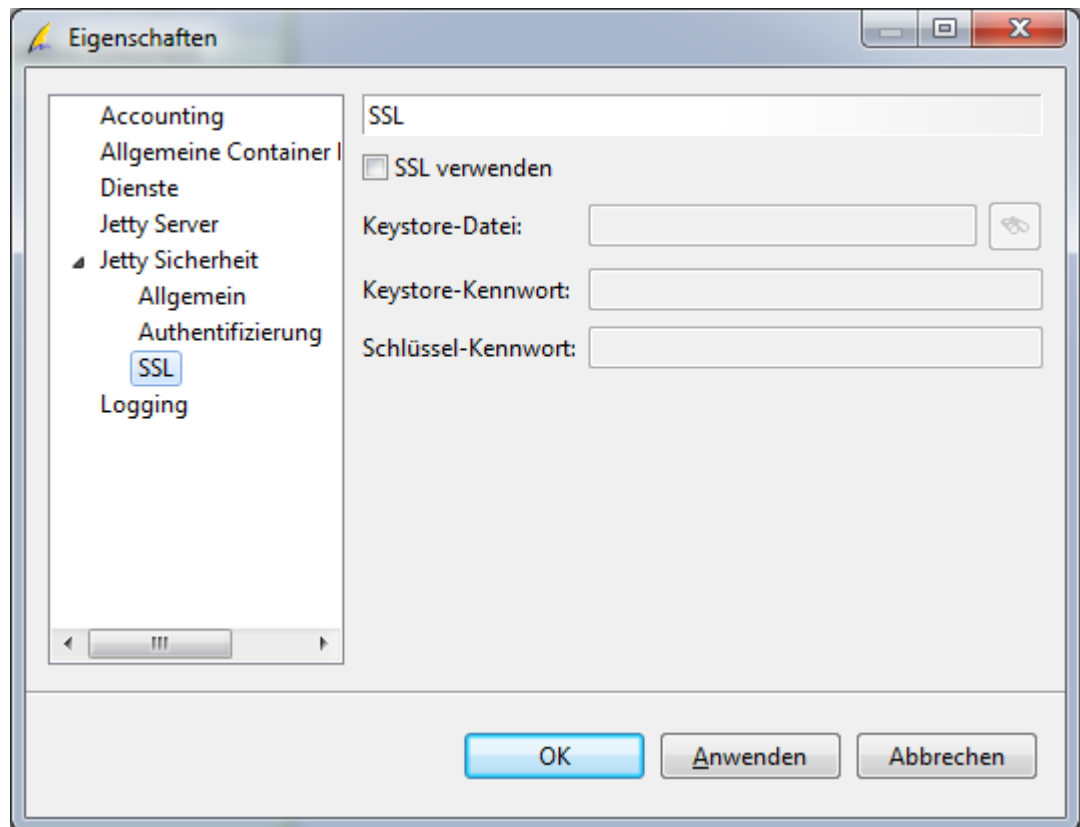
"Only local connections" is a very important, security sensitive setting. If disabled, the remote device will be available from external network devices. If enabled, only processes running on your local machine can access the remote device.

On the “Authentication” page you can set up your container to use standard HTTP authentication techniques.



Currently, only “none” or “basic authentication” is supported. With basic authentication you can define the list of authorized users.

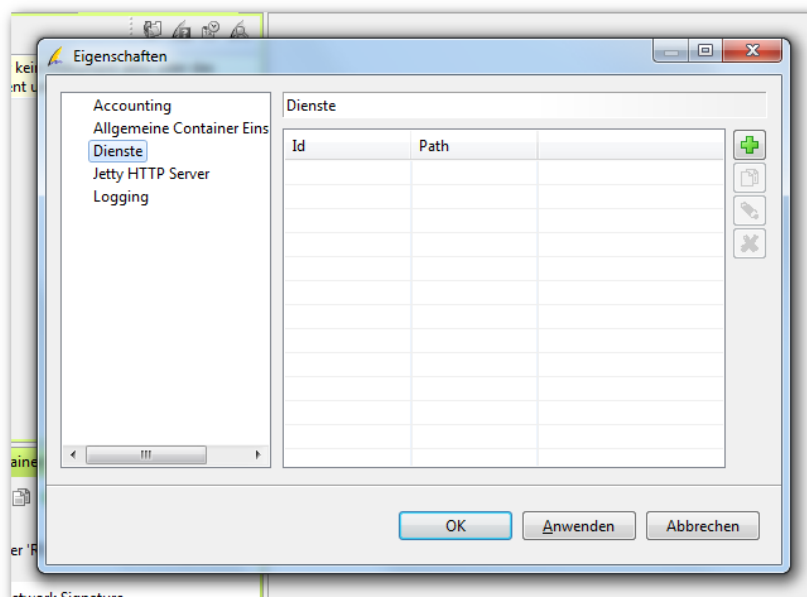
The “SSL” page allows you to configure SSL security for the connections.



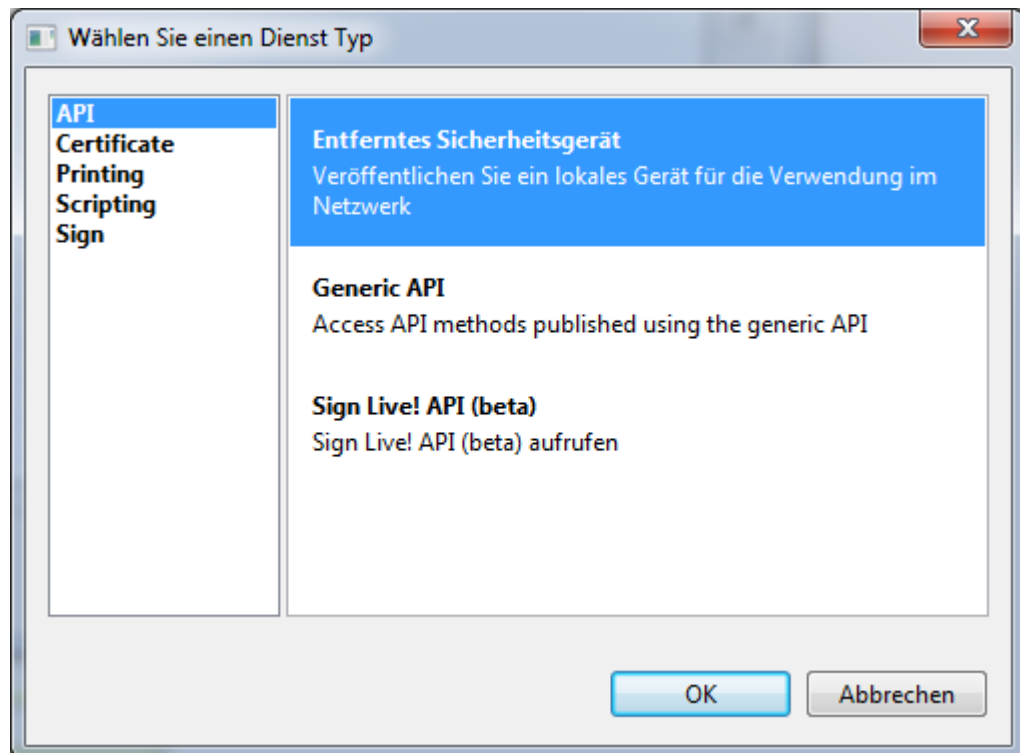
If you select "Use SSL", you can select a Java keystore file, enter the keystore password and the password for the private key itself.

You can create such keystores from the certificate store if you want to.

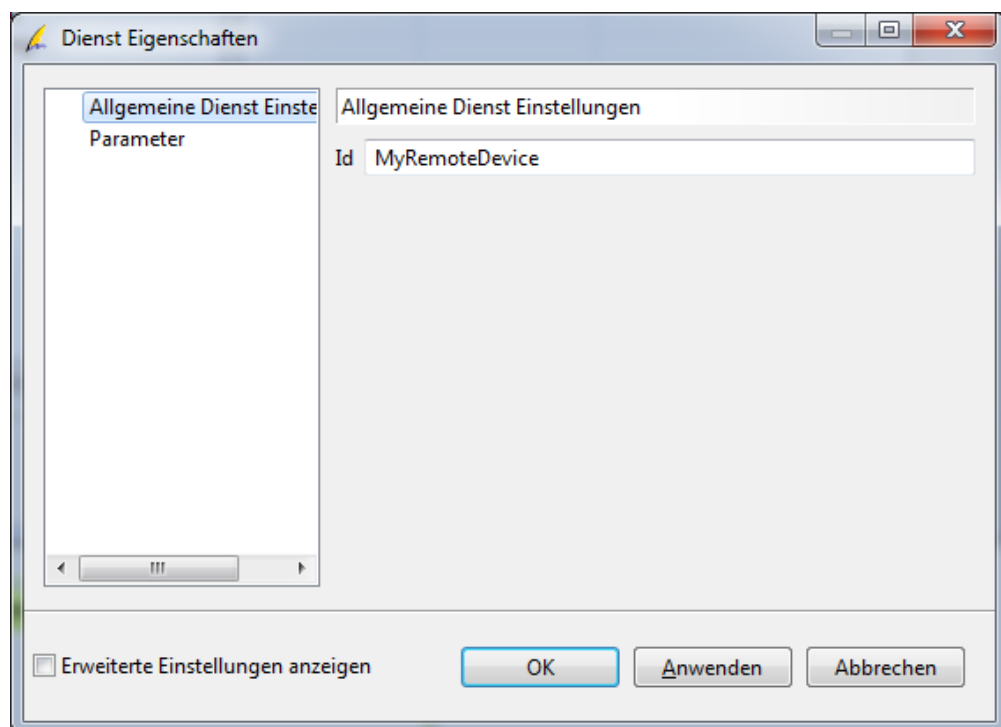
Now we add the signature service. On the "Service" Tab, select the plus icon.



The next dialog shows up some predefined services. Select "RemoteDevice" from the "API" category.



Now we're almost done. You can select a meaningful name and commit all dialog pages with "OK".



1.4.6.3 Security considerations

If you use a remote device, you must check these security considerations before use:

The client running Sign Live! CC must be a clean platform, checked by some active, up-to-date virus protection software.

The network service should be as constrained as possible. Accept local connections only whenever possible.

If used in conjunction with sessions, the session settings should be as restrictive as possible to avoid session hijacking from an intruder. You will find more information on this with the session chapter.

Here's a short summary

- Use a long, random string as session id.
- The session must have some useful timeout (e.g. the default 5 minute timeout)
- The authentication prompt should be very descriptive to enable the user to check if the request is originated by the correct client. The user should check the prompt when entering the PIN.

1.4.7 cloud suite gears signing device

1.4.7.1 Overview

The cloud suite gears signing device will use a remote cloud suite gears server for signing.

PROCESSOR FACTORY

de.intarsys.security.device.gears.processor.GearsDigestSignerFactory

ARGUMENTS

Name	Description
device	<p>Particularization of the device. If this parameter is not given or specified as "gears", the server will be called without additional parameters.</p> <p>Otherwise the parameter has to be specified as "<presets>@gears", where "presets" is the name of a preconfigured set of parameters. Use the "signIT gears" settings page in the application UI to configure these. See the cloud suite gears documentation for information on how to create a signer configuration and for supported signer options and signer args.</p>
url	<p>The URL of the cloud suite gears core web application. If this parameter is not given, the URL from the user preferences will be used. Use the "signIT gears" settings page in the application UI to configure the default URL.</p>

RESULT

The processor returns an object of type `byte[]`.

EXCEPTIONS

The processing may raise exceptions.

1.4.7.2 Example

You will not call a digest signer directly, so here no example is given. A digest signer is used as a parameter to a signer method.

1.5 The Method

1.5.1 The signing method

1.5.1.1 Overview

Currently these signing methods are supported:

- PDF internal signature
- PKCS#7 / CMS signature
- XML DSig (<http://www.w3.org/TR/xmlsig-core/>) (since Version 4.1)

The signing method is the central object, selected using the appropriate processor factory. All other components (document, digester, digestSigner) are provided as arguments to this processor.

A signing method can be selected using a well known processor factory or the generic factory, that itself selects a method based on the document type provided.

1.5.2 Generic signing method

1.5.2.1 Overview

The generic signing method delegates to the concrete signing method appropriate for your system. “Appropriate” means it will select the method that fits the document and is compatible to the current preferences and other customizations active in the platform.

PROCESSOR FACTORY

`com.cabaret.security.document.signing.DocumentSignerFactory`

ARGUMENTS

All arguments are forwarded to the selected concrete signing method.

RESULT

The processor returns an object of type
de.intarsys.security.digsig.ISignatureContainer

EXCEPTIONS

The processing may raise exceptions.

1.5.2.2 Examples

```
//  
var idoc = ..read document from somewhere...  
//  
Processor.callArgs(  
    "com.cabaret.security.document.signing.DocumentSignerFactory",  
    {  
        document: idoc,  
        digester: "SHA256",  
        digestSigner: {  
            factory:  
"com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",  
            args: {  
                signerIdentifier: 'SerialNumber:8139571262270123122;',  
                signerPassword: 'password'  
            }  
        }  
    }  
);
```

This example creates a signature, using a local identity. The signature created depends on the document type and the platform configuration.

1.5.3 PDF signing method

1.5.3.1 Overview

The PDF signing method will accept only PDF documents and creates an internal signature, compliant to the Adobe PDF standard. The signature can be read and verified in any Adobe PDF standard compliant viewer.

The processor can create both visible and invisible signature field, as well as fill in predefined signature fields for an existing form.

PROCESSOR FACTORY

com.cabaret.security.method.pdf.signing.PDFDocumentSignerFactory

ARGUMENTS

Name	Description
document	The document to be signed. For this processor this has to be a PDF document
digester	<p>A digest algorithm or an IDigester instance. See “The Digester”, page 18.</p> <p>The usage of this parameter is optional and if it is omitted, the algorithm set up in your application settings or the strongest algorithm supported by the digest signer device is used.</p>

digestSigner	An instance specification of a <i>signing device</i>
format	<p>(optional) The signature format / standard to comply to. Valid values are:</p> <p>“PAdES Basic” (default)</p> <p>Create a basic PDF signature conforming to the requirements defined in PAdES-2.</p> <p>“PAdES Enhanced”</p> <p>Create a BES or EPES signature conforming to the requirements defined in PAdES-3.</p>
signatureLabel	<p>The optional signature label for the appearance of the signature in the PDF document.</p> <p>The default depends on the digest signer.</p>
locator	<p>The optional locator to store the signature file or signed document.</p> <p>By default the locator of the input document is used.</p>
certificationSignature	<p>This flag controls the type of signature to be created. If set to true, a certification signature is created. Otherwise, a ‘normal’ approval signature is created.</p> <p>The default value is <i>false</i>.</p>
permissions	<p>A string identifying the permissible changes which may be applied to the document after creation of a certification signature. Valid values are:</p> <p>none</p> <p>No further changes allowed.</p> <p>formFilling</p> <p>Allow changes in form values.</p> <p>formFillingAndComments</p> <p>Allow changes in form values and modification of markup annotations.</p>

This argument is only used in conjunction with 'certificateSignature' set to **true**.

lock	Protect a set of fields against post-signature modification even if form-filling is allowed. The set is determined through application of a filter to the document's form fields, which is optionally parameterized by passing in an explicit list of field names.
------	--

filter	<p>A string identifying the filter to apply to the document's form fields.</p> <p>All</p> <p>Include all form fields contained in the document.</p> <p>Include</p> <p>Select only those form fields which are listed in the <i>fields</i> argument.</p> <p>Exclude</p> <p>Include all form fields contained in the document, excluding those which are listed in the <i>fields</i> argument.</p>
--------	---

fields	The list of field names which is interpreted in the context of the <i>filter</i> argument.
--------	--

field	The definition for a signature field to be used. You can use an existing field, create a new invisible or a new visible field. The default is to create a new invisible field.
-------	--

adjustForRotate	<p>true if you want the application to recompute the position and size of the field on rotated pages so they appear as they would on a non-rotated page.</p> <p>The default is "false".</p>
-----------------	---

create	true if you want to create a new field
--------	---

font	The definition for a font to be used for text in the new signature field.
------	---

fontName	Name of the font to use.
----------	--------------------------

name	The name for the signature field. The name has to be a unique name within the documents acro form.
position	<p>The position of the field in the page, defined in user space coordinates. Coordinate values are numbers, separated by "*", "x" or "@".</p> <p>The default is "0*0".</p>
size	<p>The size of the field, defined in user space units. Coordinate values are numbers, separated by "*", "x" or "@".</p> <p>Default size is "0*0", creating an invisible field.</p>
rotate	<p>An angle by which the new field's contents should be rotated. The field's coordinates will not be changed by the rotate operation. Valid values are 0, 90, 180, 270.</p> <p>The default is "0".</p>
pageRange	<p>The page or range of pages where the signature field will be applied. Valid options are</p> <p>all</p> <p>Create a field on all pages</p> <p>first</p> <p>Only on the first page</p> <p>last</p> <p>Only on the last page</p> <p><i>custom page range definition</i></p> <p>You can define a custom page range with a ";" separated list of zero based numbers or number intervals. An interval is defined as two numbers, separated by "-".</p> <p>Examples:</p> <p>"42" → page 42</p> <p>"2;4" → pages 2 and 4</p> <p>"2;5-10" → pages 2 and 5 to 10</p>

hAlign	<p>The position property is interpreted relative to the page media box. Using hAlign you can move the box around the page horizontally without knowing the exact size.</p> <p>left</p> <p>X coordinate is relative to the left page border. Default</p> <p>right</p> <p>X coordinate is relative to the right page border. Field is offset by (x + width).</p> <p>center</p> <p>X coordinate is relative to the page center. Field is offset by (x + width/2).</p>
vAlign	<p>The position property is interpreted relative to the page media box. Using vAlign you can move the box around the page vertically without knowing the exact size.</p> <p>bottom</p> <p>Y coordinate is relative to the bottom page border. Default</p> <p>top</p> <p>Y coordinate is relative to the top page border. Field is offset by (y + height).</p> <p>center</p> <p>Y coordinate is relative to the page center. Field is offset by (y + height/2).</p>
timestampServiceName	<p>The logical name of a registered timestamp service.</p> <p>You can register timestamp services in the preferences. In the result signature a timestamp attribute is included.</p>
timestampDevice	<p>The timestamp device used to create a signature timestamp which will be added as a signature attribute. See chapter 1.7 for information on the argument's syntax.</p>
includeOcsp	<p>Flag if an OCSP check should be done for the signers identity. The result will be included in the signature. This</p>

will ease the task of validation for the signers certificate.

An OCSP request is supported only for certificates where the issuer provides an OCSP service and publishes it in her certificates.

additionalInfoSet	(optional) Additional information about the signer and the signature. (Mind the uppercase names!)
Name	(optional) the signer's name
ContactInfo	(optional) the signer's contact info, like an e-mail address
Location	(optional) the signer's location (e.g.: city, country) at the time of signing
Reason	(optional) the reason for signing the document
signaturePolicy	<p>Defines a signature policy to be associated with the signature. See chapter 1.8.2 for information on the argument's syntax.</p> <p>This argument is interpreted only if format 'PAdES Enhanced' is used and will be ignored otherwise.</p>
commitmentType	<p>Defines a commitment type to be associated with the signature. See chapter 1.8.3 for information on the argument's syntax.</p> <p>This argument is interpreted only if format 'PAdES Enhanced' is used and will be ignored otherwise.</p>
decorator	<p>The object that handles the appearance of a visible PDF field. Supported values are</p> <p>* <empty></p> <p>This will use the default appearance creation strategy. Only text according to the "signatureLabel" is included in the appearance.</p> <p>* An instance specification.</p>
-.factory	<p>One of</p> <p>-</p> <p>com.cabaret.security.method.pdf.signing.decorator.Extend</p>

edDecoratorFactory

Create a combination of image and text.

-args	The supported arguments for the <code>com.cabaret.security.method.pdf.signing.decorator.ExtendedDecoratorFactory</code> are:
text	The literal text to be displayed. The text is expanded before use.
textScaleWhen	<p>How to scale the text within the field. This is one of</p> <p>always</p> <p>Always scale text to the field size</p> <p>never</p> <p>Never scale text to field size</p> <p>toobig</p> <p>Scale text down if too large for the field</p> <p>toosmall</p> <p>Scale text up when too small for the field</p>
textScaleProportional	Flag if the scaling is performed proportional, true or false .
textVAlign	<p>How to align the text within the field vertically.</p> <p>bottom</p> <p>Move text to the bottom.</p> <p>center</p> <p>Move text to the center</p> <p>top</p> <p>Move text to the top</p>
textHAlign	<p>How to align the text horizontally</p> <p>left</p>

	<p>Move text to the left.</p> <p>center</p> <p>Move text to the center.</p> <p>right</p> <p>Move text to the right</p>
icon	A locator to the icon to be displayed in the field. In the simplest case this is the name of an image file that will be included.
iconScaleWhen	<p>How to scale the image within the field. This is one of</p> <p>always</p> <p>Always scale image to the field size</p> <p>never</p> <p>Never scale image to field size</p> <p>toobig</p> <p>Scale image down if too large for the field</p> <p>toosmall</p> <p>Scale image up when too small for the field</p>
iconScaleProportional	Flag if the scaling is performed proportional, true or false .
iconVAlign	<p>How to align the image within the field vertically.</p> <p>bottom</p> <p>Move image to the bottom.</p> <p>center</p> <p>Move image to the center</p> <p>top</p> <p>Move image to the top</p>
iconHAlign	How to align the image horizontally

	<p>left</p> <p>Move image to the left.</p> <p>center</p> <p>Move image to the center.</p> <p>right</p> <p>Move image to the right</p>
layout	<p>How to merge the text and image section of the appearance</p> <p>overlay</p> <p>Show the text as an overly over the icon.</p> <p>textAboveIcon</p> <p>Show the text above the icon.</p> <p>textBelowIcon</p> <p>Show the text below the icon</p> <p>textLeftOfIcon</p> <p>Show the text left of the icon</p> <p>textRightOfIcon</p> <p>Show the text right of the icon</p>
embedFonts	<p>true if fonts and color spaces which are used during signature appearance generation shall be embedded into the document, false otherwise. Defaults to true.</p>

RESULT

The processor returns an object of type *de.intarsys.security.digsig.ISignatureContainer*

EXCEPTIONS

The processing may raise exceptions.

1.5.3.2 Examples

```
//
var idoc = ..read document from somewhere...
//
Processor.callArgs(
  "com.cabaret.security.method.pdf.signing.PDFDocumentSignerFactory",
  {
    document: idoc,
    digester: "SHA256",
    digestSigner: {
      factory:
"com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",
      args: {
        signerIdentifier: 'SerialNumber:8139571262270123122;',
        signerPassword: 'password',
      }
    },
    signatureLabel: 'Signed by ${digestsigner.subject.CN}'
  }
);
```

This example creates a PDF internal invisible signature, using a local identity.

```
//
var idoc = ..read document from somewhere...
//
Processor.callArgs(
  "com.cabaret.security.method.pdf.signing.PDFDocumentSignerFactory",
  {
    document: idoc,
    digester: "SHA256",
    digestSigner: {
      factory:
"com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",
      args: {
        signerIdentifier: 'SerialNumber:8139571262270123122;',
        signerPassword: 'password',
      }
    },
    field: {
      create: true,
      name: "sigfield1",
      position: "100*100",
      size: "200*80",
      pageRange: "all"
    }
  }
);
```

This example creates a visible signature in field “sigfield1” on all pages of the document, using a local identity.

```

Processor.callArgs(
    "com.cabaret.security.document.signing.DocumentSignerFactory",
    {
        document: idoc,
        format: "PAdES Enhanced", // "PAdES Basic" | "PAdES Enhanced"
        field: {
            create: true,
            position: "10@10", // position in PDF user space
            size: "100@100", // size in PDF user space
            pageRange: "first"; // "all" | "first" | "last" | "<page number>"
            valign: "top", // "top" | "center" | "bottom"
            halign: "right", // "right" | "center" | "left"
        },
        decorator: {
            factory:
                "com.cabaret.security.method.pdf.signing.decorator.ExtendedDecoratorFactory",
            args: {
                icon: "c:\\tmp\\some file.png",
                iconScaleWhen: "toobig", // "always" | "never" | "toobig" | "toosmall"
                iconScaleProportional: true, // true | false
                iconVAlign: "top", // "top" | "center" | "bottom"
                iconHAlign: "left", // "left" | "center" | "right"
                text: "any string, expansion supported",
                textScaleWhen: "toobig", // "always" | "never" | "toobig" | "toosmall"
                textScaleProportional: true, // true | false
                textVAlign: "top", // "top" | "center" | "bottom"
                textHAlign: "left", // "left" | "center" | "right"
                layout: "", // "overlay" | "textAboveIcon" | "textBelowIcon" |
                "textLeftOfIcon" | "textRightOfIcon"
            }
        },
        digester: "SHA256",
        digestSigner: {
            factory:
                "com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",
            args: {
                signerIdentifier: 'SerialNumber:8139571262270123122;',
                signerPassword: 'password',
            }
        }
    }
);

```

This is a nearly complete showcase for the PDF related arguments.

1.5.4 PKCS#7 signing method

1.5.4.1 Overview

The PKCS#7 signing method will accept any document and create an external signature, compliant to the PKCS#7 standard. Optionally the signature can contain the original data.

PROCESSOR FACTORY

com.cabaret.security.method.pkcs7.signing.PKCS7DocumentSignerFactory

ARGUMENTS

Name	Description
document	The document to be signed

digester	<p>A digest algorithm or an IDigester instance. See “The Digester”, page 18.</p> <p>The usage of this parameter is optional and if it is omitted, the algorithm set up in your application settings or the strongest algorithm supported by the digest signer device is used.</p>
digestSigner	An instance specification of a <i>signing device</i>
-.factory	
-.args	Arguments forwarded to a <i>signing device</i> factory
format	<p>(optional) The signature format / standard to comply to. Valid values are:</p> <p>CMS (default)</p> <p>Create a basic PDF signature conforming to CMS / PKCS#7.</p> <p>CAdES</p> <p>Create a BES or EPES signature conforming to the requirements defined in CAdES.</p>
locator	<p>The optional locator to the signature file or signed document.</p> <p>By default the locator of the input document is used and the suffix “.p7s” is added.</p>
timestampServiceName	The logical name of a registered timestamp service. You can register timestamp services in the preferences. In the result signature a timestamp attribute is included
timestampDevice	The timestamp device used to create a signature timestamp which will be added as a signature attribute. See chapter 1.7 for information on the argument’s syntax.
signaturePolicy	Defines a signature policy to be associated with the signature. See chapter 1.8.2 for information on the argument’s syntax.

This argument is interpreted only if format 'CADES' is used and will be ignored otherwise.

commitmentType	Defines a commitment type to be associated with the signature. See chapter 1.8.3 for information on the argument's syntax.
----------------	--

This argument is interpreted only if format 'CADES' is used and will be ignored otherwise.

embedDocument	Flag if the original document should be embedded in the resulting PKCS7 signature file.
---------------	---

Default is *false*.

append	Flag if existing signatures should be maintained in the PKCS7 signature file. Only used if the chosen PKCS7 signature file already exists.
--------	--

Default is *true*.

overwrite	Flag if existing PKCS7 signature file should be overwritten. Only interpreted if the chosen PKCS7 signature file already exists and append is <i>false</i> .
-----------	--

Default is *false*.

RESULT

The processor returns an object of type *de.intarsys.security.digsig.ISignatureContainer*

EXCEPTIONS

The processing may raise exceptions.

1.5.4.2 Examples

```
//
var idoc = ..read document from somewhere...
//
Processor.callArgs(
  "com.cabaret.security.method.pkcs7.signing.PKCS7DocumentSignerFactory",
  {
    document: idoc,
    digester: "SHA256",
    digestSigner: {
      factory:
"com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",
      args: {
        signerIdentifier: 'SerialNumber:8139571262270123122;',
        signerPassword: 'password',
      }
    }
  }
);
```

This example creates a PKCS#7 signature, using a local identity.

1.5.5 Internal XML Signature

PROCESSOR FACTORY

com.cabaret.security.method.xml.signing.XMLDocumentInternalSignerFactory

ARGUMENTS

Name	Description
document	The document to be signed. Must be of type "com.cabaret.document.xml.XMLDocumentType", otherwise an exception is thrown.
digester	A digest algorithm or an IDigester instance. See "The Digester", page 18. The usage of this parameter is optional and if it is omitted, the algorithm set up in your application settings or the strongest algorithm supported by the digest signer device is used.
digestSigner	An instance specification of a <i>signing device</i>
-factory	
-args	Arguments forwarded to a <i>signing device</i> factory
format	(optional) The signature format / standard to comply to.

Valid values are:

XAdES (default)

Create a BES or EPES signature conforming to XMLDSig 1.1, considering the requirements defined in XAdES.

XAdES/RFC4050

Create a BES or EPES signature conforming to XMLDSig 1.0, considering the requirements defined in XAdES. Elliptic Curve Cryptography is supported as defined in RFC 4050.

signaturePolicy	Defines a signature policy to be associated with the signature. See chapter 1.8.2 for information on the argument's syntax.
-----------------	---

commitmentType	Defines a commitment type to be associated with the signature. See chapter 1.8.3 for information on the argument's syntax.s
----------------	---

parentNodeLookup	<p>This parameter contains a XPath expression to lookup the signature's parent node inside the document. If the XPath expression uses namespace prefixes, then the parameter namespaces must be used to resolve any prefix to their corresponding namespace.</p> <p>If the XPath expression selects a set of nodes instead a single one, then the first node from the set will be used as parent node. The new signature node will always be appended as the last child of the parent node.</p> <p>If the expression fails to lookup a parent node, the parameter "parentNodeCreate" is used to create the missing parent node, if the parameter was specified, otherwise the operations fails.</p>
------------------	---

parentNodeCreate	<p>This parameter contains a full qualified path separated by "/" to the signature's parent node. The parent node will be created if it doesn't exist.</p> <p>If the path contains namespace prefixes, use the namespaces parameter to resolve any prefix to the corresponding namespace.</p> <p>If this parameter is used in conjunction with the parentNodeLookup parameter, the parentNodeLookup parameter will be evaluated first and if successful,</p>
------------------	--

parentNodeCreate will not be used at all.

references	<p>This parameter contains a list (array) of reference objects, where each object represents a Reference element according to the XML DSig specification. Each object contains following attributes:</p> <p>uri The uri attribute contains a URI referencing the resource to be signed. An empty or missing uri attribute is translated to the documents root element. This applies only in the case of an internal XML signature.</p> <p>transforms The transforms attribute contains a list (array) of Transformation objects, which are applied in order of occurrence on the referenced resource. The result of the last transformation is then signed. Transformation objects are described in the following chapter Transformations.</p>
namespaces	<p>A map containing namespace prefix to namespace mappings. It must contain all namespace prefixes used in any XPath expression above.</p>
locator	<p>The optional locator to store the XML file after signing.</p> <p>By default the locator of the input document is used.</p>

RESULT

The processor returns an object of type *de.intarsys.security.digsig.ISignatureContainer*

EXCEPTIONS

The processing may raise exceptions.

1.5.5.1 Examples

```
//
var xmldoc = ..read document from somewhere...
//
var namespaceMap = {
    msg : "uri:BMU_Waste_Interface/Message",
    lib : "uri:BMU_Waste_Interface/Bibliothek",
    en : "uri:BMU_Waste_Interface/EN",
    dsig: "http://www.w3.org/2000/09/xmldsig#"
}

var referencel = {
    uri: "", // empty uri -> sign document itself
    transforms: [
        {
            algorithm: "http://www.w3.org/2002/06/xmldsig-filter2",
            filter: "intersect",
            expression:
"/descendant::*[name()='en:ENSNERZLayer'][@lib:ATBRolle='ENT']"
        },
        {
            algorithm: "http://www.w3.org/TR/1999/REC-xpath-19991116",
            expression: "not(ancestor-or-self::dsig:Signature)"
        },
        {
            algorithm: "http://www.w3.org/2002/06/xmldsig-filter2",
            filter: "subtract",
            expression: "descendant-or-self::text()[string(normalize-space(descendant-
or-self::text()))='']"
        },
        {
            algorithm: "http://www.w3.org/2001/10/xml-exc-c14n#"
        }
    ]
};

// internal XML signature
var xmlSignatureContainer = Processor.callArgs(
    "com.cabaret.security.method.xml.signing.XMLDocumentInternalSignerFactory",
    {
        document: xmldoc,
        digestSigner: {
            factory:
"de.intarsys.stage.security.device.smartcard.signature.SmartcardDigestSignerFactory",
            args: {
                signerIdentifier: '' //will use the first certificate found on any smartcard
            }
        },
        namespaces: namespaceMap,
        // ds:signature node will be placed at the location specified here in xpath
        parentNodeLookup: "//en:ENSNERZLayer",

        // specify what to sign as an array of maps
        references: [ referencel ]
    }
);
```

This example creates a XML internal signature, using any qualified identity found on a smartcard.

1.5.6 External XML Signature

PROCESSOR FACTORY

com.cabaret.security.method.xml.signing.XMLDocumentExternalSignerFactory

ARGUMENTS

Name	Description
document	The document to be signed. Can be of any type.
digester	<p>A digest algorithm or an IDigester instance. See “The Digester”, page 18.</p> <p>The usage of this parameter is optional and if it is omitted, the algorithm set up in your application settings or the strongest algorithm supported by the digest signer device is used.</p>
digestSigner	An instance specification of a <i>signing device</i>
-.factory	
-.args	Arguments forwarded to a <i>signing device</i> factory
format	<p>(optional) The signature format / standard to comply to. Valid values are:</p> <p>XAdES (default)</p> <p>Create a BES or EPES signature conforming to XMLDSig 1.1, considering the requirements defined in XAdES.</p> <p>XAdES/RFC4050</p> <p>Create a BES or EPES signature conforming to XMLDSig 1.0, considering the requirements defined in XAdES. Elliptic Curve Cryptography is supported as defined in RFC 4050.</p>
signaturePolicy	Defines a signature policy to be associated with the signature. See chapter 1.8.2 for information on the argument’s syntax.
commitmentType	Defines a commitment type to be associated with the signature. See chapter 1.8.3 for information on the argument’s syntax.
locator	The locator to store the XML signature to. By default the path of the locator of the input document is used and the

suffix “.xml” is added. If the locator points to an existing XML file and the parameter append is set to true, then the existing file will be reused and the new signature is added.

append	Flag if new XML signatures should appended to existing XML files. Default is true.
parentNodeLookup	This parameter contains a XPath expression to lookup the signature’s parent node inside an existing XML document pointed to by the parameter locator. If the XPath expression uses namespace prefixes, then the parameter namespaces must be used to resolve any prefix to their corresponding namespace. If the XPath expression selects a set of nodes instead a single one, then the first node from the set will be used as parent node. The new signature node will always be appended as the last child of the parent node. If the expression fails to lookup a parent node, the parameter “parentNodeCreate” is used to create the missing parent node, if the parameter was specified, otherwise the operations fails.
parentNodeCreate	This parameter contains a full qualified path separated by “/” to the signature’s parent node. The parent node will be created if it doesn’t exist. If the path contains namespace prefixes, use the namespaces parameter to resolve any prefix to the corresponding namespace. If this parameter is used in conjunction with the parentNodeLookup parameter, the parentNodeLookup parameter will be evaluated first and if successful, parentNodeCreate will not be used at all.
embedDocument	Flag if the original document should be embedded in the XML signature file. Default is false .
namespaces	Namespaces is a map containing namespace prefix to namespace mappings. It must contain all namespace prefixes used in any XPath expression above.

RESULT

The processor returns an object of type
de.intarsys.security.digsig.ISignatureContainer

EXCEPTIONS

The processing may raise exceptions.

1.5.6.1 Examples

```
//
var idoc = ..read document from somewhere...

// external XML signature
var xmlSignatureContainer = Processor.callArgs(
  "com.cabaret.security.method.xml.signing.XMLDocumentInternalSignerFactory",
  {
    document: idoc,
    digestSigner: {
      factory:
"de.intarsys.stage.security.device.smartcard.signature.SmartcardDigestSignerFactory",
      args: {
        signerIdentifier: '' //will use the first certificate found on any smartcard
      }
    },
    locator: "c:/temp/embeddedSignature.xml",
    // ds:signature node will be placed as a child to the parent node specified here in
xpath
    parentNodeCreate: "/EmbeddedDocRoot",
    embedDocument=true;
  }
);
```

This example creates a external XML signature for a document and the embeds the signed document inside the XML file.

1.5.6.2 Transformations

A transformation object contains three attributes:

algorithm

a unique algorithm id in form of a URI

filter

An optional filter expression, required by some algorithms

expression

An actual transformation expression, like a XPath expression

The following transformations are supported:

1.5.6.2.1 XMLDSig Enveloped Signature

algorithm

“http://www.w3.org/2000/09/xmlsig#enveloped-signature”

filter

not used

expression

not used

1.5.6.2.2 XMLDSig XPath

algorithm

- “http://www.w3.org/TR/1999/REC-xpath-19991116”
 filter
 not used
 expression
 a XPath expression
- 1.5.6.2.3 XMLDSig XPath2
 algorithm
 “http://www.w3.org/2002/06/xmlsig-filter2”
 filter
 “intersect”, “subtract” or “union”. The default value for filter is
 “intersect”.
 expression
 a XPath expression
- 1.5.6.2.4 XMLDSig Base64
 algorithm
 “http://www.w3.org/2000/09/xmlsig#base64”
 filter
 not used
 expression
 not used
- 1.5.6.2.5 Canonical XML without comments
 algorithm
 “http://www.w3.org/TR/2001/REC-xml-c14n-20010315”
 filter
 not used
 expression
 not used
- 1.5.6.2.6 Canonical XML with comments
 algorithm
 “http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments”
 filter
 not used
 expression
 not used

- 1.5.6.2.7 Exclusive Canonical XML without comments
algorithm
"http://www.w3.org/2001/10/xml-exc-c14n#"
filter
not used
expression
not used
- 1.5.6.2.8 Exclusive Canonical XML with comments
algorithm
"http://www.w3.org/2001/10/xml-exc-c14n#WithComments"
filter
not used
expression
not used

1.6 The Wizard

1.6.1 The signing wizard

While you can call the signature processors directly as seen in the chapters before, you can also bring up a wizard to collect all the arguments you must supply. By default the wizard will call the signature processor after user interaction with the arguments entered.

You can preset all or part of the arguments for the signature process and the wizard will apply these as defaults.

WIZARD FACTORY

com.cabaret.security.document.signing.ui.DocumentSignerWizardFactory

ARGUMENTS

All arguments that are available with the signature processors can be preselected.

Name	Description
document	The document to be signed must be supplied.
documentSigner	A complete or partial set of arguments for the signature method to be applied

RESULT

The result of the wizard is the result of the signature processor selected and executed.

1.6.2 Examples

```
var idoc = ...get document from somewhere..
var signature = Wizard.callArgs(
    'com.cabaret.security.document.signing.ui.DocumentSignerWizardFactory',
    {
        document: idoc
    }
);
```

This calls the signature wizard. All arguments are preset to the values stored in the preferences from the last call (if applicable to the current document type).

1.7 Timestamp Devices

1.7.1 Overview

The timestamp device is implemented using the *de.intarsys.security.device.IDevice* abstraction. It is typically defined using a device factory and a set of arguments passed for device creation.

Currently these timestamp devices are supported:

- Timestamp device lookup
- HTTP timestamp device for access to RFC3161-compliant services

1.7.2 Timestamp device lookup

1.7.2.1 Overview

The device lookup method will search for a registered timestamping device and returns it for timestamp creation requests. In most cases, the device is configured by the user, entering the connection data of an official timestamp service.

DEVICE FACTORY

`de.intarsys.security.device.timestamp.device.TimestampDeviceProvider`

ARGUMENTS

Name	Description
id	The id of a registered timestamp device.

RESULT

The factory returns an object of type *de.intarsys.security.device.IDevice*.

EXCEPTIONS

The processing may raise exceptions.

1.7.2.2 Examples

```
//
var idoc = ..read document from somewhere...
//
Processor.callArgs(
    "com.cabaret.security.method.pkcs7.signing.PKCS7DocumentSignerFactory",
    {
        document: idoc,
        digester: "SHA256",
        digestSigner: {
            factory:
"com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",
            args: {
                signerIdentifier: 'SerialNumber:8139571262270123122;',
                signerPassword: 'password',
            }
        },
        timestampDevice: {
            factory: 'de.intarsys.security.device.timestamp.device.TimestampDeviceProvider',
            args: {
                id: 'MyTSA'
            }
        }
    }
);
```

This example creates a PKCS#7 signature, using a local identity. A signature timestamp will be created by a timestamping device registered with id 'MyTSA'.

1.7.3 HTTP Timestamp Device

1.7.3.1 Overview

The http timestamp device will connect to an RFC3161-compliant timestamp service which can be accessed using HTTP.

DEVICE FACTORY

de.intarsys.security.device.httptimestamp.device.HttpTimestampDeviceFactory

ARGUMENTS

Name	Description
url	The timestamp service's URL.
user	(optional) The user name used for HTTP Basic

Authentication.

password

(optional) The password used for HTTP Basic Authentication.

RESULT

The factory returns an object of type *de.intarsys.security.device.IDevice*.

EXCEPTIONS

The processing may raise exceptions.

1.7.3.2 Examples

```
//
var idoc = ..read document from somewhere...
//
Processor.callArgs(
    "com.cabaret.security.method.pkcs7.signing.PKCS7DocumentSignerFactory",
    {
        document: idoc,
        digester: "SHA256",
        digestSigner: {
            factory:
"com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",
            args: {
                signerIdentifier: 'SerialNumber:8139571262270123122;',
                signerPassword: 'password',
            }
        },
        timestampDevice: {
            factory:
'de.intarsys.security.device.httptimestamp.device.HttpTimestampDeviceFactory',
            args: {
                url: 'http://tsa.mycompany.com'
            }
        }
    }
);
```

This example creates a PKCS#7 signature, using a local identity. A signature timestamp will be fetched from 'http://tsa.mycompany.com'.

1.8 Explicit Policy Electronic Signatures (EPES)

1.8.1 Overview

Explicit Policy Electronic Signatures (EPES) require the definition of a signature policy reference. This policy reference is included as a signed attribute into the signature and becomes a major element. In addition, a commitment type can be included, further specifying the signer's relation to the document.

1.8.2 Defining signature policy references

Signature policy references are defined using argument sets. The following arguments are supported

Name	Description
oid	The signature policy's registered oid. Required, if policy is not defined in ASN.1 / DER format.
hash	(optional) The policy's expected hash value.
locator	A locator containing the signature policy.
qualifiers	<p>(optional) The list of qualifiers associated to the signature policy. A qualifier is defined using an oid and qualifier-specific further arguments. The following qualifier types are supported:</p> <ul style="list-style-type: none"> • URI • User Notice <p>The respective type is identified by its "oid" value. The arguments available depend on the qualifier type.</p>
URI type	
oid	The qualifier's oid. The value is always '1.2.840.113549.1.9.16.5.1'.
uri	The URI pointing to a location where the policy can be downloaded.
User Notice type	
oid	The qualifier's oid. The value is always '1.2.840.113549.1.9.16.5.2'.
explicitText	(optional) The notice's text.
noticeReference	(optional) A reference to a notice.

It is defined using the following sub-arguments	
organization	The name of the organization defining the signature policy.
noticeNumbers	A list of organization-unique notice numbers.
implied	A Boolean value used to indicate that the signature policy can be unambiguously deduced from the signed document's content. <i>true</i> if the policy is implicit, <i>false</i> otherwise (default = <i>false</i>). If set to <i>true</i> , all other arguments are ignored.

1.8.3 Defining commitment types

Commitment types are defined using argument sets. The following arguments are supported

Name	Description
oid	The commitment type's registered oid. Optional, if name is set, required otherwise.
name	<p>The name of a standard commitment type. Optional, if oid is set, required otherwise.</p> <p>Valid names are:</p> <p>proofOfApproval</p> <p>The signer has approved the content of the message.</p> <p>proofOfCreation</p> <p>The signer has created the message (but not necessarily approved, nor sent it).</p> <p>proofOfDelivery</p> <p>The signer has delivered a message in a local store accessible to the recipient of the message.</p> <p>proofOfOrigin</p> <p>The signer recognizes to have created, approved, and sent the message.</p>

proofOfReceipt

The signer recognizes to have received the content of the message.

proofOfSender

The signer has sent the message (but not necessarily created it).

1.8.4 Examples

```
//
var idoc = ..read document from somewhere...
//
Processor.callArgs(
  "com.cabaret.security.method.pkcs7.signing.PKCS7DocumentSignerFactory",
  {
    document: idoc,
    digester: "SHA256",
    digestSigner: {
      factory:
"com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",
      args: {
        signerIdentifier: 'SerialNumber:8139571262270123122;',
        signerPassword: 'password',
      }
    },
  },
  format: 'CADES',
  signaturePolicy: {
    locator: '/policies/sigpol.der',
    hash: {
      algorithm: 'SHA256',
      raw: 'BXfcly5o7PhVcV6LBSvmzOGaiKmalgJaM/iK1bwcA1U='
    },
    qualifiers: {
      0: {
        oid: '1.2.840.113549.1.9.16.5.1',
        uri: 'http://www.mycompany.com/policies/sigpol.der'
      },
      1: {
        oid: '1.2.840.113549.1.9.16.5.2',
        explicitText: 'Standard signature policy for e-invoicing.',
        noticeReference: {
          organization: 'MyCompany Inc.',
          noticeNumbers: {
            0: 15,
            1: 20
          }
        }
      }
    }
  },
  commitmentType: {
    name: 'proofOfApproval'
  }
}
);
```

This example creates a CADES signature, using a local identity. A DER-encoded signature policy is applied, its reference being included into the signature. The policy is tested for integrity by comparison to the hash argument passed (a BASE64-encoded SHA256 hash value). The included commitment type OID is set to 1.2.840.113549.1.9.16.5, the registered id for type 'proofOfApproval'.

1.9 Examples

1.9.1 Overview

In this chapter we want to provide some complete examples for interfacing with Sign Live! CC to create and customize signature applications.

First of all, we define our target scenario: We have a document file somewhere around and want it to be signed. Now, good news, we have Sign Live! CC somewhere around and can use it. What we have to do is:

Select the document

Sign it

Be aware that all of the examples here are presented without error handling or ensure safe resource cleanup. In a production environment you should add such stuff (like “try...catch...finally”).

Source code for this examples can be found in the installation subdirectory “demo/security/signing”.

1.9.2 PDF signature on an HTTP uploaded document using a local certificate

Here we will have an example including the upload of a document - an example signing a document “by reference” (via a filename) is shown in another chapter. Feel free to adapt the HTTP example to such a scenario.

For HTTP you have to setup the ULS infrastructure. If you start from an original installation, ULS and the standard HTTP server are already available. Be sure the container is started - you can check this with the ULS preferences pages. More information on this topic you will find in the “*Operator’s Guide*” and “*Developer’s Guide*”.

Your task is to provide another web application using one of the standard servlets to publish your service. For your reference a ready to use web application is provided in the directory “demo/security/signing/instruments”.

Your instrument simply declares a new web application for an existing container:

```
...
<extension point="com.cabaret.uls.containers">
  <container ref="com.cabaret.uls.container.ContainerStandardHTTP">
    <webapps path="${instrument.basedir}/webapps"/>
  </container>
</extension>
...
```

The web application itself declares a servlet forwarding the document upload request to a JavaScript

```

...
    <servlet>
        <servlet-name>sign</servlet-name>
        <servlet-
class>com.cabaret.uls.servlet.application.DocumentUploadServlet</servlet-class>
        <init-param>
            <param-name>serviceFunctor</param-name>
            <param-value>
<![CDATA[
<perform type="Script" source="scripts/sign"/>
]]>
                                </param-value>
            </init-param>
        </servlet>
...

    <servlet-mapping>
        <servlet-name>sign</servlet-name>
        <url-pattern>/perform/*</url-pattern>
    </servlet-mapping>
..

```

The associated JavaScript script file:

```

var idoc = document;
var signature = Processor.callArgs(
    'com.cabaret.security.document.signing.DocumentSignerFactory',
    {
        document: idoc,
        digestSigner: {
            factory:
'com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory',
            args: {
                signerIdentifier: 'SerialNumber:8139571262270123122;',
                signerPassword: 'password'
            }
        }
    }
);

/* don't release idoc - you didn't create it */

/*
 * return signature data,
 * indicate a file download first
 * reply is handled manually to clean up resources
 */
var locator = signature.locator;
response.setHeader(
    'Content-Disposition',
    'attachment; filename="'+ locator.getTypedName()+ '"'
);
_servlet.reply(_request, _response, locator);
/* delete is reserved word!! */
locator['delete']();

```

With the above servlet mapping you can upload a file to

```
http://localhost:8080/httpsigning/perform
```

or

```
http://localhost:8080/httpsigning/perform?tempfile=foo.bar
```

if you want to specify an additional file name.

The document is signed and returned. For your convenience a little template is included in the demo instrument allowing an upload to be triggered from the browser

<http://localhost:8080/httpsigning/console>

will bring you to that page if you installed this example from the demo instrument.

The CodeExit itself is implemented using standard techniques and some of the API's we have seen above. The document is made available as an argument by the DocumentUploadServlet, this document is then signed using the generic signer and the demo certificate from the local key store.

You can see some important little patterns here:

The document is forwarded as an CodeExit argument

The document is under control of the servlet. It has created it and it will release it after service.

You can set the "Content-Disposition" to force a download.

If you want to take actions (cleanup) after replying the result data, you must call "_servlet.reply" yourself and clean up afterwards.

"delete" is a reserved word with JavaScript. So, if you want to call the method "delete", just select it as an object member (which is a function object) and call it.

1.9.3 PDF signature using a local certificate via XMLRPC

For XMLRPC you have to setup the ULS infrastructure. If you start from an original installation, ULS, the standard HTTP server and the XML RPC servlet are already available. All you have to do is check "Autostart" in the preferences page for the respective ULS Container. More information on this topic you will find in the "*Operator's Guide*" and "*Developer's Guide*".

Now you define your object model, more exact the methods to be published with your XML RPC server. Detailed information on this you will find in the "*Developer's Guide*" chapter "XML RPC". Here we assume you are already familiar with the required syntax.

You create an instrument and extend *com.cabaret.claptz.objectmodel.members* where you declare a method named *Document.demo_sign* for the implementor *com.cabaret.uls.servlet.xmlrpc.XmlRpcServer*. This method accepts a parameter *file*, which should reference a valid file.


```
...
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.uls.servlet.xmlrpc.XmlRpcServer"
    name="Document.demo_sign">
    <perform type="Script" source="scripts/sign">
      <declarations>
        <arg name="file"/>
      </declarations>
    </perform>
  </method>
</extension>
...
```

The associated JavaScript script file:

```
var idoc = Processor.callArgs(
  'DocumentLoaderFactory', {
    locator: file
  });
Processor.callArgs(
  'com.cabaret.security.document.signing.DocumentSignerFactory', {
    document: idoc,
    digestSigner: {
      factory:
        'com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory',
      args: {
        signerIdentifier: 'SerialNumber:8139571262270123122;',
        signerPassword: 'password'
      }
    }
  });
/* dont forget to release */
idoc.release();
```

Upon receiving a XMLRPC request on the URL

```
http://localhost:8080/xmlrpc/server
```

to the method *Document.demo_sign*, this object model declaration is selected and executed. The declaration is needed here to “name” the indexed argument from XMLRPC and make the first one available as the “file” argument to the JavaScript.

The method itself is implemented using standard techniques and some of the API’s we have seen above. First, a loader is used to instantiate the document object, this document is then signed using the generic signer and a local key store.

1.9.4 PKCS#7 signature using a smartcard via ActiveX

To use ActiveX you must have installed the ActiveX support upon installation of Sign Live! CC.

To get general information about this API, you can consult “*Developer’s Guide*” chapter “ActiveX”.

As with XML RPC you define your object model. Detailed information on this you will find again in the “*Developer’s Guide*” chapter “ActiveX”. Here we assume you are already familiar with the required syntax.

You create an instrument and extend *com.cabaret.claptz.objectmodel.members* where you declare a method

named *demo_sign* with your document type. For sure you will not need a parameter here, as the document to be signed is the target of your method.

```
...
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.activedoc.CommonActiveDoc"
    name="demo_sign">
    <perform type="Script" source="scripts/sign"/>
  </method>
</extension>
...
```

The associated JavaScript script file.

```
var idoc = jEvent.target.document;
Processor.callArgs(
  'com.cabaret.security.method.pkcs7.signing.PKCS7DocumentSignerFactory',
  {
    document: idoc,
    digestSigner: {
      factory:
'de.intarsys.stage.security.device.smartcard.signature.SmartcardDigestSignerFactory',
      args: {
        signerIdentifier: 'SerialNumber: 89983439847',
        signerPassword: '888888'
      }
    }
  }
);
```

Your client creates an instance of the ActiveX object by loading the respective file. Then you call “demo_sign” without arguments via one of the ActiveX invocation API’s. The object model declaration is selected and executed.

The method itself is implemented using standard techniques and some of the API’s we have seen above. First, the document object is selected from the call event. This document is then signed using the PKCS7 signer and a smartcard device. Don’t forget to plug in a card. Be sure to adress your own certificate and use the correct password!

For details on programming the client, see “*Developer’s Guide*” chapter “ActiveX” and the demo sources deployed with Sign Live! CC.

1.9.5 PKCS#7 signature, Commandline version

To use the commandline, there is nothing you have to prepare, this is simply builtin. We recommend to create a small “cli” file where you script the commandline options and which later on is used to call Sign Live! CC. More information on the commandline you will find in “Operator’s Guide” chapter “Command Line Interface”.

Here is the “sign.cli” file:

```

-silent
-perform -pt JavaScript -ps
"
var idoc = Processor.callArgs(
  'DocumentLoaderFactory', {
    locator: '${options.1}'
  });
Processor.callArgs(
  'com.cabaret.security.document.signing.DocumentSignerFactory', {
    document: idoc,
    digestSigner: {
      factory: 'com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory',
      args: {
        signerIdentifier: 'SerialNumber:8139571262270123122;',
        signerPassword: 'password'
      }
    }
  });
/* dont forget to release */
idoc.release();
"
-popall

```

You can execute this via a command shell

```
start "SignLiveCC" "bin\SignLiveCC.exe" -cf sign.cli "%1"
```

Remember that “-cf” has to be the only option if it is used at all. All other options, including “-config” and so on must be written in the command line file.

The complete command file is expanded before use. The options to “-cf” are forwarded for string expansion and are available under the string variable namespace “options”. So, as in this example, you can access the filename with \${options.1}. \${options.0} is the commandfile itself - this is just as you are used to from plain batch files.

1.9.6 PDF signature for use in Script Center

As you know, there is also an interactive tool with Sign Live! CC to create, manage and run used defined scripts, the Script Center.

You can create a new script “sign.js” in the Script Center and save it somewhere, keeping a link on to the script in the center.

```

var idoc = jEvent.target.document;
Processor.callArgs(
  "com.cabaret.security.method.pdf.signing.PDFDocumentSignerFactory",
  {
    document: idoc,
    digester: "SHA256",
    digestSigner: {
      factory:
        "com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",
      args: {
        signerIdentifier: 'SerialNumber:8139571262270123122;',
        signerPassword: 'password'
      }
    },
    field: {
      create: true,
      position: "100*300",
      size: "200*80",
      pageRange: "first"
    }
  }
);

```

This script will create a signature on an open, active PDF document, when you just double click it in the Script Center.

1.9.7 Signature wizard activation via ActiveX

To use ActiveX you must have installed the ActiveX support upon installation of Sign Live! CC.

To get general information about this API, you can consult “*Developer’s Guide*” chapter “ActiveX”.

As with XML RPC you define your object model. Detailed information on this you will find again in the “*Developer’s Guide*” chapter “ActiveX”.

Here we assume you are already familiar with the required syntax.

You create an instrument and extend *com.cabaret.claptz.objectmodel.members* where you declare a method named *demo_wizard_sign* with your document type.

```

...
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.actedoc.CommonActiveDoc"
    modifiers="gui"
    name="demo_wizard_sign">
    <perform type="Script" source="scripts/wizard_sign"/>
  </method>
</extension>
...

```

The associated JavaScript script file:

```
var idoc = jEvent.target.document;
var signature = Wizard.callArgs(
    'com.cabaret.security.document.signing.ui.DocumentSignerWizardFactory',
    {
        document: idoc
    }
);
/* be aware of cancellation */
if (signature != null) {
    /* return the file name of the signature */
    signature.locator.fullName;
}
```

Your client creates an instance of the ActiveX object by loading the respective file. Then you call “demo_wizard_sign” without arguments via one of the ActiveX invocation API’s. Again, the object model declaration is selected and executed.

For the method you see a special declaration:

```
... modifiers="gui"
...
```

This is necessary because of the execution model for SWT, the GUI library used by Sign Live! CC. All access to GUI elements must be from an appropriate thread. This modifier ensures that execution from the client is delegated to this GUI thread. Now we can access GUI code like the wizard is.

Again, the document object is selected from the call event. This document is forwarded as an argument to the wizard. The wizard will care for the rest.

For details on programming the client, see “*Developer’s Guide*” chapter “ActiveX” and the demo sources deployed with Sign Live! CC.

2. Validation

2.1 Overview

This chapter presents the validation features for Sign Live! CC. You will learn how to validate a document in respect to the following signing methods:

- PDF internal signature
- PKCS#7 signature
- XML signature
- Evidence Record

The basic steps for validation are

- Select a document
- Validate the document
- Return the state information

The security framework provides some abstractions for the steps and objects involved.

Document The document to be validated. The signatures for the document are passed explicitly or looked up automatically dependent on the document type.

Validator The method used for validation. This is the process of looking up signatures for the document and validation of the signatures. The result is a document validation state.

Beyond, the framework offers document-independent validation methods, targeted at certificates.

2.2 The Document

A document is represented using the *de.intarsys.document.model.IDocument* abstraction from the application

platform. More information on this topic you can find in “*Developer’s Guide*” chapter “Working with documents”.

You will find basic support for these document types:

- PDF
- XML
- Text flavors
- Image flavors (BMP, PNG, Tiff, GIF, JPG)
- Generic transparent document (binary data)

2.3 The Method

2.3.1 Generic document validation method

2.3.1.1 Overview

PROCESSOR FACTORY

`com.cabaret.security.document.validation.DocumentValidatorFactory`

ARGUMENTS

Name	Description
document	The document to be validated.
signatureContainers	The signature containers to be used for document validation

RESULT

The processor returns an object of type
`com.cabaret.security.document.validation.IVSSignedDocument`

EXCEPTIONS

The processing may raise exceptions

2.3.1.2 The IVSSignedDocument

The `com.cabaret.security.document.validation.IVSSignedDocument` is the result object for a document validation.

2.3.1.2.1 Public methods

`getState`

Return the computed document signature state as an integer value where


```
public static final int STATE_UNDEFINED = -1;
public static final int STATE_VALID = 0;
public static final int STATE_UNKNOWN = 2;
public static final int STATE_INVALID = 3;
```

The computation for the combined state is currently fixed. It depends upon the settings in your validation preferences (for example, how to deal with qualified certificates).

In a future release you will be able to customize the rules for computing a combined state in a more detailed way.

Additionally, more methods will be published to access detailed state information of the `ISignedDocument`, for example for every signature entry in the document.

2.3.2 Certificate validation method

2.3.2.1 Overview

PROCESSOR FACTORY

`com.cabaret.security.app.validation.CertificateValidatorFactory`

ARGUMENTS

Name	Description
certificate	The certificate to be validated.

RESULT

The processor returns an object of type *de.intarsys.security.validation.IVSCertificate*

EXCEPTIONS

The processing may raise exceptions

2.3.2.2 The IVSCertificate

The *de.intarsys.security.validation.IVSCertificate* is the result object for a certificate validation.

2.3.2.2.1 Public methods getState

Return the computed certificate state as an integer value where

```
public static final int STATE_UNDEFINED = -1;

public static final int STATE_VALID = 0;

public static final int STATE_UNKNOWN = 2;

public static final int STATE_INVALID = 3;
```

The computation for the combined state is currently fixed. It depends upon the settings in your validation preferences (for example, how to deal with qualified certificates).

In a future release you will be able to customize the rules for computing a combined state in a more detailed way.

Additionally, more methods will be published to access detailed state information of the IX509Certificate, for example for its revocation information.

2.4 Examples

2.4.1 Overview

Here you find complete examples for interfacing with Sign Live! CC to create and customize validation applications.

The target scenario: We have a document file somewhere around and want it to be validated. Steps:

- Select the document
- Validate it

Be aware that all of the examples here are presented without error handling or ensure safe resource cleanup. In a production environment you should add such stuff (like “try...catch...finally”).

2.4.2 Validation of an HTTP uploaded document

For HTTP you have to setup the service container infrastructure - this is already mentioned in previous chapters, so we will skip.

Again, for your reference a ready to use service configuration is provided in the directory “sdk/HTTP/demo/Validation/instruments”.

After the well-known web application declaration

```
...
<extension point="com.cabaret.uls.containers">
  <container ref="com.cabaret.uls.container.ContainerStandardHTTP">
    <webapps path="${instrument.basedir}/webapps"/>
  </container>
</extension>
...
```

This is an excerpt from the web.xml. Just define a DocumentUploadServlet and tell it to forward to “scripts/validate.js”, which is found relative to the web application base directory.

```

...
    <servlet>
        <servlet-name>validate</servlet-name>
        <servlet-
class>com.cabaret.uls.servlet.application.DocumentUploadServlet</servlet-class>
        <init-param>
            <param-name>serviceFunctor</param-name>
            <param-value>
<![CDATA[
<perform type="Script" source="scripts/validate"/>
]]>
            </param-value>
        </init-param>
    </servlet>
...

    <servlet-mapping>
        <servlet-name>validate</servlet-name>
        <url-pattern>/perform/*</url-pattern>
    </servlet-mapping>
..

```

For clarity reasons, the JavaScript implementation is extracted from the web.xml and saved in `<web application home>/scripts/validate.js`.

```

var idoc = document;
var state = Processor.callArgs(
    'com.cabaret.security.document.validation.DocumentValidatorFactory',
    {
        document: idoc
    }
);

/* create report */
var report = CodeExit.callArgs(
    "Action",
    "de.intarsys.action.signlive.report", {
        document: state,
        format: "HTML",
        append: false
    }
);

if (report != null) {
    /*
     * return report as continuous stream
     */
    _servlet.reply(_request, _response, report);

    /* don't forget to release report */
    report.release();
}

/* don't release idoc - you didn't create it */

```

This example assumes you want to stream back a validation report in HTML syntax. The validation report referenced in this example may be not available with your installation, depending upon the product you purchased. You can simply replace it with another report or statement of your choice.

With the above servlet mapping you can upload a file to

<http://localhost:8080/httpvalidation/perform>

or

```
http://localhost:8080/httpvalidation/perform?tempfile=foo.bar
```

if you want to specify an additional file name.

The document is validated and the report is returned.

For your convenience a little template is included in the demo instrument allowing an upload to be triggered from the browser

```
http://localhost:8080/httpvalidation/console
```

will bring you to that page if you installed this example from the demo instrument.

The CodeExit itself is implemented using standard techniques and some of the API's we have seen above. The document is made available as an argument by the DocumentUploadServlet, this document is then validated using the common validator. The resulting state is processed by a template that creates a HTML summary of the state information.

2.4.3 Validation triggered via XMLRPC

Setup the ULS and XMLRPC infrastructure as described in previous chapters.

Create an instrument and extend

com.cabaret.claptz.objectmodel.members where you declare a method named *Document.demo_validate* for the implementor *com.cabaret.uls.servlet.xmlrpc.XmlRpcServer*. This method accepts a parameter *file*, which should reference a valid file.

```

...
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.uls.servlet.xmlrpc.XmlRpcServer"
    name="Document.demo_validate">
    <perform type="Script" source="scripts/validate">
      <declarations>
        <arg name="file"/>
      </declarations>
    </perform>
  </method>
</extension>
...
The associated JavaScript script file:
/* load document */
var idoc = Processor.callArgs(
  'DocumentLoaderFactory', {
    locator: file
  }
);
/* validate state */
var state = Processor.callArgs(
  'com.cabaret.security.document.validation.DocumentValidatorFactory',
  {
    document: idoc
  }
);
Packages.com.cabaret.security.app.validation.ValidationTools.showState(state);

/* dont forget to release */
idoc.release();

/* return state integer */
state.getState();

```

Upon receiving a XMLRPC request on the URL

<http://localhost:8080/xmlrpc/server>

to the method *Document.demo_validate*, this object model declaration is selected and executed.

The method itself is implemented using standard techniques and some of the API's we have seen above. First, a loader is used to instantiate the document object, this document is then validated using the generic validator.

2.4.4 Validation triggered via ActiveX

To use ActiveX you must have installed the ActiveX support upon installation of Sign Live! CC.

To get general information about this API, you can consult "*Developer's Guide*" chapter "ActiveX".

As with XML RPC you define your object model. Detailed information on this you will find again in the "*Developer's Guide*" chapter "ActiveX". Here we assume you are already familiar with the required syntax.

You create an instrument and extend *com.cabaret.claptz.objectmodel.members* where you declare a method named *demo_validate* with your document type. For sure you will not

need a parameter here, as the document to be validated is the target of your method.

```
...
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.activedoc.CommonActiveDoc"
    name="demo_validate">
    <perform type="Script" source="scripts/validate"/>
  </method>
</extension>
...
```

The associated JavaScript script file

```
var idoc = jEvent.target.document;

/* validate state */
var state = Processor.callArgs(
  'com.cabaret.security.document.validation.DocumentValidatorFactory',
  {
    document: idoc
  }
);

Packages.com.cabaret.security.app.validation.ValidationTools.showState(state);

/* return state integer */
state.getState();
```

Your client creates an instance of the ActiveX object by loading the respective file. Then you call “demo_validate” without arguments via one of the ActiveX invocation API’s. Again, the object model declaration is selected and executed.

The method itself is implemented using standard techniques and some of the API’s we have seen above. First, the document object is selected from the call event. This document is then validated using the generic validator.

For details on programming the client, see “*Developer’s Guide*” chapter “ActiveX” and the demo sources deployed with Sign Live! CC.

2.4.5 Document validation, Commandline version

To use the commandline, there is nothing you have to prepare, this is simply builtin. We recommend to create a small “cli” file where you script the commandline options and later on use to call Sign Live! CC. More information on the commandline you will find in “*Operator’s Guide*” chapter “Automation”.

Here is the “validate.cli” file:

```

-silent
-perform -pt JavaScript -ps
"
var idoc = Processor.callArgs(
    'DocumentLoaderFactory', {
        locator: '${options.1}'
    });
var state = Processor.callArgs(
    'com.cabaret.security.document.validation.DocumentValidatorFactory',
    {
        document: idoc
    }
);

Packages.com.cabaret.security.app.validation.ValidationTools.showState(state);

/* dont forget to release */
idoc.release();
"
-popall

```

You can execute this via a command shell

```
start "SignLiveCC" "bin\SignLiveCC.exe" -cf validate.cli "%1"
```

Remember that “-cf” has to be the only option if it is used at all. All other options, including “-config” and so on must be written in the command line file.

The complete command file is expanded before use. The options to “-cf” are forwarded for string expansion and are available under the string variable namespace “options”. So, as in this example, you can access the filename with \${options.1}. \${options.0} is the commandfile itself - this is just as you are used to from plain batch files.

2.4.6 Validation for use in Script Center

As you know, there is also an interactive tool with Sign Live! CC to create, manage and run used defined scripts, the Script Center.

You can create a new script “validate.js” in the Script Center and save it somewhere, keeping a link on to the script in the center.

```

var idoc = jEvent.target.document;
Processor.callArgs(
    "com.cabaret.security.document.validation.DocumentValidatorFactory",
    {
        document: idoc
    }
);

```

This script will validate an open, active document, when you just double click it in the Script Center.

3. Encryption

3.1 Overview

This chapter presents the encryption features for Sign Live! CC. You will learn how to encrypt a document using different methods like “PKCS#7” or “PDF internal”, on different devices like local keystores or smartcards.

You will see how to do this completely programmatically or by calling the interactive wizard of Sign Live! CC itself.

The basic steps for encrypting data using PPK techniques are

- Generate a content encryption key (CEK) and encrypt the document’s binary representation symmetrically using this key.
- Encrypt the CEK with PPK techniques using an appropriate device.
- Store the encrypted data in some way.

The security framework provides some abstractions for the steps and objects involved.

- **Document**
The data to be encrypted. The document object is important to identify document specific or proprietary ways of encrypting data. This is for example the case with PDF internal encryption.
- **Symmetric Encryption Algorithm**
The algorithm used to encrypt the document’s content using a CEK. The selection of a symmetric encryption algorithm may be interdependent with the choice of the encryption method.
- **Asymmetric Encryption Device**
The algorithm and technical device used to encrypt the CEK. “Device” is an abstraction that can be implemented locally, using keystores or the windows certificate repository or a security token like a smartcard.

- **Encryption Method**

The method used to encrypt the document. This describes the process applied on the source document and the syntax and semantics of the serialized encrypted object. An encryption method may be a vendor defined one (like a PDF internal encryption) or a de facto standard like PKCS#7.

Most of these can be freely assembled to your encryption application of choice.

3.2 The Document

A document is represented using the *de.intarsys.document.model.IDocument* abstraction from the application platform. More information on this topic you can find in “*Developer’s Guide*”.

You will find basic support for these document types:

- PDF
- XML
- Text flavors
- Image flavors (BMP, PNG, Tiff, GIF, JPG)
- Generic transparent document (binary data)

Some encryption methods support all document types, some are restricted to specific types. This information you will find along with the encryption method.

3.3 The Device

3.3.1 The symmetric encryption algorithm

3.3.1.1 Overview

The symmetric encryption device is only the reference to the algorithm to be used for encrypting the document’s data.

3.3.1.2 Supported algorithms

The following is a list of supported algorithms for use with encryption

- AES-128 (CBC)
- AES-128 (CFB)
- AES-128 (ECB)
- AES-128 (OFB)
- AES 192 (CBC)
- AES-192 (CFB)
- AES-192 (ECB)
- AES-192 (OFB)
- AES 256 (CBC)

- AES-256 (CFB)
- AES-256 (ECB)
- AES-256 (OFB)
- 3DES-192 (CBC)

AES-128 (CBC) is the default algorithm.

3.3.2 The asymmetric encryption device

3.3.2.1 Overview

As the asymmetric encryption device is needed for the encryption method, we will introduce that first.

The asymmetric encryption device is an abstraction from both the algorithm used (for example RSA or elliptic curves) as well as the device (local CPU, security token, remote implementation, ...).

The asymmetric encryption device is implemented using the *de.intarsys.processor.model.IProcessor* abstraction from the application platform. More information on this topic you can find in “*Developer’s Guide*” chapter “Working with processors”.

The following asymmetric encryption devices are supported:

- Local keystore
- Smartcard

Some of these implementations may not be available with your installation, depending upon the license you achieved.

3.3.3 Local keystore encryption device

3.3.3.1 Overview

The local keystore encryption device supports certificates stored locally on the machine. It uses by default the Sign Live! CC key store implementation. Here you can access identities from the windows certificate repository or import from all popular formats like Java keystore or PKCS#12.

More about this key store can be found in the documentation about the certificate administration.

PROCESSOR FACTORY

`com.cabaret.security.device.keystore.encryption.KeyStoreEncryptorFactory`

ARGUMENTS

Name	Description
<hr/>	

`recipientIdentifier` Select the certificate used for encrypting the data.

This is a polymorphic argument that can accept a

- `de.intarsys.security.cert.IX509Certificate`
- `de.intarsys.security.cert.IX509CertificateSelector`

String identifying a certificate. In this string you can use the `SerialNumber`, **Subject** and **Issuer** to select the certificate from the store.

`recipients` Select multiple certificates for encrypting the data. This is a list of `recipientIdentifier` instances.

RESULT

The processor returns an object of type
`java.util.List<de.intarsys.security.cert.IX509PublicKeyCertificate>`

EXCEPTIONS

The processing may raise exceptions.

3.3.3.2 Example

You will not call an asymmetric encryption device directly, so here no example is given. An asymmetric encryption device is used as a parameter to an encryption method.

For reference purposes we will give you here the argument values needed to encrypt with the demo certificate in the local key store.

```
recipientIdentifier: 'SerialNumber:8139571262270123122;'
```

3.3.4 Smartcard encryption device

3.3.4.1 Overview

This encryption implementation uses a smartcard device and supports access to encryption certificates.

For a list of supported smartcards and smartcard readers see the operating documentation or online help.

PROCESSOR FACTORY

`de.intarsys.stage.security.device.smartcard.encryption.SmartcardEncryptorFactory`

ARGUMENTS

Name	Description
recipientIdentifier	<p>Select the certificate used for encrypting the data.</p> <p>This is a polymorphic argument that can accept a</p> <ul style="list-style-type: none"> • <code>de.intarsys.security.cert.IX509Certificate</code> • <code>de.intarsys.security.cert.IX509CertificateSelector</code> <p>String identifying a certificate. In this string you can use the <code>SerialNumber</code>, Subject and Issuer to select the certificate from the store.</p>

RESULT

The processor returns an object of type
`java.util.List<de.intarsys.security.cert.IX509PublicKeyCertificate>`

EXCEPTIONS

The processing may raise exceptions.

3.3.4.2 Example

You will not call an asymmetric encryption device directly, so here no example is given. An asymmetric encryption device is used as a parameter to an encryption method.

3.4 The Method

3.4.1 The encryption method

3.4.1.1 Overview

The encryption method is implemented using the *de.intarsys.processor.model.IProcessor* abstraction from the application platform. More information on this topic you can find in “*Developer’s Guide*” chapter “Working with documents”.

Currently these encryption methods are supported:

- PKCS#7 encryption

The encryption method is the central object, selected using the appropriate processor factory. All other components (document, symmetric encryption algorithm, asymmetric encryption device) are provided as arguments to this processor.

An encryption method can be selected using a well known processor factory or the generic factory, that itself selects a method based on the document type provided.

3.4.2 Generic encryption method

3.4.2.1 Overview

The generic encryption method delegates to the concrete encryption method appropriate for your system. “Appropriate” means it will select the method that fits the document and is compatible to the current preferences and other customizations active in the platform.

PROCESSOR FACTORY

`com.cabaret.security.document.encryption.DocumentEncryptorFactory`

ARGUMENTS

All arguments are forwarded to the selected concrete encryption method.

RESULT

The processor returns an object of type `de.intarsys.security.crypt.IEncryptedData`

EXCEPTIONS

The processing may raise exceptions.

3.4.2.2 Examples

```
//  
var idoc = ..read document from somewhere...  
//  
var encryptedData = Processor.callArgs(  
    "com.cabaret.security.document.encryption.DocumentEncryptorFactory",  
    {  
        document: idoc,  
        contentEncryptor: "AES-128 (CBC)",  
        cekEncryptor: {  
            factory:  
"com.cabaret.security.device.keystore.encryption.KeyStoreEncryptorFactory",  
            args: {  
                recipientIdentifier: 'SerialNumber:8139571262270123122;'  
            }  
        }  
    }  
);
```

This example encrypts a document, using the local Sign Live! CC demo certificate. The encryption format depends on the document type and the platform configuration.

3.4.3 PKCS#7 encryption method

3.4.3.1 Overview

The PKCS#7 encryption method will accept any document and encrypt it, compliant to the PKCS#7 standard.

PROCESSOR FACTORY

com.cabaret.security.method.pkcs7.encryption.PKCS7DocumentEncryptorFactory

ARGUMENTS

Name	Description
document	The document to be encrypted.
contentEncryptor	A symmetric encryption algorithm name as described above
cekEncryptor	The instance specification used to encrypt the content encryption key.
-.factory	
-.args	The arguments passed to the CEK encryptor
locator	The optional locator where to save the encrypted data. If <i>locator</i> is defined, the decrypted data is written to the file designated by this argument. A relative filename is interpreted within the directory of the input document. If no <i>locator</i> is defined, the default behavior depends upon the argument <i>createTransient</i> . For the <i>locator</i> name, by default the <i>locator</i> name of the input document is used, the suffix “p7m” is added.
createTransient	If no <i>locator</i> argument is provided, this flag determines the storage behavior for the encrypted data. If <i>createTransient</i> is <i>true</i> , the <i>locator</i> for the encrypted data is memory based, persistent (file based) otherwise. The default for <i>createTransient</i> is <i>false</i> .

RESULT

The processor returns an object of type *de.intarsys.security.crypt.IEncryptedData*

EXCEPTIONS

The processing may raise exceptions.

3.4.3.2 Examples

```
//
var idoc = ..read document from somewhere...
//
var encryptedData = Processor.callArgs(
  "com.cabaret.security.method.pkcs7.encryption.PKCS7DocumentEncryptorFactory",
  {
    document: idoc,
    contentEncryptor: "AES-128 (CBC)",
    cekEncryptor: {
      factory:
"com.cabaret.security.device.keystore.encryption.KeyStoreEncryptorFactory",
      args: {
        recipientIdentifier: 'SerialNumber:8139571262270123122;'
      }
    }
  }
);
```

This example creates a PKCS#7 encrypted file, using the local Sign Live! CC demo certificate.

```
//
var idoc = ..read document from somewhere...
//
var encryptedData = Processor.callArgs(
  "com.cabaret.security.method.pkcs7.encryption.PKCS7DocumentEncryptorFactory",
  {
    document: idoc,
    contentEncryptor: "AES-128 (CBC)",
    cekEncryptor: {
      factory:
"com.cabaret.security.device.keystore.encryption.KeyStoreEncryptorFactory",
      args: {
        recipients: [
          'SerialNumber:8139571262270123122;',
          'SerialNumber:8492403122200240190;'
        ]
      }
    }
  }
);
```

This example creates a PKCS#7 encrypted file, using multiple local identities.


```
//
var idoc = ..read document from somewhere...
//
var encryptedData = Processor.callArgs(
    "com.cabaret.security.method.pkcs7.encryption.PKCS7DocumentEncryptorFactory",
    {
        document: idoc,
        contentEncryptor: "AES-128 (CBC)",
        cekEncryptor: {
            factory:
"com.cabaret.security.device.keystore.encryption.KeyStoreEncryptorFactory",
            args: {
                recipients: [
                    'SerialNumber:8139571262270123122;',
                    {
                        recipientIdentifier: 'SerialNumber:8492403122200240190;'
                    }
                ]
            }
        }
    }
);
```

This is an alternate syntax that allows for more (not yet defined) arguments to identify a recipient. This example creates a PKCS#7 encrypted file, using multiple identities from different stores.

3.5 The Wizard

3.5.1 The encryption wizard

While you can call the encryption processors directly as seen in the chapters before, you can also bring up a wizard to collect all the arguments you must supply and call the encryption processor after this user interaction.

WIZARD FACTORY

com.cabaret.security.document.encryption.ui.DocumentEncryptorWizardFactory

ARGUMENTS

All arguments that are available with the encryption processors can be preselected.

Name	Description
document	The document to be encrypted must be supplied.

RESULT

The result of the wizard is the result of the encryption processor selected and executed.

3.5.2 Examples

```
//
var idoc = ...get document from somewhere..
//
var encryptedData = Wizard.callArgs(
  'com.cabaret.security.document.encryption.ui.DocumentEncryptorWizardFactory',
  {
    document: idoc
  }
);
```

This calls the encryption wizard. All arguments are preset to the values stored in the preferences from the last call (if applicable to the current document type).

3.6 Examples

3.6.1 Overview

In this chapter we want to provide some complete examples for interfacing with Sign Live! CC to create and customize encryption applications.

The standard scenario here is:

- Select the document
- Encrypt it

Be aware that all of the examples here are presented without error handling or ensure safe resource cleanup. In a production environment you should add such stuff (like “try...catch...finally”).

3.6.2 PKCS#7 encryption using a local keystore via ActiveX

As always, you define the object model... Detailed information on this you will find again in the “*Developer’s Guide*” chapter “ActiveX”.

You create an instrument and extend *com.cabaret.claptz.objectmodel.members* where you declare a method named *demo_encrypt* with your document type.

```
...
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.activedoc.CommonActiveDoc"
    modifiers="gui"
    name="demo_encrypt">
    <perform type="Script" source="scripts/encrypt"/>
  </method>
</extension>
...
```

The associated JavaScript script file:

```

var idoc = jEvent.target.document;

var encryptedData = Processor.callArgs(
    "com.cabaret.security.method.pkcs7.encryption.PKCS7DocumentEncryptorFactory",
    {
        document: idoc,
        contentEncryptor: "AES-128 (CBC)",
        cekEncryptor: {
            factory:
                "com.cabaret.security.device.keystore.encryption.KeyStoreEncryptorFactory",
            args: {
                recipientIdentifier: 'SerialNumber:8139571262270123122;'
            }
        }
    }
);

/* return the data location */
encryptedData.locator.fullName

```

Your client creates an instance of the ActiveX object by loading the respective file. Then you call “demo_encrypt” without arguments via one of the ActiveX invocation API’s.

The document is encrypted using the Sign Live! CC demo certificate and the result is stored at the default location.

3.6.3 Encryption wizard activation via ActiveX

To use ActiveX you must have installed the ActiveX support upon installation of Sign Live! CC.

To get general information about this API, you can consult “*Developer’s Guide*” chapter “ActiveX”.

Again, you first define your object model. Detailed information on this you will find again in the “*Developer’s Guide*” chapter “ActiveX”. Here we assume you are already familiar with the required syntax.

You create an instrument and extend *com.cabaret.claptz.objectmodel.members* where you declare a method named *demo_wizard_encrypt* with your document type.

```

...
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.activedoc.CommonActiveDoc"
    modifiers="gui"
    name="demo_wizard_encrypt">
      <perform type="Script" source="scripts/wizard_encrypt"/>
    </method>
  </extension>
...

```

The associated JavaScript script file:

```
var idoc = jEvent.target.document;

/* validate state */
var encryptedData = Wizard.callArgs(
    'com.cabaret.security.document.encryption.ui.DocumentEncryptorWizardFactory',
    {
        document: idoc
    }
);

/* return the data location */
encryptedData.locator.fullName
```

Your client creates an instance of the ActiveX object by loading the respective file. Then you call “demo_wizard_encrypt” without arguments via one of the ActiveX invocation API’s.

Remember the special declaration:

```
... modifiers="gui"
...
```

Again, the document object is selected from the call event. This document is forwarded as an argument to the wizard. The wizard will care for the rest.

For details on programming the client, see “*Developer’s Guide*” chapter “ActiveX” and the demo sources deployed with Sign Live! CC.

4. Decryption

4.1 Overview

This chapter presents the decryption features for Sign Live! CC. You will learn how to decrypt a document using different methods like “PKCS#7” or “PDF internal”, on different devices like local key stores or smartcards.

You will see how to do this completely programmatically or by calling the interactive wizard of Sign Live! CC itself.

The basic steps for decrypting data using PPK techniques are

- Determine an applicable decryption method.
- Decrypt the content encryption key (CEK) using an appropriate device.
- Decrypt the document using the CEK.

The security framework provides some abstractions for the steps and objects involved.

- Document
The data to be decrypted. The document object is important to identify document specific or proprietary ways of decrypting data. This is for example the case with PDF internal encryption.
- Decryption Method
The method used to decrypt the document. This describes the process applied on the source document. A decryption method may be a vendor defined one (like a PDF internal decryption) or a de facto standard like PKCS#7. The decryption method is determined by the encryption format.
- Decryption Device
The algorithm and technical device used to decrypt the CEK. “Device” is an abstraction that can be implemented locally, using

keystores or the windows certificate repository or a security token like a smartcard.

4.2 The Device

4.2.1 Overview

As the decryption device is needed for the decryption method, we will introduce that first.

The decryption device is an abstraction from both the algorithm used (for example RSA or elliptic curves) as well as the device (local CPU, security token, remote implementation, ...).

The decryption device is implemented using the *de.intarsys.processor.model.IProcessor* abstraction from the application platform. More information on this topic you can find in “*Developer’s Guide*” chapter “Working with processors”.

The following decryption devices are supported:

- Local keystore
- Smartcard

Some of these implementations may not be available with your installation, depending upon the license you achieved.

4.2.2 Local keystore decryption device

4.2.2.1 Overview

Decrypting using a certificate from a local keystore supports certificates stored locally on the machine. It uses by default the Sign Live! CC key store implementation. Here you can access identities from the windows certificate repository or import from all popular formats like Java keystore or PKCS#12.

More about this key store can be found in the documentation about the certificate administration.

PROCESSOR FACTORY

`com.cabaret.security.device.keystore.decryption.KeyStoreDecryptorFactory`

ARGUMENTS

Name	Description
<code>decryptorIdentifier</code>	Select the certificate used for decrypting the data. This is a polymorphic argument that can accept a

`de.intarsys.security.cert.IX509Certificate`

`de.intarsys.security.cert.IX509CertificateSelector`

String identifying a certificate. In this string you can use the **SerialNumber**, Subject and Issuer to select the certificate from the store.

`decryptorPassword` The password to open the signers private key.

RESULT

The processor returns an object of type
`de.intarsys.security.device.common.decryption.IDecryptor`

EXCEPTIONS

The processing may raise exceptions.

4.2.2.2 Example

You will not call a decryption device directly, so here no example is given. A decryption device is used as a parameter to a decryption method.

For reference purposes we will give you here the argument values needed to encrypt with the Sign Live! CC demo certificate in the local key store.

```
decryptorIdentifier: 'SerialNumber:8139571262270123122;'
decryptorPassword: 'password'
```

4.2.3 Smartcard decryption device

4.2.3.1 Overview

This decryption implementation uses a smartcard device and supports access to decryption certificates.

For a list of supported smartcards and smartcard readers see the operating documentation or online help.

PROCESSOR FACTORY

`de.intarsys.stage.security.device.smartcard.decryption.SmartcardDecryptorFactory`

ARGUMENTS

Name	Description
------	-------------

decryptorIdentifier Select the certificate used for decrypting the data. This is a polymorphic argument that can accept a

- `de.intarsys.security.cert.IX509Certificate`
- `de.intarsys.security.cert.IX509CertificateSelector`

String identifying a certificate. In this string you can use the **SerialNumber**, Subject and Issuer to select the certificate from the store.

decryptorPassword The password to open the signers private key.

RESULT

The processor returns an object of type
`de.intarsys.security.device.common.decryption.IDecryptor`

EXCEPTIONS

The processing may raise exceptions.

4.2.3.2 Example

You will not call a decryption device directly, so here no example is given. A decryption device is used as a parameter to a decryption method.

4.3 The Method

4.3.1 The decryption method

4.3.1.1 Overview

The decryption method is implemented using the *de.intarsys.processor.model.IProcessor* abstraction from the application platform. More information on this topic you can find in “*Developer’s Guide*” chapter “Working with documents”.

Currently these decryption methods are supported:

- PKCS#7 decryption

The decryption method is the central object, selected using the appropriate processor factory. All other components (document, decryption device) are provided as arguments to this processor.

A decryption method is selected implicitly based on the document type provided.

4.3.2 Generic decryption method

4.3.2.1 Overview

The generic decryption method delegates to the concrete decryption method appropriate for your system. “Appropriate” means it will select the method that fits the document and is compatible to the current preferences and other customizations active in the platform.

PROCESSOR FACTORY

`com.cabaret.security.document.decryption.DocumentDecryptorFactory`

ARGUMENTS

All arguments are forwarded to the selected concrete decryption method.

RESULT

The processor returns an object of type *de.intarsys.security.crypt.IDecryptedData*.
The document must be released by the caller explicitly.

EXCEPTIONS

The processing may raise exceptions.

4.3.2.2 Examples

```
//
var encryptedIdoc = ..read encrypted document from somewhere...
//
var decryptedData = Processor.callArgs(
    "com.cabaret.security.document.decryption.DocumentDecryptorFactory",
    {
        document: encryptedIdoc,
        cekDecryptor: {
            factory:
"com.cabaret.security.device.keystore.decryption.KeyStoreDecryptorFactory",
            args: {
                decryptorIdentifier: 'SerialNumber:8139571262270123122;',
                decryptorPassword: 'password'
            }
        }
    }
);
```

This example decrypts a document in-memory, using a local certificate.
The decryption method depends on the document type and the platform configuration.

Don't forget to release the result document you created!

4.3.3 PKCS#7 decryption method

4.3.3.1 Overview

The PKCS#7 decryption method will accept any document compliant to the PKCS#7 standard and decrypt it

PROCESSOR FACTORY

com.cabaret.security.method.pkcs7.decryption.PKCS7DocumentDecryptorFactory

ARGUMENTS

Name	Description
document	The document to be decrypted.
cekDecryptor	The instance specification used to decrypt the content encryption key.
-factory	
-args	The arguments passed to the CEK decryptor.
locator	The optional locator where to save the decrypted data. If locator is defined, the decrypted data is written to the file designated by this argument. A relative filename is interpreted within the directory of the encrypted document. If no <i>locator</i> is defined, the default behavior depends upon the argument <i>createTransient</i> . For the locator name, by default the locator name of the input document is used, the suffix is stripped. If the remaining name has no more suffix, a new suffix is guessed from the data content.
createTransient	If no <i>locator</i> argument is provided, this flag determines the storage behavior for the decrypted data. If <i>createTransient</i> is <i>true</i> , the locator for the decrypted data is memory based, persistent (file based) otherwise. The default for <i>createTransient</i> is <i>true</i> .

RESULT

The processor returns an object of type *de.intarsys.security.crypt.IDecryptedData* .
The document must be released by the caller explicitly.

EXCEPTIONS

The processing may raise exceptions.

4.3.3.2 Examples

```
//
var encryptedIdoc = ..read encrypted document from somewhere...
//
var decryptedData = Processor.callArgs(
    "com.cabaret.security.method.pkcs7.decryption.PKCS7DocumentDecryptorFactory",
    {
        document: encryptedIdoc,
        cekDecryptor: {
            factory:
                "com.cabaret.security.device.keystore.decryption.KeyStoreDecryptorFactory",
            args: {
                decryptorIdentifier: 'SerialNumber:8139571262270123122;',
                decryptorPassword: 'password'
            }
        }
    }
);
```

This example decrypts a PKCS#7 file in-memory, using a local identity.

4.4 The Wizard

4.4.1 The decryption wizard

While you can call the decryption processors directly as seen in the chapters before, you can also bring up a wizard to collect all the arguments you must supply and call the decryption processor after this user interaction.

WIZARD FACTORY

com.cabaret.security.document.decryption.ui.DocumentDecryptorWizardFactory

ARGUMENTS

All arguments that are available with the encryption processors can be preselected.

Name	Description
document	The document to be decrypted must be supplied.

RESULT

The result of the wizard is the result of the decryption processor selected and executed.

4.4.2 Examples

```
var encryptedIdoc = ...get document from somewhere..  
//  
var decryptedData = Wizard.callArgs(  
    'com.cabaret.security.document.decryption.ui.DocumentDecryptorWizardFactory',  
    {  
        document: encryptedIdoc  
    }  
);
```

This calls the decryption wizard. All arguments are preset to the values stored in the preferences from the last call (if applicable to the current document type).

4.5 Examples

4.5.1 Overview

In this chapter we want to provide some complete examples for interfacing with Sign Live! CC to create and customize decryption applications.

The standard scenario here is:

- Select the document
- Decrypt it

Be aware that all of the examples here are presented without error handling or ensure safe resource cleanup. In a production environment you should add such stuff (like “try...catch...finally”).

4.5.2 Simple Decryption wizard activation via ActiveX

This demo is a little bit different. When we load an encrypted document via the ActiveX API, an *de.intarsys.document.model.IDocument* for the encrypted data is created. This is what we want - but, when opening a viewer on this document, there is “magic” in Sign Live! CC that will detect that this is most probably not a document type to be viewed of its own. Instead of viewing the document, a wizard is opened that queries the information needed to decrypt. The decrypted result document is viewed instead.

This is why simply loading an encrypted document instance in an interactive ActiveX instance results directly in performing the wizard and displaying the result.

4.5.3 Headless Decryption wizard via ActiveX

As always, create an object model declaration. You create an instrument and extend *com.cabaret.claptz.objectmodel.members* where you declare a method named *decrypt* with your document type.

```
...
<extension point="com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.activedoc.CommonActiveDoc"
    modifiers="gui"
    name="demo_decrypt">
    <perform type="Script" source="scripts/decrypt"/>
  </method>
</extension>
...
```

The associated JavaScript script file:

```
var idoc = jEvent.target.document;

var decryptedData = Processor.callArgs(
  "com.cabaret.security.document.decryption.DocumentDecryptorFactory",
  {
    document: idoc,
    createTransient: false,
    cekDecryptor: {
      factory:
"com.cabaret.security.device.keystore.decryption.KeyStoreDecryptorFactory",
      args: {
        decryptorIdentifier: 'SerialNumber:8139571262270123122;',
        decryptorPassword: 'password'
      }
    }
  }
);
```

Your client creates an instance of the ActiveX object by loading the respective file - be sure to do this “headless”. Then you call “demo_decrypt” without arguments via one of the ActiveX invocation API’s.

5. Workflow integration

5.1 Overview

In many situations you may prefer to automate a complete workflow using the application instead of doing everything by hand. A typical example is

- Finish a document (by saving it to a special location or printing it)
- Make some special preparation steps with the document
- Detect contextual information from the document (like email address or signer information).
- Sign it
- Send the signed document via Mail to your business partner
- Archive your Document locally
- Make some finishing step with the document

Sign Live! CC can help you to achieve this using some preinstalled business function components.

Using the service container framework you can publish this workflow on all Sign Live! CC channels like

- Printer
- File System Monitor
- HTTP and SOAP

For more information on the service container framework see the “Operators Guide”.

5.2 Cosima

Cosima is the simplest approach to get a fully functional workflow. You just parameterize the process using a graphical user interface. Every process step is neatly predefined. Integration of signature in your

everyday business can't get easier. This gets especially clear when you use Cosima as a virtual printer, signing, mailing and archiving your business documents.

Cosima integrates the following steps

- Document conversion
- Tag detection
- Signature
- Mailing
- Archiving

From a high level view, Cosima is a business function itself and as such works the same as the other processor components you have found so far.

Internally the arguments to Cosima are simply split and forwarded to the respective processor components for each step.

5.2.1 Conversion

5.2.1.1 Overview

An incoming document can be converted to another format at the start of the workflow. Basically there are two possibilities today.

- Convert Postscript to PDF
This is useful when accepting input directly as a virtual printer. You will need to install Ghostscript to use this feature.
- No conversion
You use the document just as it is, for example if you ERP system creates directly PDF output and the Sign Live! CC File System Monitor accepts it as input.

The result of the conversion is the basis for all subsequent steps.

5.2.1.2 Arguments

Name	Description
converter.processor	The processor to be used for conversion <ul style="list-style-type: none"> - com.cabaret.pdf.exchange.ps.PSImporterFactory - <none>
converter.args.*	Arguments to the converter processor

5.2.2 Tagging (keywords)

5.2.2.1 Overview

A document can be enriched with “tags” or “keywords” that may be used anywhere else in the processing context. For example you can embed (invisibly) a mail address in the document that is extracted and used later on when creating and sending the mail.

The tags themselves do not carry any semantics – the subsequent processing steps simply access and interpret them. If a processor uses tags, it is explicitly mentioned in the respective documentation.

A special feature in Cosima is the propagation of tag content into the arguments for later use. You will find information on this in a later chapter.

In Cosima, the following tagging processors and sources are supported

- <none>
No tag processing performed
- PDF tag extraction
From within the PDF content, text enclosed in “@@” characters is extracted.
- Property file format
From an attached file in Java property file format tags are extracted
- XML file format
From an attached file in XML format tags are extracted

5.2.2.2 Arguments

Name	Description
tagger.processor	<p>The processor to be used for tagging</p> <ul style="list-style-type: none"> - com.cabaret.pdf.processor.tagdetector.PDFContentTagDetectorFactory - com.cabaret.application.document.processor.PropertiesTagDetectorFactory - com.cabaret.application.document.processor.XMLTagDetectorFactory - <none>
tagger.args.*	Arguments to the tagging processor

5.2.3 Signature

5.2.3.1 Overview

For the document under processing a signature can be created. All available signature methods are described above. Shortly, you should select one

- `com.cabaret.security.document.signing.DocumentSignerFactory`
The signature processor is selected based on the document type under processing. E.g. for a PDF document an internal signature is applied, for other documents an external PKCS#7 signature is created.
- `com.cabaret.security.method.pkcs7.signing.PKCS7DocumentSignerFactory`
Create an external PKCS#7 signature

5.2.3.2 Arguments

Name	Description
<code>signer.processor</code>	The processor to be used for signing. The signature processors are already described above and can all be used here (with all descendent arguments).
<code>signer.args.*</code>	Arguments to the signature processor. Details you will find in the chapters above

5.2.4 Mail

5.2.4.1 Overview

The document, along with its attachments, can be sent by mail.

This step is especially well suited for integrating document tags. For example you can easily set the receiver for an email using the tag

```
@@emailTo=foo@bar.com@@
```

5.2.4.2 Well known tags

These tags are automatically mapped to arguments internally.

Key	Alias	Mailer Argument
-----	-------	-----------------

emailTo	aa	mailer.args.to
emailCc	cc	mailer.args.cc
emailSubject	ed	mailer.args.subject
emailMsg	sa	mailer.args.msg
emailMsgLocator	mb	mailer.args.msgLocator
emailInteractive		mailer.args.interactive

5.2.4.3 Arguments

Name	Description
mailer.processor	<p>The processor to be used for mailing</p> <ul style="list-style-type: none"> - com.cabaret.processor.mail.MapiMailerFactory Send Mail using the local MAPI installation. - com.cabaret.processor.mail.SmtplibMailerFactory Send mail using the SMTP configuration of Sign Live! CC - com.cabaret.processor.mail.AppleScriptMailerFactory As one can easily discover, this one is available on the Mac platform only. - <none>
mailer.args.*	Arguments to the mailing processor
mailer.args.msgLocator	This is set to "templates/email.txt" by default.

5.2.4.4 Example

For a simple example, create an "email.txt" like this and copy it to the "templates" directory in your profile directory.

```

${tags.salutation},

We are proud to introduce to you: the invoice for ${tags.month}!

Kind regards!

```

Providing the corresponding tags, for example embedded in the PDF

```

..
@@salutation=Dear sirs@@
@@month=June 2013@@
..

```

you will get the following mail content

```

Dear sirs,

We are proud to introduce to you: the invoice for June 2013!

Kind regards!

```

5.2.5 Archive

5.2.5.1 Overview

After all these steps you may want to make an archive copy of the task. Archiving is a broad domain and we are giving here only some examples:

- com.cabaret.archive.filesystem.FileSystemArchiverFactory
Copy to a dedicated Folder
- More available depending on your individual license. For example support is available for FTP, Alfresco, ELO and more.

5.2.5.2 Arguments

Name	Description
archiver.processor	The processor to be used for archiving <ul style="list-style-type: none"> - com.cabaret.archive.filesystem.FileSystemArchiverFactory - <none>
archiver.args.*	Arguments to the archiving processor

5.2.6 Notification

5.2.6.1 Overview

You can add any action at the end of the process, for example to inform some other system component of the termination. A notification is only created when the process was successful.

The processor used here is a generic “CodeExit” processor.

5.2.6.2 Arguments

Name	Description
notifier.processor	<p>The processor to be used for notification</p> <ul style="list-style-type: none"> - com.cabaret.claptz.common.codeexit.processor.CodeExitRunnerFactory - <none>
notifier.args.type	CodeExit type
notifier.args.source	CodeExit source

5.2.7 Installation

5.2.7.1 Ghostscript

For working with the Postscript conversion a current Ghostscript Installation must be present.

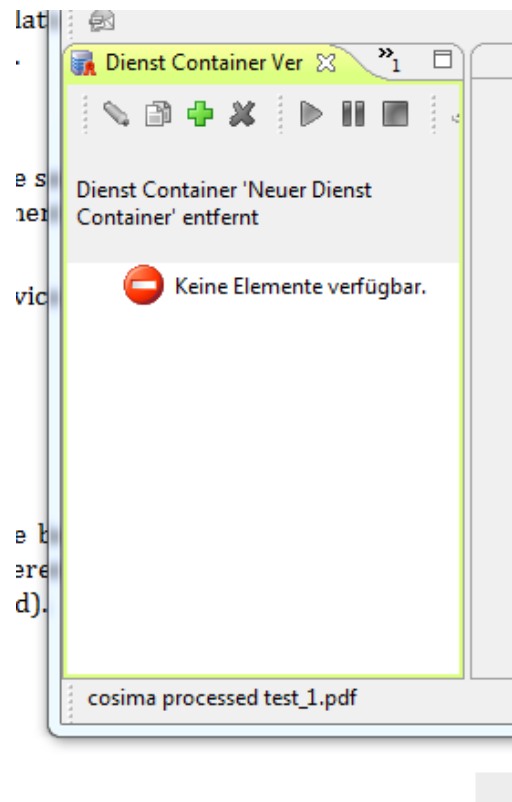
More information on the postscript conversion can be found in the “Operators Guide”.

5.2.7.2 Cosima

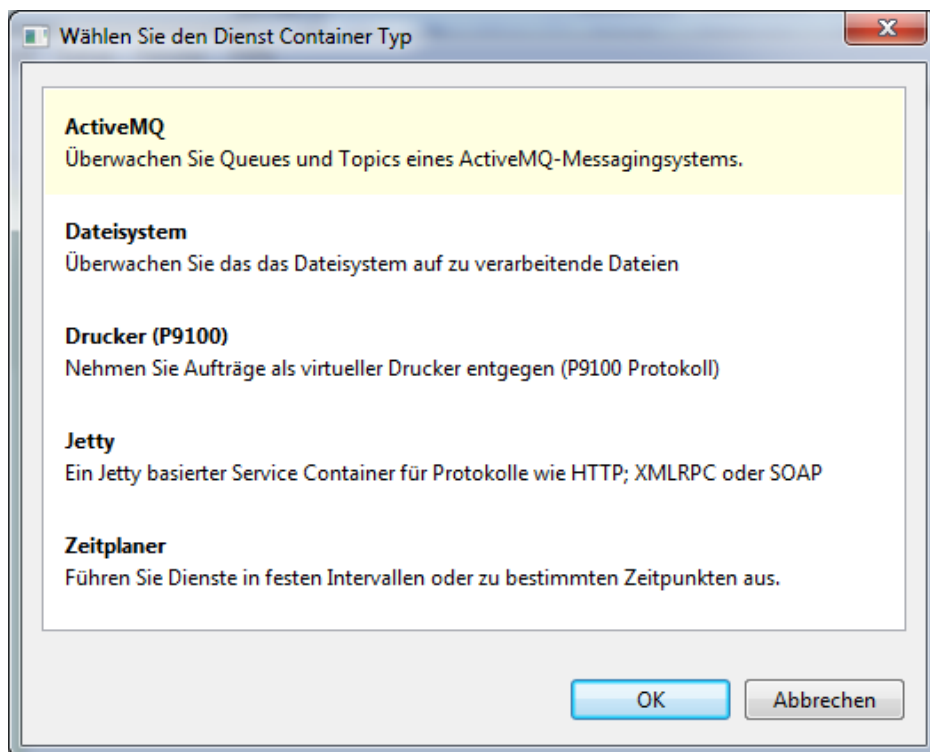
Cosima is part of the standard installation and can be selected in the service container definition process.

5.2.8 Configuration

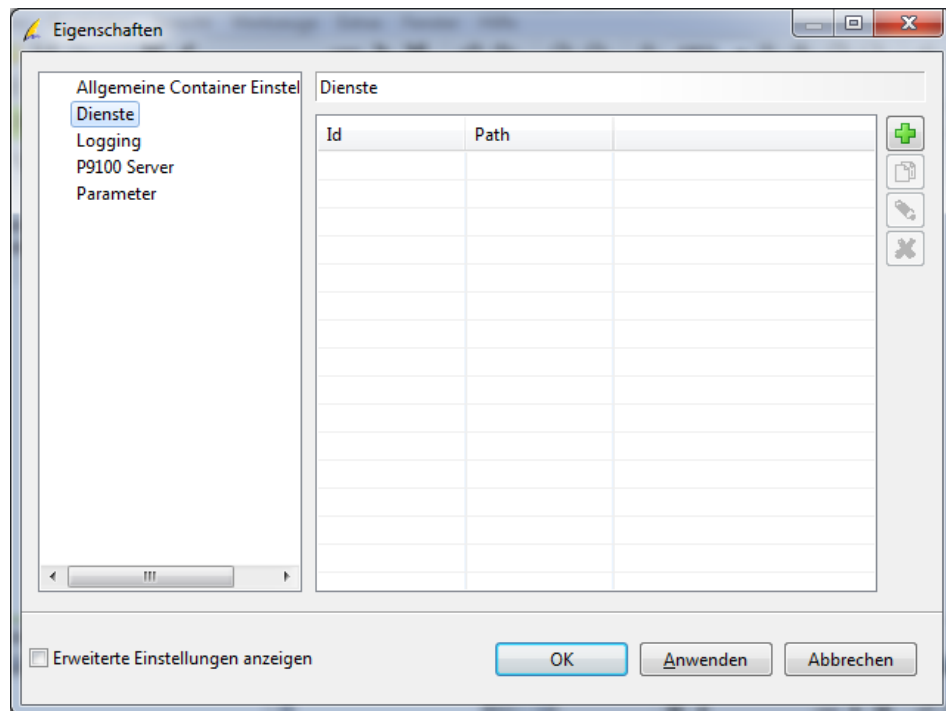
Cosima is best used embedded in the service container framework. So, open the service container management (Menu Extras->Services->Service container management).



Select the + sign to create a new service container.

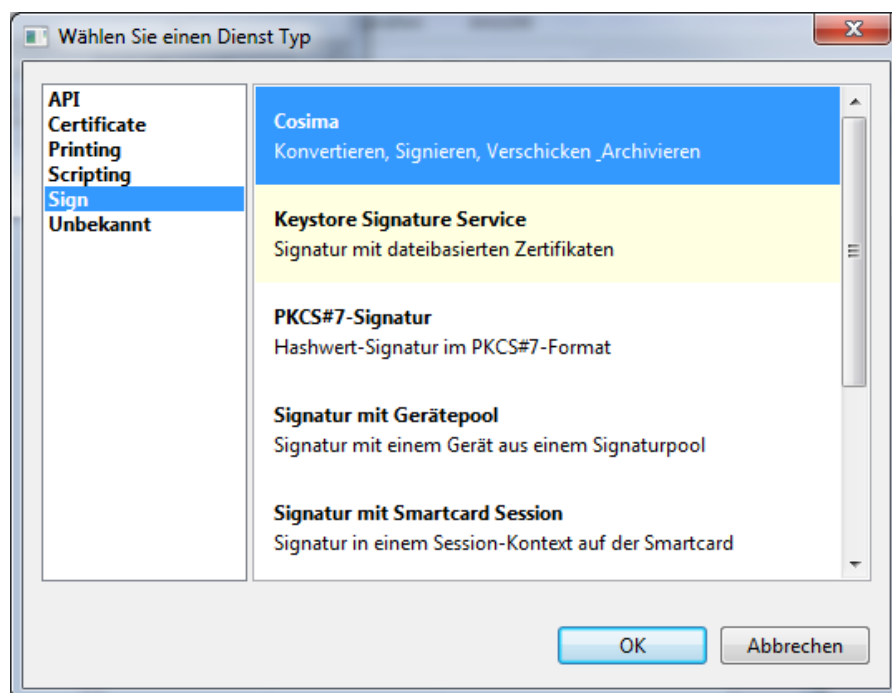


The virtual printer and the File System Monitor are best suited for simple workflows. We select the printer for our example.

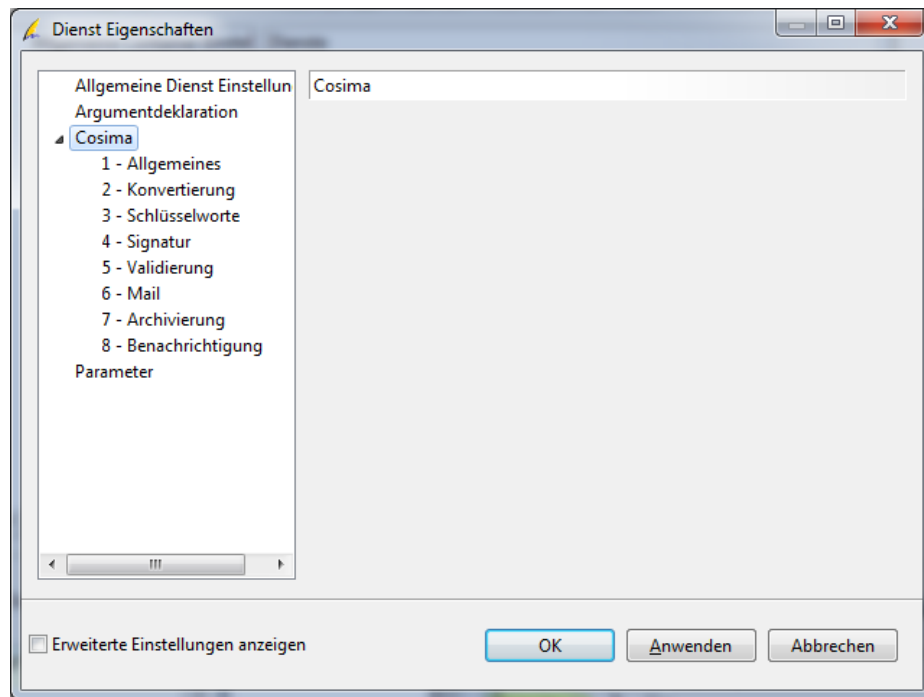


You can set up some details for the service container – more information can be looked up in the “Operators Guide”. For us, the default is just fine. We will create a network printer on port 9100.

From here select the “+” on the “Services” tab.



The “Cosima” processor can be found in the “Sign” section. You will see a detail configuration for the Cosima service now.



For any step in the chapter above you will see a dedicated tab for your customization.

After “OK”ing all dialogs your new service is available and only has to be started.

Your virtual printer is available at port 9100. It is best used with a Ghostscript installation using the Ghostscript PS printer driver.

5.2.9 Tipps and Tricks

5.2.9.1 “Staged” installation

The simplest solution to differentiate between a production and test system is the installation of two different service container instances (along with virtual printers if needed).

5.2.10 Arguments

For most scenarios the arguments configurable via the GUI should be enough. For more complicated settings you can fall back to other arguments sources described here - you can hand arguments to Cosima even if you use non API services like printer or file system.

The argument sources are presented in ascending level of visibility. This means that an argument defined as a service parameter is always overwritten by an argument defined by a client via API.

5.2.10.1 GUI argument settings

Most of the arguments described above can be defined on specific GUIs when configuring the Cosima service.

5.2.10.2 Generic argument definition

If you have arguments that are not supported by some specialized GUI, you can still add the argument in the service configuration. There's a specialized settings tab "Arguments".

For more information about generic service arguments, see the "Operators Guide".

5.2.10.3 Client arguments

A client operating via synchronous APIs like HTTP or SOAP can send arguments directly.

5.2.10.4 Tag based arguments

You can embed arguments in tags. To do so you can simply define a tag like

```
@@args.<argname>=<value>@@
```

Example

```
@@args.signer.args.field.create=true@@
```

This will force the creation of a visible signature field via a tag.

Using this technique you can create a complete argument list for the cosima process.

5.2.10.5 Argument "crunching"

Cosima and its processor steps are powerful, but there's still a small gap missing – and here's the solution.

The best example to introduce this is the signature field position. You can hardcode the argument in the Cosima service configuration on the "Arguments" tab – this will do the job most of the times.

Now, it would be nice if a client could provide this information on a per signature base. No problem, this is working, too. You can add an argument entry to the client arguments list, depending on the protocol.

If you don't have an explicit protocol (like a HTTP call) and can not defined arguments, you can revert to embedding the arguments into the document itself. Add a tag in the document like

```
@@args.signer.args.field.position=100x100@@
```

But, what if the information we need is not available at the time we embed the argument in the document? As an example we want to add a signature field at some place in text where a "marker" has been left. We want the signature field to appear where the "magic marker" is in the text. To get this done you need two new features of Cosima:

- the metatags

- the argument cruncher

Metatags are created by (some) tagging processors, especially the tagging based on the PDF content. To each and every tag found, information is stored about:

- the lower left corner of the bounding rect
- the upper right corner of the bounding rect
- the size of the bounding rect
- the page index of the bounding rect

The respective tags to lookup are

```
meta.<tag>.llx
meta.<tag>.lly
meta.<tag>.urx
meta.<tag>.ury
meta.<tag>.width
meta.<tag>.height
meta.<tag>.page
```

where you replace the <tag> with the real tag that you inserted in the PDF.

Lets assume you use "separated" tags in the document. Now you can do magic like

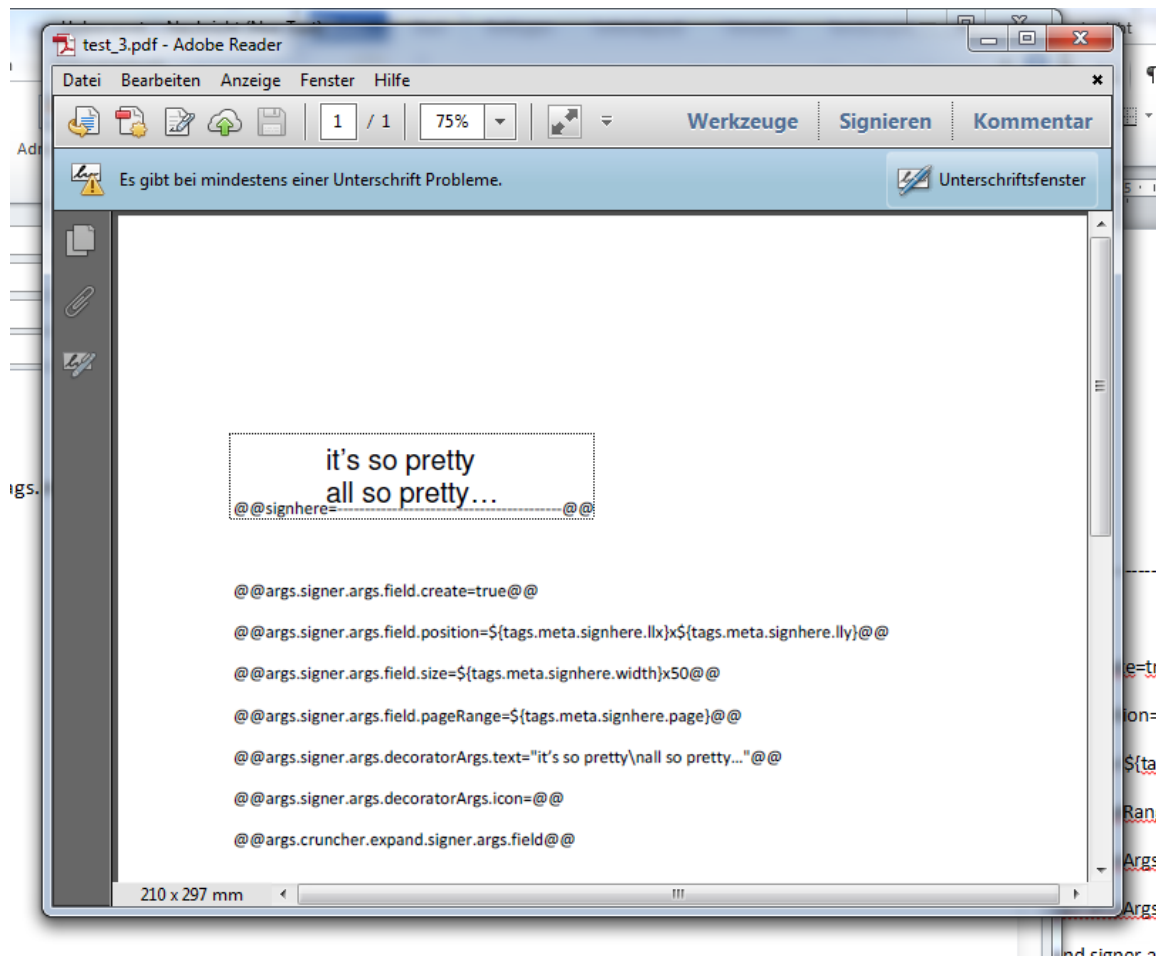
```
@@signhere=-----@@
...
@@args.signer.args.field.create=true@@
@@args.signer.args.field.position=${tags.meta.signhere.llx}x${tags.meta.signhere.lly}@@
@@args.signer.args.field.size=${tags.meta.signhere.width}x50@@
@@args.signer.args.field.pageRange=${tags.meta.signhere.page}@@
@@args.signer.args.decoratorArgs.text="it's so pretty\nall so pretty..."@@
@@args.signer.args.decoratorArgs.icon=@@
```

What is really sad, though, is the fact that the signature processor does not expand the "field" argument...

The solution is simply to get someone else to do it for you: for every argument where the processor does no expansion itself, you can instruct a delegate, the "cruncher" to do so. This is a processor that manipulates arguments before the "real" processors are run. This processor supports "setting", "removing" and "expanding" (!) existing args. It is included in the Cosima workflow by default and can be addressed at the "cruncher" argument binding. You have to add another argument that instructs the cruncher to expand the "field" argument.

```
...
@@args.cruncher.expand.signer.args.field@@
```

The result looks like this



Well, for real world scenarios you should make the tags invisible, but we're pretty close...

6. Workflow Processors

6.1 Postscript Conversion

6.1.1 Overview

Accept a print output directly as a virtual printer. Using Ghostscript (a postscript interpreter) a PDF version is created and injected in the workflow.

6.1.2 Synopsis

PROCESSOR FACTORY

com.cabaret.pdf.exchange.ps.PSImporterFactory

ARGUMENTS

Name	Description
importLocator	Reference to the input data.
outputLocator	<p>A template defining where to store the converted output.</p> <p>For string expansion the following variables are available in addition to the standard ones (see “Operators Guide”).</p> <ul style="list-style-type: none"> - all DSC header comments
ghostscriptArgs	<p>A list of commandline arguments for Ghostscript. Every argument is expanded before use.</p> <p>These are for example the default arguments sent to Ghostscript (the line breaks are here for readability purposes</p>

only!)

```
-dNOPAUSE; \
-dNOPAGEPROMPT; \
-dNOPROMPT; \
-dBATCH; \
-I${processor.includepath}; \
-sOutputFile=${processor.output}; \
-sDEVICE=pdfwrite; \
-r300; \
-c; \
.setpdfwrite; \
-c; \
save; \
-c; \
pop; \
-f; \
${processor.input}
```

Be careful what you are doing when providing your own commandline arguments. Only these are used and they are used as you provide them. As we call Ghostscript synchronously, the system may hang with “wrong” combinations, for example when you omit the commands for acting in non-interactive batch mode.

6.2 PDF “@@” tag extraction

6.2.1 Overview

From the PDF text all tags enclosed in “@@” are extracted.

Example

```
@@emailTo=support@intarsys.de@@
```

Normally this text is printed invisible (white characters with white background) in the document, but text extraction will find it anyway.

Two syntax formats are accepted

- prefixed
A 2 character prefix, followed by directly by the tag value

Example

```
@@aaValue@@
```

- separated
Within @@ you provide the key, a “=” followed by the value. This is more flexible and better readable.

Example

```
@@key=value@@
```

6.2.2 Caveats

Tag extraction depends highly on the external systems and resources.

- Your document must be “well formed” with regard to the text chunks. PDF is not really a readable text. Text extraction depends on detecting characters being near other characters and drawing commands being in the correct order! You must test your input if the processor is able to find your tags. Especially multi line tags and tags in tables may cause problems.
- Your document must contain text together with a well-defined encoding. This may depend on the combination of system platform (Mac, Windows), the word processor and the font used. If you can’t extract text from your document in Acrobat Reader, we cannot either!

6.2.3 Synopsis

PROCESSOR FACTORY

```
com.cabaret.pdf.processor.tagdetector.PDFContentTagDetectorFactory
```

ARGUMENTS

Name	Description
document	The document we attach the tags to
syntax	One of <ul style="list-style-type: none"> - prefixed - separated

6.3 Tags in property file format

6.3.1 Overview

Tags can be extracted from file attachments in standard Java Property file format. For more information on the format see the Java language specification. It is important to keep in mind

- You have to encode your content using ISO 8859-1
- All other characters can be encoded using Unicode escapes
- This particular implementation supports duplicate keys

Attachments are created automatically when you are using the File System monitor.

Synchronous APIs like HTTP or SOAP can use the argument “attachments”.

Using the printer you cannot create attachments.

6.3.2 Synopsis

PROCESSOR FACTORY

com.cabaret.application.document.processor.PropertiesTagDetectorFactory

ARGUMENTS

Name	Description
attachments	A list of locators to attachments
extension	The attachment extension to look for. Default is “.tags”.
document	The document we attach the tags to

6.3.3 Example

A typical “.tags” file will look like

```
foo=bar  
diedel=doedel
```

6.4 Tags in XML file format

6.4.1 Overview

Tags can be extracted from file attachments in XML format.

Attachments are created automatically when you are using the File System monitor.

Synchronous APIs like HTTP or SOAP can use the argument “attachments”.

Using the printer you cannot create attachments.

6.4.2 Synopsis

PROCESSOR FACTORY

com.cabaret.application.document.processor.XMLTagDetectorFactory

ARGUMENTS

Name	Description
attachments	A list of locators to attachments
extension	The attachment extension to look for. Default is “.xtags”.
document	The document we attach the tags to
tagExpression	The xpath expression to select the tag elements from the XML attachment. Default is “//tag” (all elements named “tag”)
keyAttribute	The name of the attribute in the tag element that denotes the key. Default is “key”.
valueAttribute	The name of the attribute in the tag element that denotes the value. Default is “value”.
valueSelection	One of <ul style="list-style-type: none"> - attribute (Default) Read the value from the attribute mentioned above - text Read the value from the element text - textTrim - Read the value from the element text, trim and normalize spaces

6.4.3 Example

A typical “.xtags” file will look like


```
<tags>
  <tag key="foo" value="bar"/>
  <tag key="diedel" value="doedel"/>
</tags>
```

6.5 All Mail processors

6.5.1 Overview

All mail processors share some common behavior and arguments that are described here.

6.5.2 Mail Content

The mail content can be defined either literally via “msg” or by reference, using “msgLocator”.

Example, tag embedded in a PDF document

```
..
@@emailMsg=Attached you will find your Invoice.
Kind regards!@@
..
```

The mail content itself gets expanded before use (see string expansion in the “Operators Guide”). This makes it easy to create a static template that gets individualized before sending.

Example, static template file referenced in “msgLocator” later. For maximum portability, the text encoding expected is “UTF-8”.

```
${tags.salutation},

We are proud to introduce to you: the invoice for ${tags.month}!

Kind regards!
```

This file will be looked up relative to your <profiledir>. You can set the location for this file in the “msgLocator” argument.

6.5.3 Synopsis

PROCESSOR FACTORY

-

ARGUMENTS

Name	Description
to	The recipient
cc	Carbon copy recipient

subject	The mail subject
msg	The mail message text
msgLocator	A reference to the mail message text

6.6 Mail via MAPI

6.6.1 Overview

Send the mail using the local native MAPI application.

6.6.2 Synopsis

PROCESSOR FACTORY

com.cabaret.processor.mail.MapiMailerFactory

ARGUMENTS

Name	Description
interactive	Flag if the MAPI application should show up in interactive mode.

6.7 Mail via SMTP

6.7.1 Overview

Send the mail using a SMTP server in the network.

6.7.2 Synopsis

PROCESSOR FACTORY

com.cabaret.processor.mail.SmtpMailerFactory

ARGUMENTS

Name	Description
smtpHost	Hostname/IP of SMTP server

smtpPort	Port of SMTP service
smtpSsl	Flag if SSL is to be used
smtpSender	A default for the “sender” attribute oif the mail
smtpTimeout	Timeout for SMTP protocol in millis
smtpUser	SMTP user for authentication
smtpPassword	SMTP password for authentication

6.8 Mail via AppleScript

6.8.1 Overview

Send mail on Mac using AppleScript.

6.8.2 Synopsis

PROCESSOR FACTORY

com.cabaret.processor.mail.AppleScriptMailerFactory

ARGUMENTS

Name	Description
------	-------------

6.9 File system archiving

6.9.1 Overview

The document and the current attachments are copied to a dedicated folder in the file system.

6.9.2 Synopsis

PROCESSOR FACTORY

com.cabaret.archive.filesystem.FileSystemArchiverFactory

ARGUMENTS

Name	Description
archiveDir	A template referencing the directory where to save the document and its attachments. This one is looked up relative to your profile dir.

7. Advanced Signature Features

7.1 Pools

7.1.1 Pools

The devices in the examples so far were used for a single security operation at a time. For signing a new document a new signer had to be created, the PIN had to be entered and so on.

For high performance applications and appliances this is not a reasonable environment. Here we'd like to create and prepare a device for fast and efficient reuse in a high throughput scenario.

This is where Sign Live! CC pooling support comes into play. You can provide a virtual pooled device to your security application client, backed up by a rack of card readers and mass signature smartcards. Sign Live! CC allows for the card reader administration and selection, spread over multiple security pools based on your selection and load balancing criteria. This way you get a high availability, high performance security operation environment.

7.1.2 Signing with Pools

7.1.2.1 Smartcard signature device pools

7.1.2.1.1 Overview

The smartcard signature device pool is a collection of real smartcard signature devices that have an active connection to the card and hold an authenticated transaction to the card application. This way a signature operation can be performed very efficient.

7.1.2.1.2 Create a signature device pool

First a pool must be declared that holds the concrete smartcard devices used for signing. The pool itself is implemented as a standard processor, so all standard ways of creating one are available.

The maximum number of signature pools available may be constrained by license restrictions.

The recommended way is to use an instrument declaration, either manually in an instrument file or via the interactive GUI support in the “Signature Pool Console”.

FACTORY

To launch a smartcard signature pool you use the standard processor launcher extension point described in the “*Developer’s Guide*” chapter “Working with processors”. The processor factory to be used is *de.intarsys.stage.security.device.smartcard.signature.SmartcardDigestSignerPoolFactory*. The *autoStart* flag must be *true* in this configuration scenario.

ARGUMENTS

Name	Description
maxActive	<p>The maximum number of active devices in the pool. This is the number of devices concurrently used in a multithreaded environment for signing.</p> <p>This may be constrained by license restrictions</p>
maxPoolSize	<p>The maximum number of devices in the pool. This is the number of devices that are available for use in the pool. This number must be greater or equal to <i>maxActive</i>. If the number is greater, the pool holds open devices in reserve for signing operations, for example when one of the other devices expires and the operator needs to re-enter the PIN.</p>
maxUptime	<p>This value constrains the maximum lifetime of the pooled device to the amount specified in minutes. After device expiration it is removed from the pool and a new one must be created and initialized depending on the pool configuration.</p> <p>A <i>maxUptime</i> of “-1” allows an unlimited use of the device.</p> <p>The default is “-1”.</p>
maxUsageCount	<p>This value constrains the maximum number of times a device can be used from the pool before it expires. After device expiration it is removed from the pool and a new</p>

one must be created and initialized depending on the pool configuration.

A *maxUsageCount* of “-1” allows for any number of uses, but may still be restricted by hardware or smartcard constraints used in the pool.

The default is “-1”.

suspended

If *suspended* is *true*, the pool starts in an inactive mode and will not try to acquire devices.

The default is *false*.

timeout

This is the maximum time in milliseconds that a client application will wait for a device to be made available by the pool. If no device is available after the specified time, an error is thrown.

A *timeout* of “-1” will wait indefinitely. The default for this value is “-1”.

cardTerminalFilter

An expression selecting from the card terminals connected to the system. The pool will accept smartcards only from terminals matching the filter.

The filter is a regular expression that is matched against the card terminal name published by the system card terminal manager.

Example

```
name:(?i).*reiner.*;
```

This example will accept any card terminal that identifies itself as a “reiner” product using a case insensitive match pattern.

certificateFilter

An expression selecting from the signature certificates on the smartcard. The pool will accept only smartcard devices containing a matching signature certificate. The security application will be opened and the PIN is requested.

The syntax of the selector is the same as in the other certificate selection scenarios.

Example

```
Usage:qualified_signature;
```

Select the certificate marked as available for a qualified signature.

```
Usage:signature;
```

Select the certificate marked as available for a signature operation.

```
SerialNumber:8139571262270123122;
```

Select the certificate with the serial number "8139571262270123122".

7.1.2.1.3 Create a signature pool via API

You can create a signature pool via the standard processor API.

Example:

```
var pool = Processor.callArgs(
  "de.intarsys.stage.security.device.smartcard.signature.SmartcardDigestSignerPoolFactory"
,
  {
    id: "mypool",
    certificateFilter: "Usage:qualified_signature"
  }
);
```

The pool returned is automatically registered upon start with the pool registry. If you don't want the pool to be active immediately, you can add the *suspend* argument.

7.1.2.1.4 Access a signature device pool

All signature device pools are registered with the DigestSignerPoolRegistry that can be accessed by a VM singleton *de.intarsys.stage.security.device.pool.signature.DigestSignerPoolRegistry.get()*.

A dedicated pool can be accessed using *lookupPool*. The return value is a processor instance of type *de.intarsys.stage.security.device.pool.signature.IDigestSignerPool*.

Example

```
var registry =
  Packages.de.intarsys.stage.security.device.pool.signature.DigestSignerPoolRegistry.get()
;
var pool = registry.lookupPool("mypool");
```

7.1.2.1.5 Suspend a pool

A pool can be suspended using *suspend*. All pooled inactive connections and resources are closed and released. Active devices are released as soon as they are returned to the pool. The pool will not acquire new resources until it is resumed.

Example

```
var pool = registry.lookupPool("mypool");
pool.suspend();
```

7.1.2.1.6 Resume a pool

A pool can be resumed using *resume*. The pool will start to acquire new resources according to its configuration.

Example

```
var pool = registry.lookupPool("mypool");
pool.resume();
```

7.1.2.1.7 Stop a pool

A pool can be stopped using *stop*. The pool will first suspend its activities (see *suspend*) and finally deregister itself. The pool is no longer available for any interaction.

Example

```
var pool = registry.lookupPool("mypool");
pool.stop();
```

7.1.2.2 The signature pool device

7.1.2.2.1 Overview

The signature pool device is a reference to one of the pooled real devices that can be used by a client application to perform a signature operation. Its API behavior and semantics is just like that of a “keystore device” or a “smartcard device”. It is implemented by *de.intarsys.stage.security.device.pool.signature.PooledDigestSignerFactory*.

FACTORY

de.intarsys.stage.security.device.pool.signature.PooledDigestSignerFactory

ARGUMENTS

Name	Description
id	Select the pool whose shared devices should be used. The pool must be registered.
attributeCertificates	Select a single or a list of attribute certificates, which target the signer certificate and refine its usage in this situation. The usage of this parameter is optional.

This is a polymorphic argument that can accept a single or a list of

`de.intarsys.security.cert.IX509Certificate`

`de.intarsys.security.cert.IX509CertificateSelector`

String identifying a certificate. In this string you can use the **SerialNumber**, Subject and Issuer to select the certificate from the store.

RESULT

The processor returns an object of type `byte[]`.

EXCEPTIONS

The processing may raise exceptions

7.1.2.2.2 Example

You will not call a digest signer directly, so here no example for this scenario is given. A digest signer is used as a parameter to a signer method.

This is an example how to use the pooled device in the context of a document signer.

```
var idoc = document;
var signature = Processor.callArgs(
    'com.cabaret.security.document.signing.DocumentSignerFactory',
    {
        document: idoc,
        digestSigner: {
            factory:
                'de.intarsys.stage.security.device.pool.signature.PooledDigestSignerFactory',
            args: {
                id: 'mypool'
            }
        }
    }
);
```

7.2 Session

7.2.1 Overview

A session provides the possibility to reuse a security application in the same security context. The security application is created at the beginning of the session. While the session is alive and the contained security application is valid, a client can access the device for security operations.

Upon security application expiration **or** session expiration the security application is disposed. Until then, the resources are kept by the session - so be sure to terminate your session as soon as possible.

The session lifecycle itself is bound to the protocol session lifecycle defined by the service framework (see Operators Guide, ch. Session Handling). This is why there exist two different expiration conditions for the security application: first, from the service level session that is defined by the service framework. Within that session, the security application is hosted. If the session expires, the security application is disposed. On top of that, the security application has a lifecycle itself – you can express explicit expiration conditions like “expire after 100 usages”. After the security application expires, it is removed from the (still active) service session.

7.2.2 Opening a session

FACTORY

A security application can be opened in a session explicitly using *de.intarsys.stage.security.device.session.signature.SessionDigestSignerOpenerFactory*. The main argument to this call is *digestSigner*, exactly the same as when you are calling a document signer directly. They define the security application to be cached in the session.

The security application created by this call is bound to the service session.

An example for lazy opening is given in the "sign" chapter.

There can only be a single security application of a given type in a session at the moment.

ARGUMENTS

Name	Description
digestSigner	The digest signer instance specification to be used within the session
expire...	Explicit expiration conditions for the security application (not the session), see chapter below

EXCEPTIONS

If the session can't be created an exception is thrown.

7.2.2.1 Example

```
var session = Processor.callArgs(  
    "de.intarsys.stage.security.device.session.signature.SessionDigestSignerOpenerFactory",  
    {  
        digestSigner: {  
            factory: "com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",  
            args: {  
                // some more device specific arguments  
            }  
        }  
    }  
);
```

The result of this call is the service session - when calling from remote this is serialized to its unique id.

7.2.3 Signing on a session

PROCESSORFACTORY

To use a security application from the session for signing, you request a digest signer with the processor factory
de.intarsys.stage.security.device.session.signature.SessionDigestSignerFactory.

If the session is opened explicitly as described above, there are no “digestSigner.args” for the call. Everything was provided when the session was created, the security application in the session is selected.

If you are using lazy session creation, the “real” security application used must be described nested in the “digestSigner.args” of the session security application.

7.2.3.1 Example (from session context)

```
Processor.callArgs(  
    "..some document signer...", {  
        // ..document signer arguments as needed...  
        digestSigner: {  
            factory:  
                "de.intarsys.stage.security.device.session.signature.SessionDigestSignerFactory",  
            args: {  
            }  
        }  
    }  
);
```

7.2.3.2 Example (lazy opening)

```
Processor.callArgs(  
    "..some document signer...", {  
        // session arguments  
        _session: {  
            id: "my unique session id",  
            lazy: true  
        },  
        // ..document signer arguments as needed...  
        digestSigner: {  
            factory:  
                "de.intarsys.stage.security.device.session.signature.SessionDigestSignerFactory",  
            args: {  
                // ...some more session specific arguments  
                digestSigner: {  
                    factory:  
                        "com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",  
                    args: {  
                        // some more device specific arguments  
                    }  
                }  
            }  
        }  
    }  
);
```

7.2.4 Closing a session

PROCESSORFACTORY

If you no longer need a session you should close it to release its resources. This is done using

de.intarsys.stage.security.device.session.signature.SessionDigestSignerCloserFactory.

The call has no arguments and terminates the security application within the session associated with the service call.

7.2.4.1 Example

```
var session = Processor.callArgs(
  "de.intarsys.stage.security.device.session.signature.SessionDigestSignerCloserFactory",
  {
  }
)
```

7.2.5 Expiration

To ensure a session will be cleaned up even in case of malfunction and to ensure timing restrictions on the session, you can define expiration conditions.

The conditions are checked on every use of the session device and on a regular base by the session registry itself.

After an expiration condition is met, the session device no longer can be used.

The expiration is defined as an argument to the session creation. The following expiration conditions are available.

Name	Description
expire.type	after: The session expires after "expireValue" milliseconds, regardless of the last use.
	at: The session expires exactly at the absolute timemark "expireValue" in milliseconds since midnight, January 1, 1970 UTC
	timeout: Expire after "expireValue" milliseconds of inactivity.
	usage: Expire after "expireValue" successful uses of the device in the session.
	never: This session will never expire.
expire.value	The numeric value to be used in evaluating the expiration condition, interpreted in terms of "expireType"

If no expiration condition is defined, the default expiration of "expire.type=timeout; expire.value=300000" (5 minutes timeout) is used.

8. Monitoring

8.1 Overview

In many scenarios Sign Live! CC is running in some headless or server mode. Crypto operations are performed in a service container, requested via file system listener, web service requests or other non interactive event sources.

Without a GUI or someone to monitor the console interactively you need some other means of monitoring your application. Sign Live! CC provides enterprise level monitoring by the propagation of its state via JMX, the Java monitoring standard.

This way you can use any standard tool, from Hyperic to Nagios and even jConsole to get informed about the current application state.

If you have an unattended scenario but still do not have established enterprise monitoring standards, you can still benefit from the monitoring framework: builtin with Sign Live! CC you can express notification forwarding for urgent events to your operators email account.

The domain of all Sign Live! CC MBeans is

```
de.intarsys.signlive
```

8.2 Pool MBean

DESCRIPTION

Every pool instance is represented by a single MBean.

OBJECT NAME

The object name is built from the application domain and the following properties.

Name	Description
type	The type string for the MBean. This is "signerpool" for a pool of signature devices
id	The id of the signature pool as defined in your configuration setting

ATTRIBUTES

This is the list of attributes published by the MBean.

Name	Description
state	The current activitiy state of the pool 0 = stopped 1 = started 2 = suspended
pending	The number of pending requests. This is the number of requests that is waiting for a device.
poolSize	The number of devices available in the pool (active and inactive)

NOTIFICATIONS

The list of notifications triggered by the MBean.

Name	Description
target.started	Sent when the pool is started.
target.suspended	Sent when the pool is suspended.
target.resumed	Sent when the pool is resumed.

target.stopped	Sent when the pool is stopped.
target.error	The pool encountered an error condition
target.noDevice	A “checkout” request was encountered when no device is registered
device.limitWarn	This is sent when you requested a notification upon reaching one of the configured limits for the device.
device.limitReached	This is sent when the device has reached its limit and can no longer be used.
device.authenticationRequired	The device needs authentication
device.authenticationFinished	The device is successfully authenticated
device.authenticationFailed	The device authentication failed
jmx.attribute.change	A pool attribute has changed (AttributeChangeNotification) <ul style="list-style-type: none"> poolSize

Example event scenario:

```
target.started

# insert valid card & enter PIN
device.authenticationRequired
device.authenticationFinished
jmx.attribute.changed ( PoolSize )

# insert valid card & enter PIN failure
device.authenticationRequired
device.authenticationFailed

# checkout a device when none is registered
target.noDevice

# remove card
jmx.attribute.changed ( PoolSize )

# suspend & resume
target.suspended
target.resumed

target.stopped
```

8.3 Notification String expansion

You can use string expansion for defining textual content in Sign Live! CC JMX notification sinks (both Mail and Message Boxes). Besides the basic application string variables you can access the current notification using “notification”. The following properties are provided:

Name	Description
message	The JMX notification message
sequenceNumber	The sequence number
timestamp	The time stamp
type	The notification type
attributeName	The name of the attribute changed (if this is an AttributeChangeNotification)
oldValue	The old attribute value (if this is an AttributeChangeNotification)
newValue	The new attribute value (if this is an AttributeChangeNotification)

9. Smartcard Tools

9.1 Overview

Smartcard security function are usually protected by some form of a secret, shared only between the card holder and the smartcard itself. This secret knowledge equates to either a PIN (Personal Identification Number) or a password, where a PIN contains just numbers and a password can consists of letters, numbers and symbols. Both forms are used by common smartcards. The following tool processors hide any encoding details and thus combines PINs and passwords in the form of a credential.

In the following paragraphs two important tools are presented, that are needed to successfully work with smartcards:

A credential changer - changes or initializes PINs and passwords

A credential resetter - resets blocked PINs and thus blocked security functions

9.2 Credential Changer

9.2.1 Overview

The CredentialChanger can change any kind of credential, like PINs, passwords, transport and resetting credentials. On top of that, it can initialize resetting credentials, if they are supported by the smartcard.

Transport credentials are preinitialized credentials for a security function which must be changed before the protected functions can be used the first time. A transport credential can only be changed once.

Resetting credentials are commonly known as PUKs (Personal Unblocking Key). Their purpose is to reset an associated credential if it is blocked and thus cannot be used anymore to authorize a smartcard security function.

PROCESSOR FACTORY

de.intarsys.stage.security.device.smartcard.tools.CredentialChangerFactory

ARGUMENTS

Name	Description
credentialIdentifier	The <i>credentialIdentifier</i> uses a certificate identifier to select it's associated credential. See <i>Smartcard signing device</i> , parameter <i>signerIdentifier</i> for a detailed description. If a credential is associated to more than one card holder certificate, it doesn't matter which card holder certificate is used to identify the credential.
secret	The current value of the selected credential for the security function, specified as a String.
newSecret	The new value for the selected credential for the security function, specified as a String.
resettingSecret	Use this parameter to select the resetting credential associated with the security function identified by <i>credentialIdentifier</i> .
resettingNewSecret	Use this parameter to select the new value for the resetting credential associated with the security function identified by <i>credentialIdentifier</i>

RESULT

The processor does not return any result, if it succeeds.

EXCEPTIONS

The processing may raise exceptions.

9.2.2 Examples

9.2.2.1 Zero pin or transport pin change

```
Processor.callArgs(  
  "de.intarsys.stage.security.device.smartcard.tools.CredentialChangerFactory",  
  {  
    credentialIdentifier: 'SerialNumber:8139571262270123122;',  
    secret: "000000",  
    newSecret: "111111",  
  }  
);  
Processor.callArgs(  
  "de.intarsys.stage.security.device.smartcard.tools.CredentialChangerFactory",  
  {  
    credentialIdentifier: 'SerialNumber:8139571262270123122;',  
    secret: "12345",  
    newSecret: "111111",  
  }  
);
```

Depending on the smartcard operating system, defining the initial pin uses a “Zero Pin” or “Transport Pin” method. Both methods are presented in the example above.

9.2.2.2 Pin change

```
Processor.callArgs(  
  "de.intarsys.stage.security.device.smartcard.tools.CredentialChangerFactory",  
  {  
    // select a certificate to identify PIN  
    credentialIdentifier: 'SerialNumber:8139571262270123122;',  
    secret: "123456",  
    newSecret: "111111",  
  }  
);
```

This example changes the credential value for the smartcard application represented by the selected certificate. Be aware that most smartcards share credentials across applications - for example, most often this is true for decryption and authentication functions, thus the credential change operation above would affect both.

9.2.2.3 Activate resetting credential for a credential

```
Processor.callArgs(  
  "de.intarsys.stage.security.device.smartcard.tools.CredentialChangerFactory",  
  {  
    credentialIdentifier: 'SerialNumber:2701231228139571262;',  
    secret: "123456",  
    newSecret: "111111",  
    resettingNewSecret: "88888888"  
  }  
);
```

This example changes the credential value for the smartcard signature function and enables its associated resetting credential with the new value “88888888”.

This may not be supported on all smartcards.

9.2.2.4 Change resetting credential

```
Processor.callArgs(
    "de.intarsys.stage.security.device.smartcard.tools.CredentialChangerFactory",
    {
        // use the "signature" certificate the select the signature credential
        credentialIdentifier: 'SerialNumber:2701231228139571262;',
        resettingSecret: "11111111",
        resettingNewSecret: "22222222"
    }
);
```

This call changes the resetting credential for the credential associated with the application identified by the certificate selector.

On some card operating systems, prior to changing the resetting credential you must activate it.

This may not be supported on all smartcards.

9.3 Credential changer wizard

As usual there is also an interactive, wizard based interface for changing the credentials.

WIZARD FACTORY

de.intarsys.stage.security.device.smartcard.tools.CredentialChangerWizardFactory

ARGUMENTS

None

RESULT

None

9.3.1 Examples

```
Wizard.callArgs(
    'de.intarsys.stage.security.device.smartcard.tools.CredentialChangerWizardFactory', {
});
```

This calls the credential change wizard. All arguments are preset to the values stored in the preferences from the last call.

9.4 Credential Resetter

9.4.1 Overview

The CredentialResetter resets a blocked credential. This application may not be accessible on all smartcard operating systems.

Prior to resetting a blocked credential, it may be necessary to activate the resetting credential itself (see above).

PROCESSOR FACTORY

de.intarsys.stage.security.device.smartcard.tools.CredentialResetterFactory

ARGUMENTS

Name	Description
credentialIdentifier	The <i>credentialIdentifier</i> uses a certificate identifier to select it's associated credential to reset. See <i>Smartcard signing device</i> , parameter <i>signerIdentifier</i> for a detailed description. If a credential is associated to more than one card holder certificate, it doesn't matter which card holder certificate is used to identify the credential.
resettingSecret	The current value of the resetting credential, specified as a String.
newSecret	The new value for the application credential after resetting, specified as a String.

RESULT

The processor does not return any result, if it succeeds.

EXCEPTIONS

The processing may raise exceptions.

9.4.2 Examples

```
Processor.callArgs(  
    "de.intarsys.stage.security.device.smartcard.tools.CredentialResetterFactory",  
    {  
        // use the "signature" certificate to select the signature credential  
        credentialIdentifier: 'SerialNumber:2701231228139571262;',  
        resettingSecret: "88888888",  
        newSecret: "111111",  
    }  
);
```

This example resets the credential value for the smartcard signature function. It uses the "signature" certificate to identify the associated credential to be the one to reset, using the resetting code "88888888" and giving the credential the new value "111111" after the reset.

9.5 Credential reset wizard

As usual there is also an interactive, wizard based interface for resetting the credentials.

WIZARD FACTORY

de.intarsys.stage.security.device.smartcard.tools.CredentialResetterWizardFactory

ARGUMENTS

None

RESULT

None

9.5.1 Examples

```
Wizard.callArgs(  
    'de.intarsys.stage.security.device.smartcard.tools.CredentialResetterWizardFactory', {  
    });
```

This calls the credential reset wizard. All arguments are preset to the values stored in the preferences from the last call.