

Sign Live! CC Security Application Deployment Whitepaper

Februar 2022

intarsys GmbH

Sign Live! CC Security Application Deployment Whitepaper

Version 7.1

A whitepaper on security application deployment using the
service container framework

Preface

- Author and company

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

- 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

[_top](#)

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	13
1. Installation	15
1.1 Overview	15
1.2 Download	15
1.3 Installation	15
1.4 Licenses	15
1.5 Documentation	16
1.6 Demo code	16
1.6.1 ActiveX	16
1.6.2 HTTP	16
1.6.3 WebService	16
1.6.4 ...and some more	16
2. Web service calls	17
2.1 Overview	17
2.2 The console	17
2.3 Adding a web container	19
2.4 Adding a service	21
2.5 Start the container	24
3. HTTP "plain" client	25
3.1 Overview	25
3.2 Run the demo client	25
3.3 Examine the demo client	26

Content

4. SOAP client	27
4.1 Overview	27
4.2 Container configuration changes	27
4.3 Run the demo client	28
4.4 Examine the demo client	28
5. Configuration details	30
5.1 Overview	30
5.2 Static argument definition	30
5.3 Persistence	30
5.3.1 Preferences	30
5.3.2 Dynamic instruments	30
6. Alternate "signer" implementations	32
6.1 Overview	32
6.2 Smartcard pool	32
6.2.1 Overview	32
6.3 Swisscom AIS	32
6.3.1 Overview	32
6.3.2 Preferences	32
6.3.3 Service container	34
6.3.4 HTTP Demo client	36
6.3.5 SOAP Demo client	36
7. Shared library calls	38
7.1 Overview	38
7.2 Adding a local container	38
7.3 Adding a service	39
7.4 Start the container	40
7.5 The C-Style client	40
7.5.1 Overview	40
7.5.2 Launching	41
7.5.3 Calling	41
7.5.4 Method name	41
7.5.5 Arguments	41
7.5.6 Results	42
7.5.7 Exceptions	42
7.6 Using the demo code	42
7.7 Using individual launch arguments	43
7.7.1 nowakeonstart	43
7.7.2 profile	43
7.7.3 itdirs	43
7.8 Signature example	43
7.8.1 Service container instrument	43
7.8.2 Signature method	44
7.8.3 Launch application	44
7.8.4 Single document	44
7.8.5 Multi document	45
7.8.6 Document integrity	45

8. Advanced signature scenarios	46
8.1 Overview	46
8.2 Visible signatures	46
8.3 Visible signatures with dynamic field position and size	47
8.3.1 Embedded tags	47
8.3.2 Embedded dynamic tags	49
8.4 Timestamps	51
8.4.1 Overview	51
8.4.2 Select a timestamp service	51
8.4.3 Use a pre-registered timestamp service	52
8.4.4 Use an on-the-fly timestamp service	55

Introduction

Sign Live! CC provides a complex set of components that can be composed in a variety of ways as a standalone desktop application, embedded library or standalone server.

While most of the information is available in the documentation, information for some scenarios are spread over

- Developers Guide
- Operators Guide
- Security Application Developers Guide
- Online Help
- Demo code

This whitepaper is a "short track" handbook for deploying a complete security application service in Sign Live! CC .

Where possible, we strive for compatibility with the existing demo and snippet code, so that you can test your configuration immediately using one of the demo apps.

1. Installation

1.1 Overview

Sign Live! CC installation media exists for Windows, Linux and MacOS. For this paper we use the windows installation. For other environments you may need to adapt physical information like pathnames.

1.2 Download

You can download the application at

```
https://www.intarsys.de/download
```

1.3 Installation

Just start the installer and perform a standard installation.

Afterwards you should have a clean installation at

```
<program path>/Sign Live CC <version>
```

To start the application, simply start

```
bin/SignLiveCC.exe
```

in this folder.

1.4 Licenses

For some of the features covered in this paper you will need a valid license.

In this case you can request demo licenses at sales@intarsys.de.

A license is installed by simply

- drag & drop on the application
- copy into the "licenses" subfolder of the installation

1.5 Documentation

In addition to this paper you will find more documentation in the installation subdirectory "doc/en".

- The "**Readme**" gives an overview and information on system requirements.
- "**Operators Guide**" will give detail information on installation and configuration of the application.
- "**Developers Guide**" will help you out if you need to implement new features or use an API of the application.
- "**Security Applications Developers Guide**" will enumerate all features and parameters available for, well, security applications.

Even more information may be available in the online help.

1.6 Demo code

Code snippets and ready to use code can be found in the SDK subdirectory of the installation.

For most protocols and features here is a prefabricated server & client configuration.

1.6.1 ActiveX

You can launch the application as an Active X container on Windows, allowing for example easy integration into scripting languages or other native Windows applications.

1.6.2 HTTP

Here you find plain HTTP (form URL encoded) examples.

1.6.3 WebService

Here are the SOAP based examples

1.6.4 ...and some more

This is left for your exploration at the moment.

2. Web service calls

2.1 Overview

The "service framework" is in depth treated in the "Operators Guide".

For now you must know that it is a protocol independent abstraction for a service to be executed when an event is triggered.

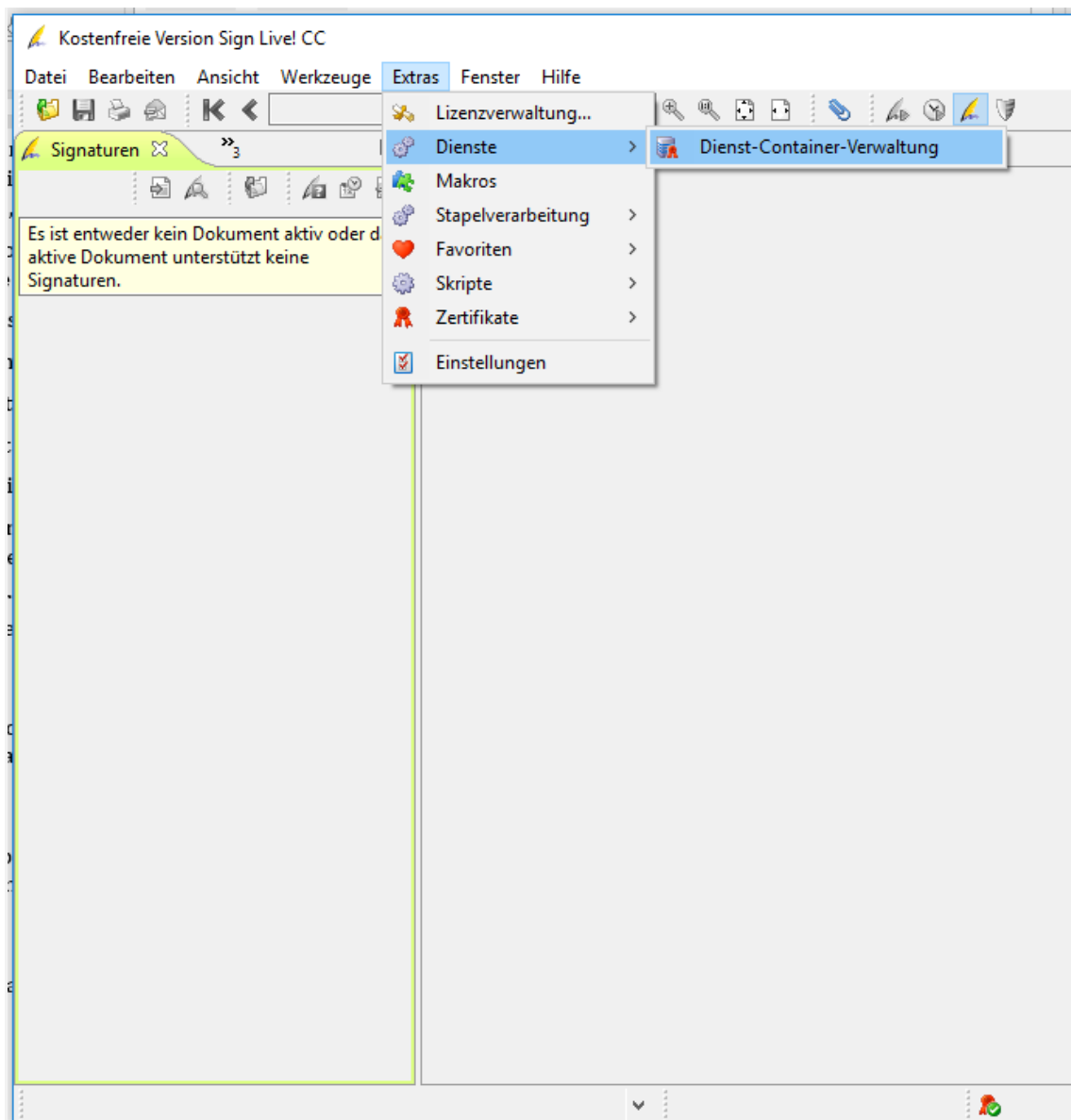
Event sources can be for example

- A file is found in a directory
- A message is received on a message queue
- A printjob is received via the printer queue
- A network message arrived
- A timer based event

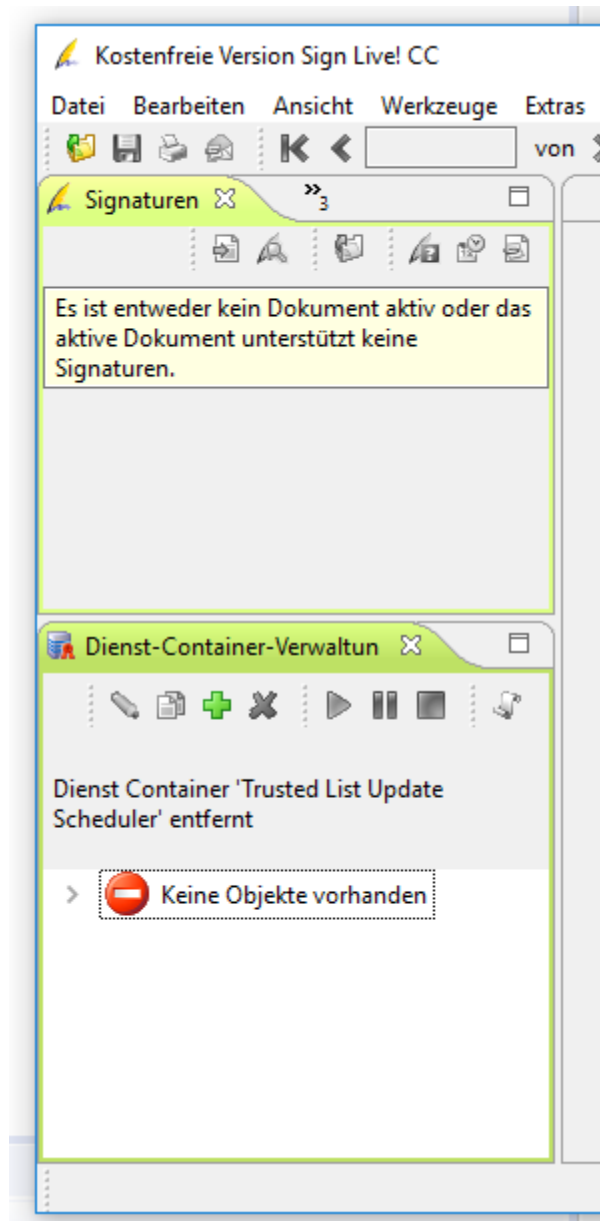
For our paper we are interested in setting up a service that can be accessed via a HTTP protocol.

2.2 The console

If the service management console is not already available after starting the application, you can add it by calling "Extras->Services->Service Container".

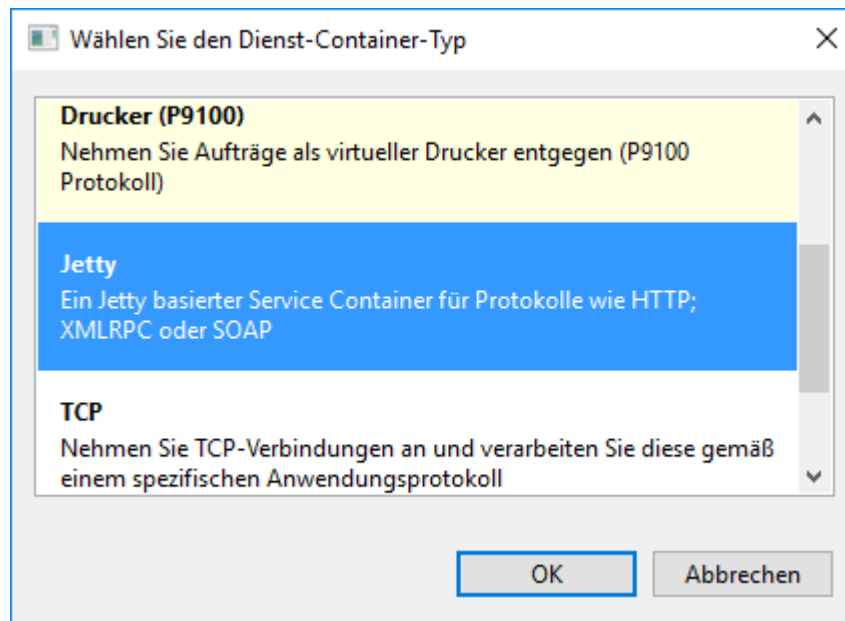


Eventually you will see the service container console in the lower left

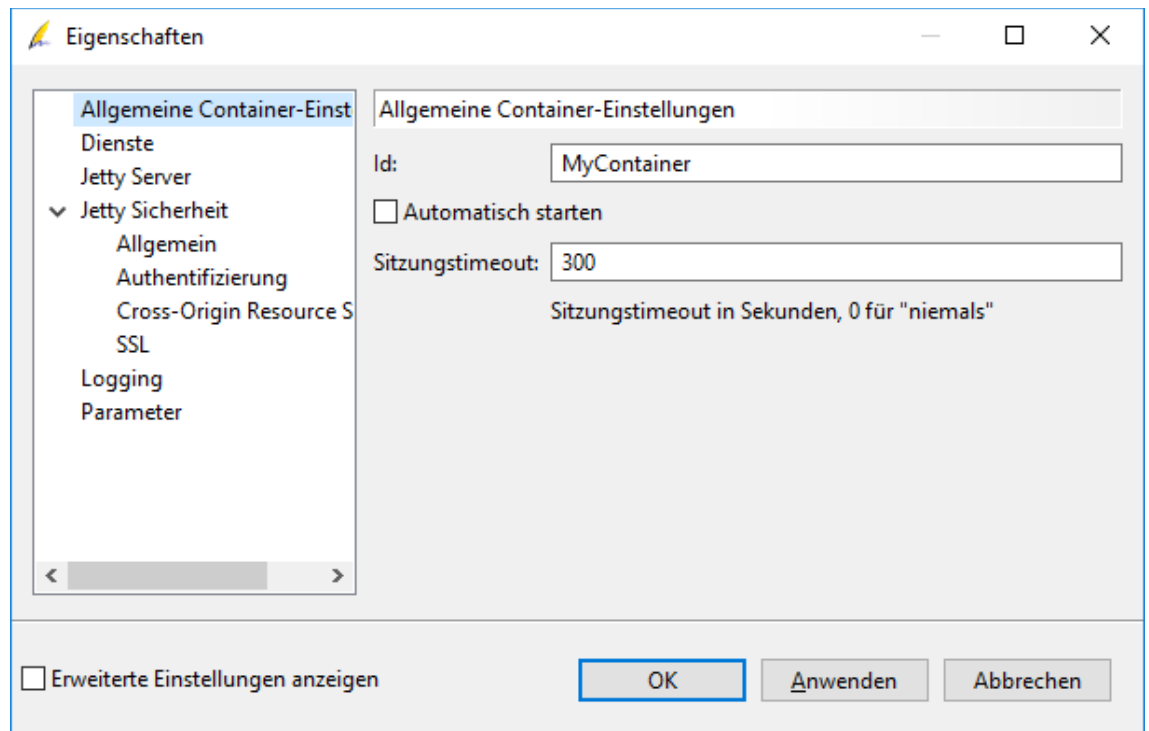


2.3 Adding a web container

To add a new container, press the plus sign. In the next dialog you should select "Jetty" to create an embedded Jetty container.

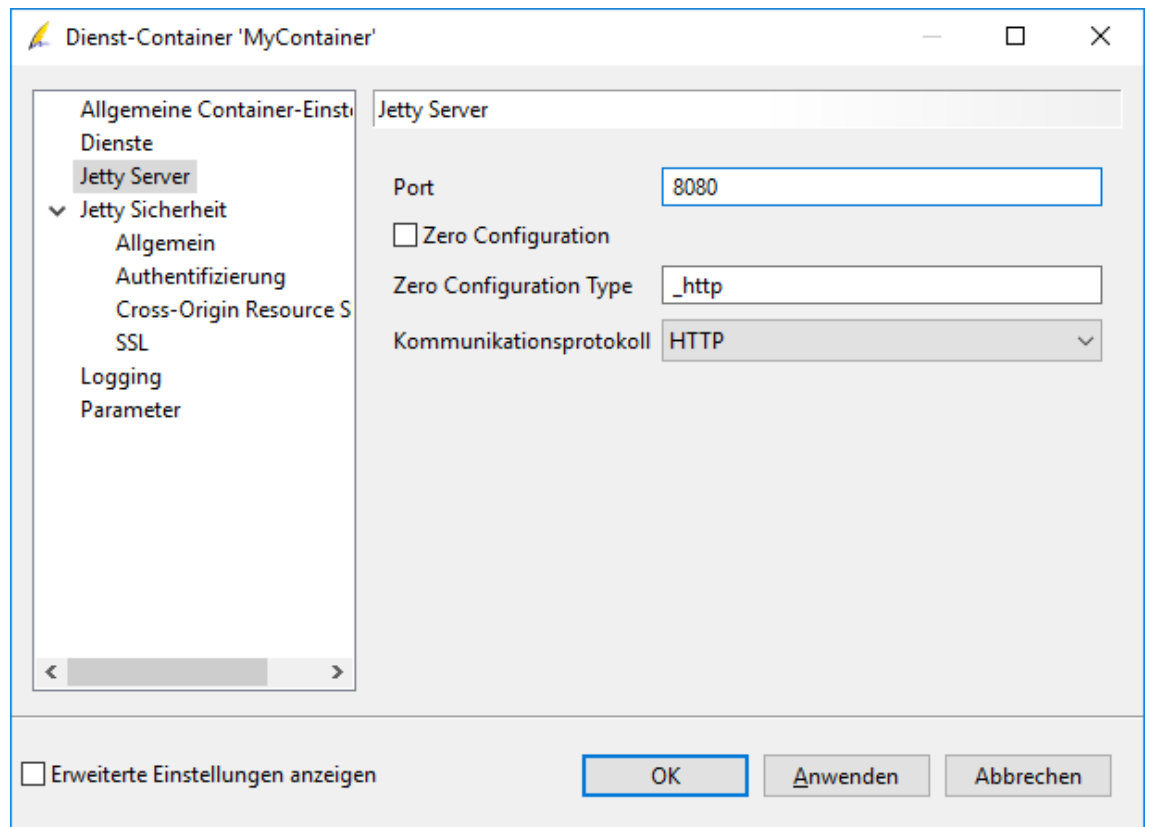


Now you can edit the properties for the new container.



There's a lot of stuff you can edit here - but for our simple purpose we recommend only to set a new "id", say "MyContainer".

If you want to be able to call the service using one of the demo clients, you should switch to port 8080 for the listener.

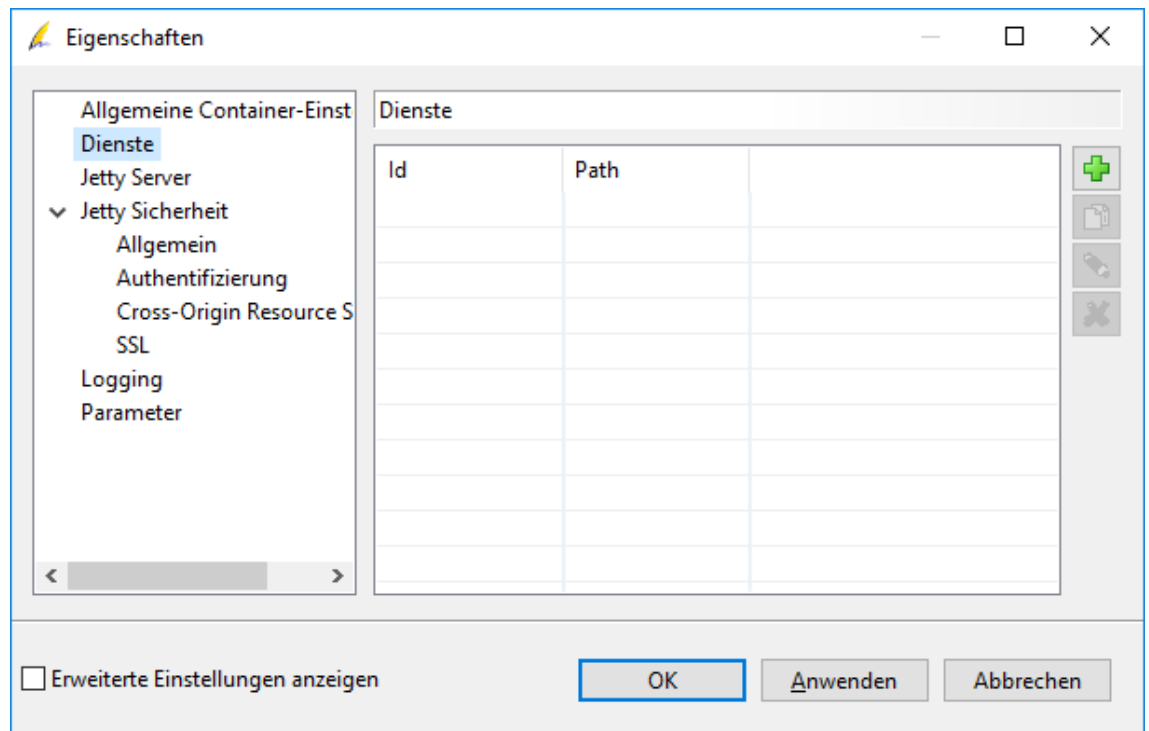


That's all for now, we will now add a service.

2.4 Adding a service

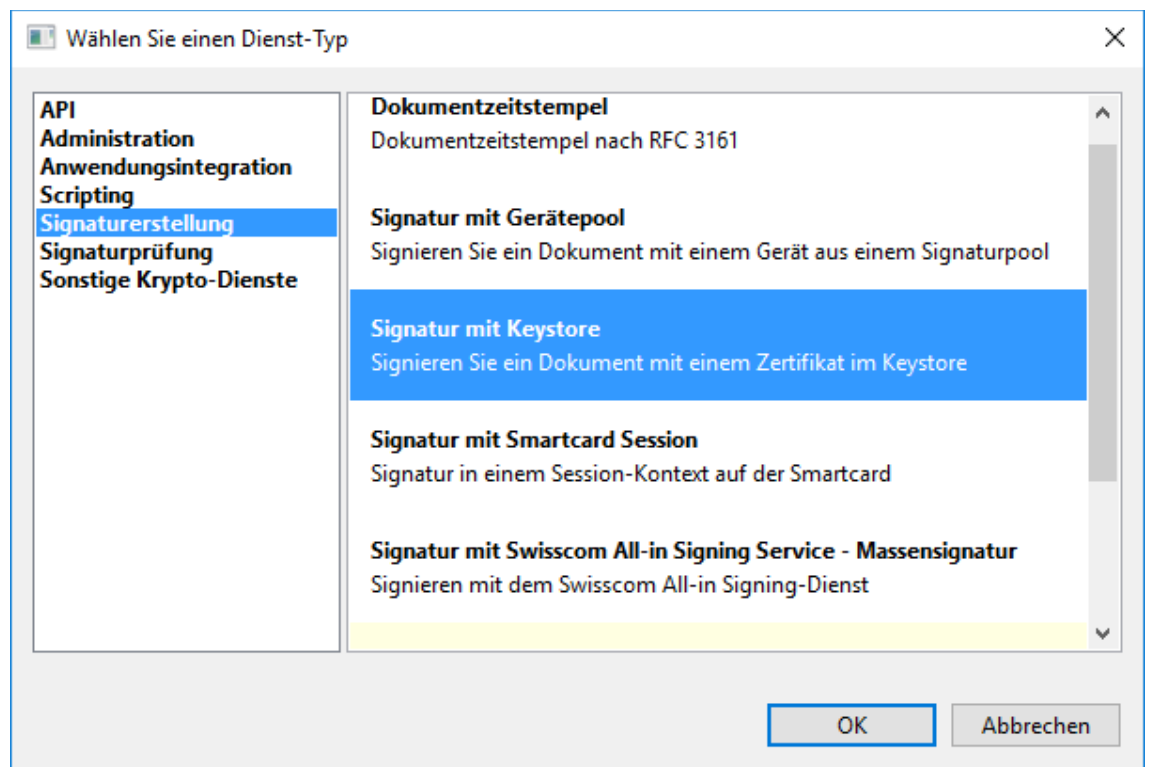
Adding a service can be done on the "Services" tab of the properties dialog.

Press the "plus" button here.



Again, you have bunch of predefined services you can deploy, from a complete signature creation or validation to completely free form scripted JavaScript code.

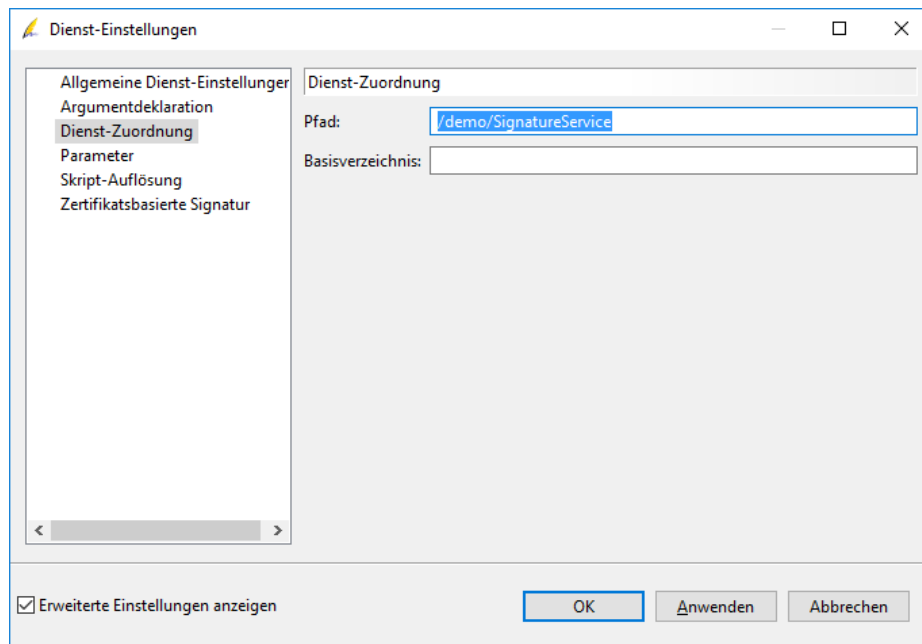
For a quick and dirty setup we select the "Signature creation" tab and then the "Signature with keystore" entry.



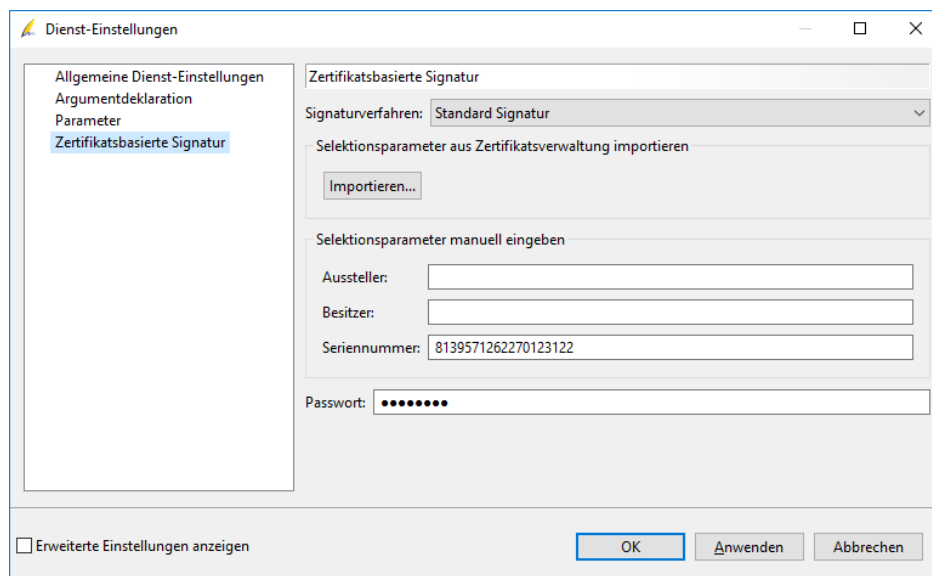
Here you edit the properties of the service. Give it a name (like "MyService").

Once again there is an additional step if you want to use the demo clients deployed with the application. Select "Show Extended Properties" in the lower left, then the "Service Mapping" tab. In the "Path" field you should enter

/demo/SignatureService



Now lets have a look at the "Certificate based signature" tab.

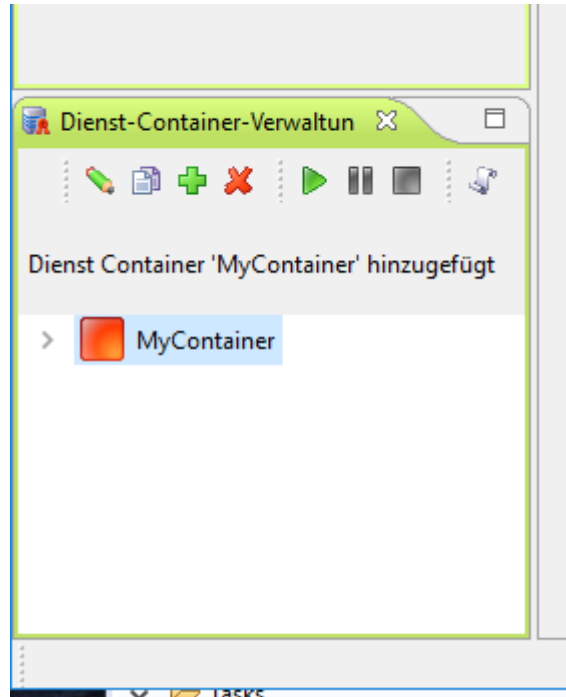


Here you can enter some details on the key to be used for signing (the key can be selected via dynamic arguments, too, for sure).

For your convenience, a plain demo key is already selected, so you can press "OK".

2.5 Start the container

Now you should have an inactive container with a single service.



You can start (and later stop) it using the buttons in the local toolbar. After starting you can send a post request to your new service.

3. HTTP "plain" client

3.1 Overview

If you have followed the configuration up to here, we now have a running server at `http://localhost:80/` with a signature service at `/demo/SignatureService`.

Any request to this URL will be forwarded to the newly configured signature service.

3.2 Run the demo client

A matching demo client can be found in the

```
sdk/HTTP/demo/signature
```

subfolder of the installation.

If you have followed the guide, you should be able to simply start

```
bin/run_demo.bat
```

This will sign the document

```
data/doc_unsigned.pdf
```

and you will get a signed document at

```
<user profile>/doc_unsigned.result.pdf
```

There are some simple issues you may encounter here:

- You did not use the settings from the paper above (port 8080 and path `/demo/SignatureService`). Then you should adapt the settings, the batch file or the client source to match

- You have not a "full" installation (as is the default from our downloads). Then the embedded JRE is missing and accordingly the batch file files. Adapt the "java" reference in the batch file.
- You did not start the container - simply start it...
- You did not add a license. Depending on the scenario & platform this may lead to a complete failure or a message box that warns you that the result will have a prominent watermark. Simply add the license.

3.3 Examine the demo client

We will have a short look at the minimalistic client code here

```
HttpClient client = new DefaultHttpClient();
HttpPost method = new HttpPost(url.toString());
List<NameValuePair> nvps = new ArrayList<NameValuePair>();

FileInputStream filestream = new FileInputStream(inName);
byte[] doc = StreamTools.toByteArray(filestream);
String encodedDoc = new String(Base64.encode(doc));

// add parameters
nvps.add(new BasicNameValuePair("document.content", encodedDoc));
nvps.add(new BasicNameValuePair("document.name", inName));

UrlEncodedFormEntity entity = new UrlEncodedFormEntity(nvps, "UTF-8");
method.setEntity(entity);

// perform call
response = client.execute(method);
if (response.getStatusLine().getStatusCode() != HttpStatus.SC_OK) {
    // handle error
    return;
}

HttpEntity resultEntity = response.getEntity();
InputStream is = null;
OutputStream os = null;
File out = new File(outName);
try {
    is = resultEntity.getContent();
    os = new FileOutputStream(out);
    StreamTools.copyStream(is, false, os, false);
} finally {
    StreamTools.close(is);
    StreamTools.close(os);
}
```

First, we setup a standard apache http-client environment.

After reading and base64 encoding the file to be signed we add the only two form fields needed for the signature, "document.content" and "document.name".

After studying the "Security Applications Developers Guide" you can add much more sophisticated parameters here, but the idea stays the same.

We execute the POST and receive the signed document as a binary stream.

That's it.

4. SOAP client

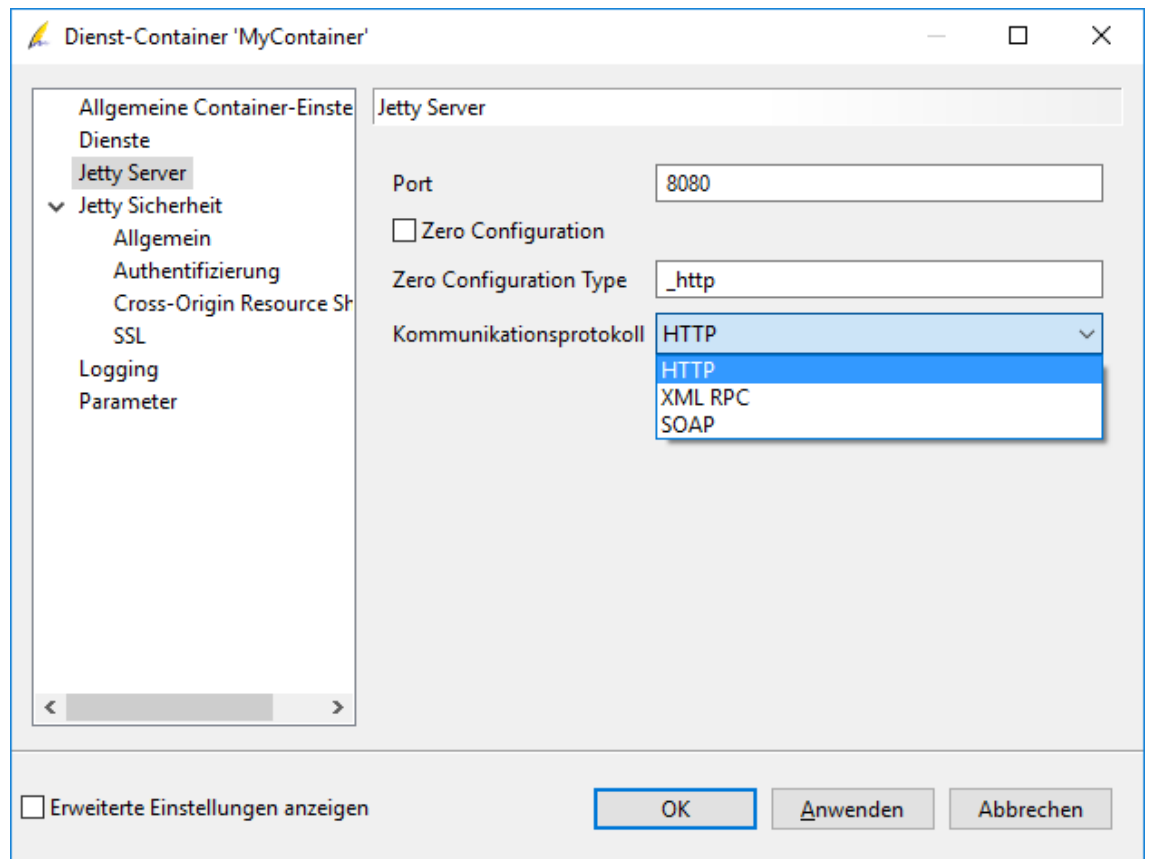
4.1 Overview

Now let's switch to a SOAP scenario. For many clients (such as SAP or Microsoft based products), SOAP is a builtin remote activation client, so usage of a SOAP container maybe even simpler than "plain HTTP".

4.2 Container configuration changes

To switch to the SOAP protocol you simply have to make some changes in the Jetty container configuration.

On the "Jetty Server" tab, switch the protocol field to SOAP.



That's it.

4.3 Run the demo client

Again we have a demo client at

```
sdk/WebService/demo/Signature
```

that can be launched with the batch file

```
bin/run_demo.bat
```

Everything else stays the same.

4.4 Examine the demo client

The code for the SOAP client is deployed, too - and even a little bit smaller.

```
JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
factory.setServiceClass(CallService.class);
factory.setAddress(new URL("http", hostname, port, servicename)
    .toExternalForm());
CallService client = (CallService) factory.create();

FileInputStream filestream = new FileInputStream(inName);
byte[] doc = StreamTools.toByteArray(filestream);

Map serviceArgs = new Map();
serviceArgs.put("document.content", doc); //$NON-NLS-1$
serviceArgs.put("document.name", inName); //$NON-NLS-1$

Map result = (Map) client.callArgs("sign", serviceArgs);
byte[] resultContent = (byte[]) result.get("content");

File out = new File(outName);
FileTools.write(out, resultContent);
```

This is boilerplate JAVA SOAP code, instantiating a stub for the

```
com.cabaret.api.webservice.CallService
```

interface and adding arguments, just like in the HTTP example.

You can request the WSDL when your service is running in a browser at

```
http://localhost:8080/demo/SignatureService/?wsdl
```

5. Configuration details

5.1 Overview

Now that we are able to create a container configuration we should have a closer look at the internals.

5.2 Static argument definition

5.3 Persistence

In many cases you will develop and test a configuration in a desktop installation using the GUI and later on deploy to a headless server. It is quite easy to reuse the desktop configuration in this case.

There are two information chunks you need to transfer.

5.3.1 Preferences

The application preferences are stored per user in the directory

```
<user>/.SignLiveCC_<version>/preferences
```

If you are unsure which preferences to copy, just take all.

5.3.2 Dynamic instruments

An "instrument" is the modularization primitive of the application. When you define a service, the outcome is an "instrument" containing the persisted configuration that represents and installs the service at runtime.

It is safe to use the GUI to define the instrument and then copy the result to a production environment in most cases.

Dynamic instrument definitions are stored at

```
<user>/SignLiveCC_<version>/instruments
```

The subdirectory

```
com.cabaret.service
```

holds all information about services you just configured.

6. Alternate "signer" implementations

6.1 Overview

In the initial setup we encountered a signature scenario based on soft certificates from a builtin keystore.

Now we present some alternate scenarios the are far more advanced and secure.

6.2 Smartcard pool

6.2.1 Overview

In certain scenarios an unattended use of smartcards is required. You can achieve this by defining a "smartcard pool" that may contain unlimited smartcard resources and a service that dynamically requests a smartcard from the pool for each signature event.

6.3 Swisscom AIS

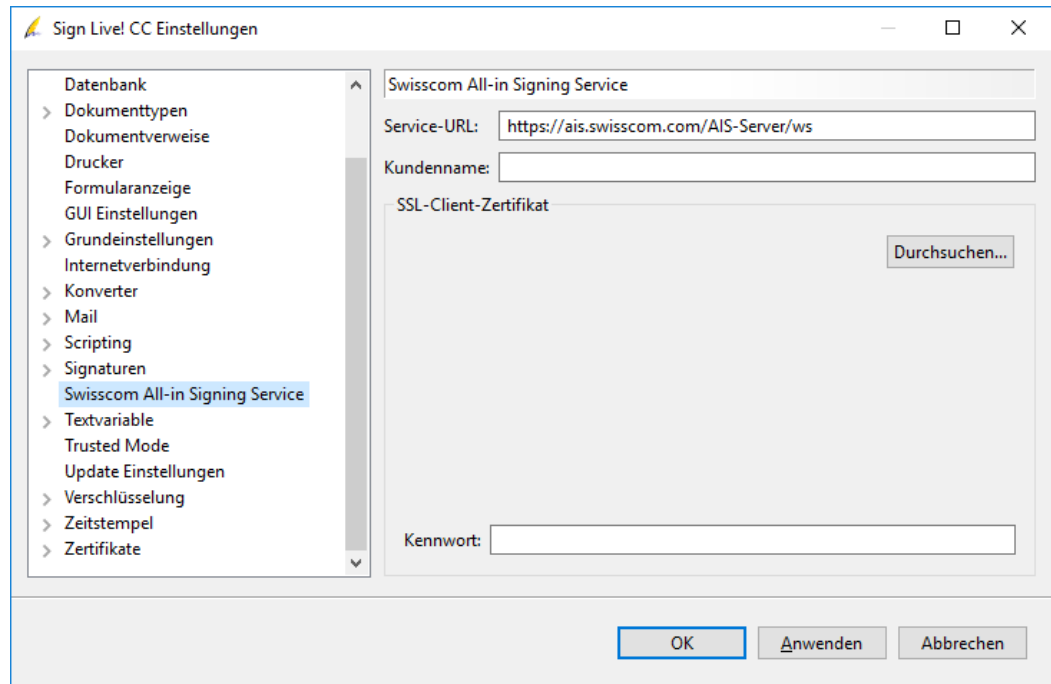
6.3.1 Overview

Swisscom AIS is a server side signing method that offers different processing flavors and security levels.

For using this service you need a contract with Swisscom, in turn you get the credentials we need for setting up the application.

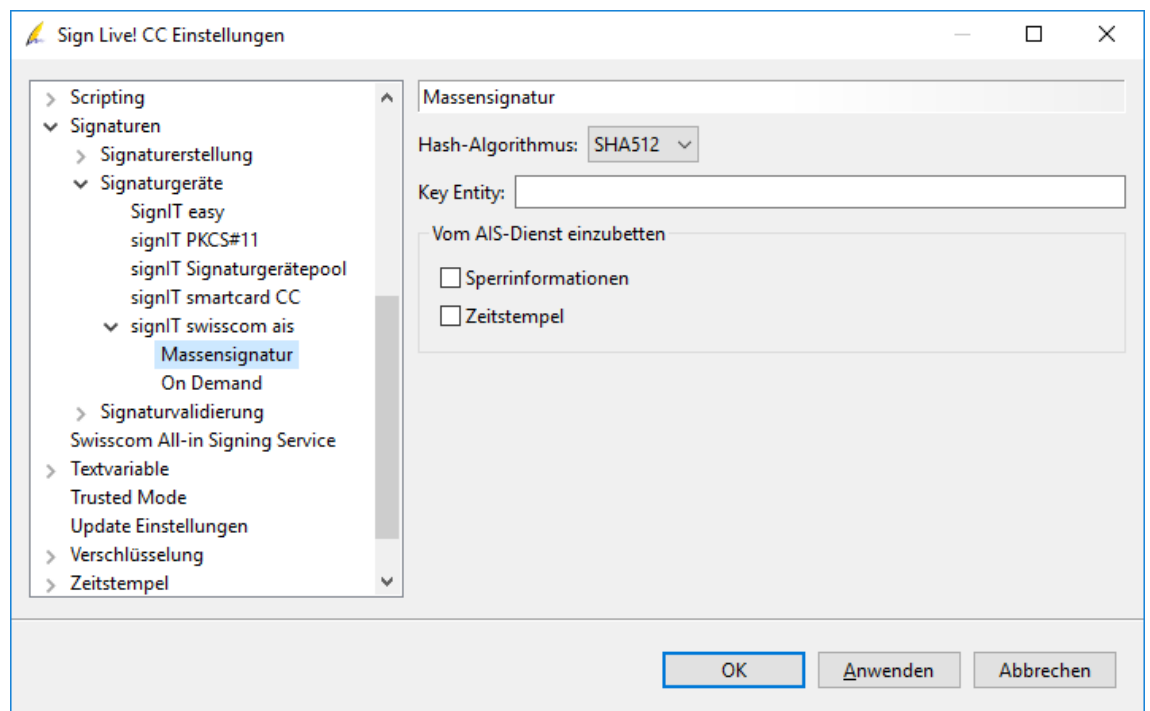
6.3.2 Preferences

The main preferences hold the service URL and the credentials for your account. You exchange this information with Swisscom and make it available here.

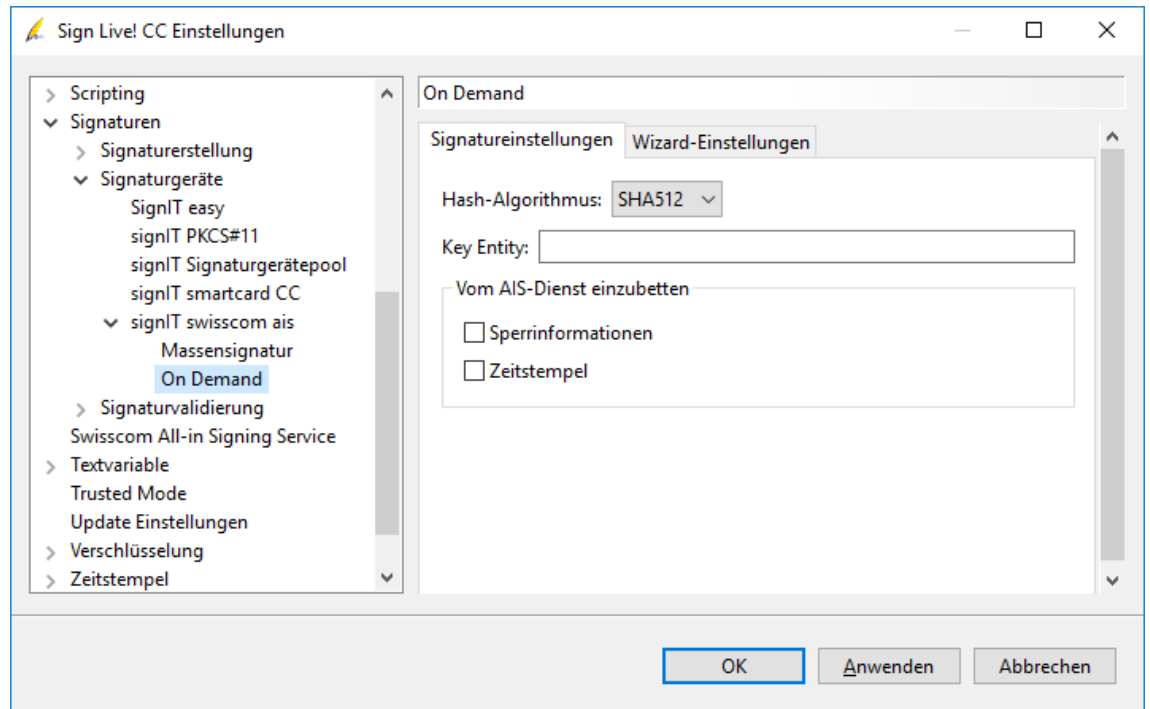


Next, you have preference pages for the "Mass signature" (static signature). Here you enter the name for the key entity that is handed to you by Swisscom.

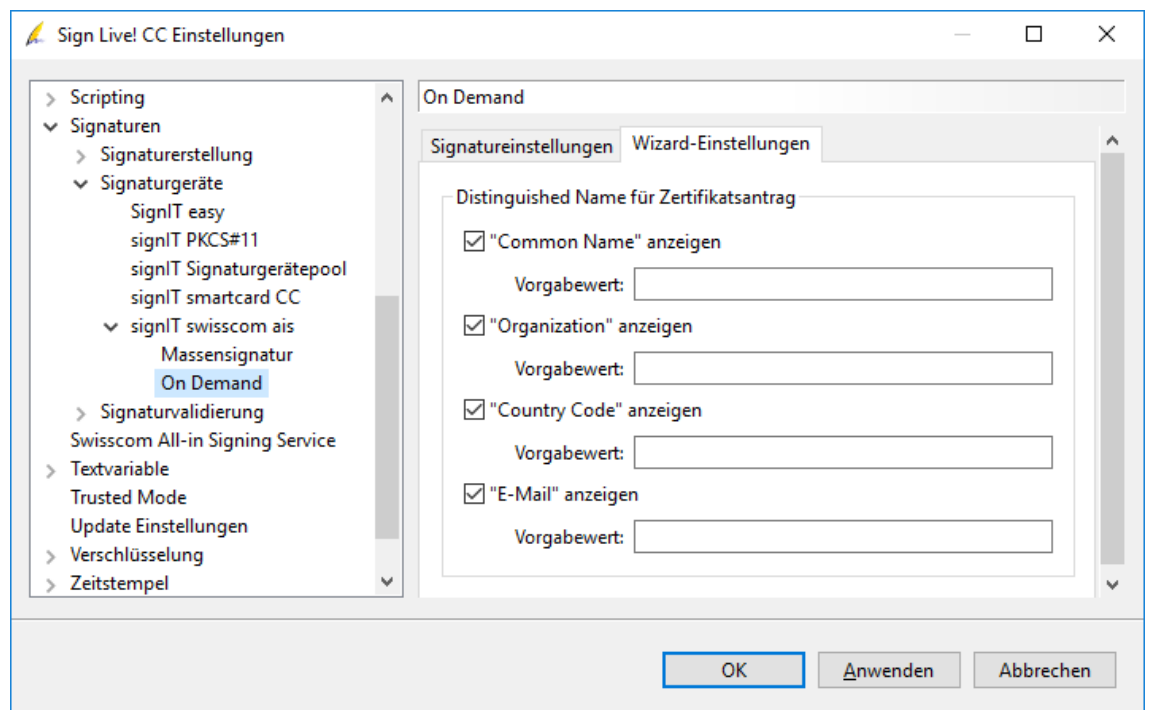
Optionally you can request that revocation lists or timestamps have to be embedded in the signature.



The preference page for "On Demand" signature looks much the same. You need the key entity and some optional information.



As the On Demand service needs a special distinguished name, you can make a "preset" on the "Wizard Settings" tab.

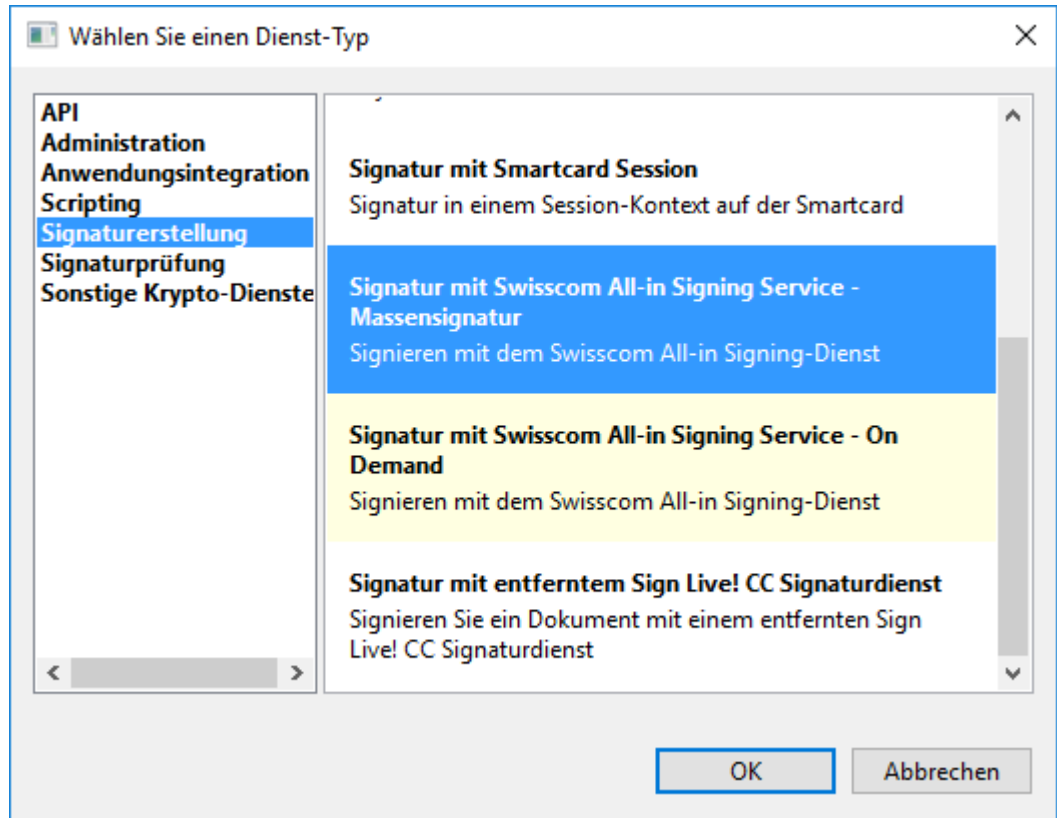


6.3.3 Service container

If you have a valid Swisscom account and entered your credentials correctly, you can now setup the container to call the AIS backend.

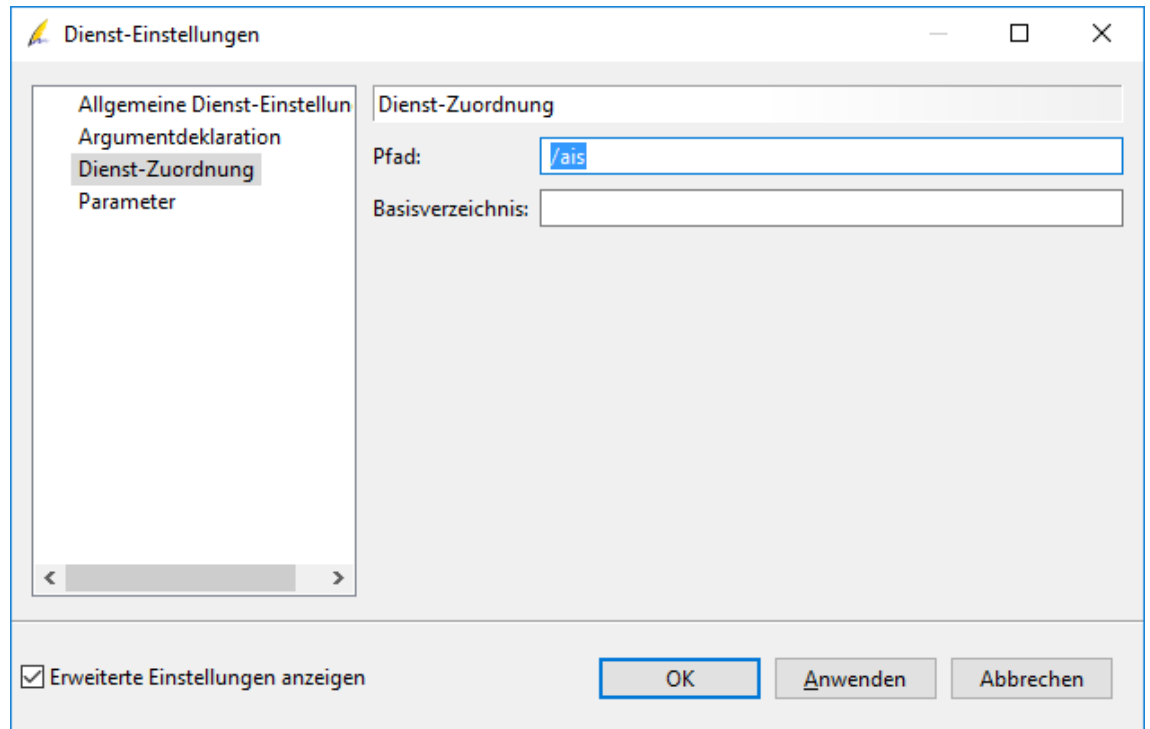
All you have to do is go back to the service container properties and remove the "old" keystore signature service, then add a new one.

You have to select here between the "Mass signature" and "On Demand" backend service.



The configuration process is the same for both service types.

If you want to use the demo client "as is", you should map the service to "/ais" by selecting "Show advanced settings" and enter the path.



No further settings are needed, so press "OK".

6.3.4 HTTP Demo client

The source for the demo is a little different from the other ones, as we have some additional required arguments for the call.

6.3.5 SOAP Demo client

Currently there is no SOAP version of the demo. Mixing the argument structure from the example above and the SOAP demo skeleton from other demos should be straightforward.

```
// prepare HTTP call
HttpClient client = new DefaultHttpClient();
HttpPost method = new HttpPost(url.toString());
List<NameValuePair> nvps = new ArrayList<NameValuePair>();

FileInputStream filestream = new FileInputStream(inName);
byte[] doc = StreamTools.toByteArray(filestream);
String encodedDoc = new String(Base64.encode(doc));

// add parameters
nvps.add(new BasicNameValuePair("document.content", encodedDoc));
nvps.add(new BasicNameValuePair("document.name", inName));
nvps.add(new BasicNameValuePair("digestSigner.args.keyEntity", keyEntity));
nvps.add(new BasicNameValuePair("digestSigner.args.distinguishedName",
distinguishedName));
// optional parameters
// digestSigner.args.stepUpLanguage
// digestSigner.args.stepUpMessage
// digestSigner.args.stepUpMSISDN

UrlEncodedFormEntity entity = new UrlEncodedFormEntity(nvps, "UTF-8");
method.setEntity(entity);

// perform call
response = client.execute(method);
if (response.getStatusLine().getStatusCode() != HttpStatus.SC_OK) {
    // handle error
    return;
}

HttpEntity resultEntity = response.getEntity();
InputStream is = null;
OutputStream os = null;
File out = new File(outName);
try {
    is = resultEntity.getContent();
    os = new FileOutputStream(out);
    StreamTools.copyStream(is, false, os, false);
} finally {
    StreamTools.close(is);
    StreamTools.close(os);
}
```

The "digestSigner" (this is the one who creates the cryptographic primitive) needs additional arguments, at least

- keyEntity
- distinguishedName

Depending on your scenario you may have to add

- stepUpLanguage
- stepUpMessage
- stepUpMSISDN

More information on the argument list to this service you can find in the "Security Applications Developers Guide".

7. Shared library calls

7.1 Overview

As already mentioned, the service container framework abstracts from the client and protocol that causes a service execution.

All clients for the application can be configured and managed in similar way.

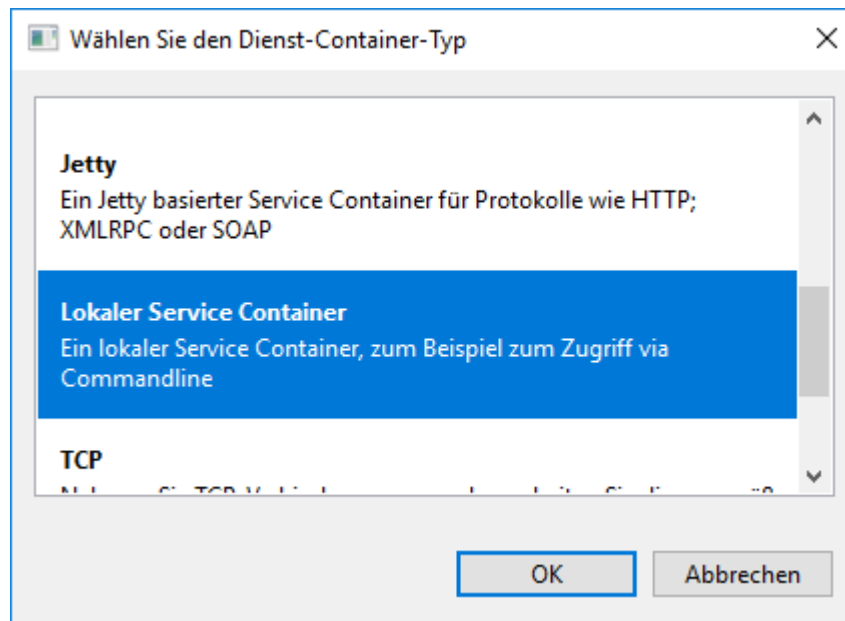
One of the more challenging integrations consists in loading out application in your process space and calling services directly from your C-style application.

This is done basically by wrapping the Java VM and forward calls via the native method specification. The good news is with our abstraction you do not need to worry about most of the outfalls of this approach.

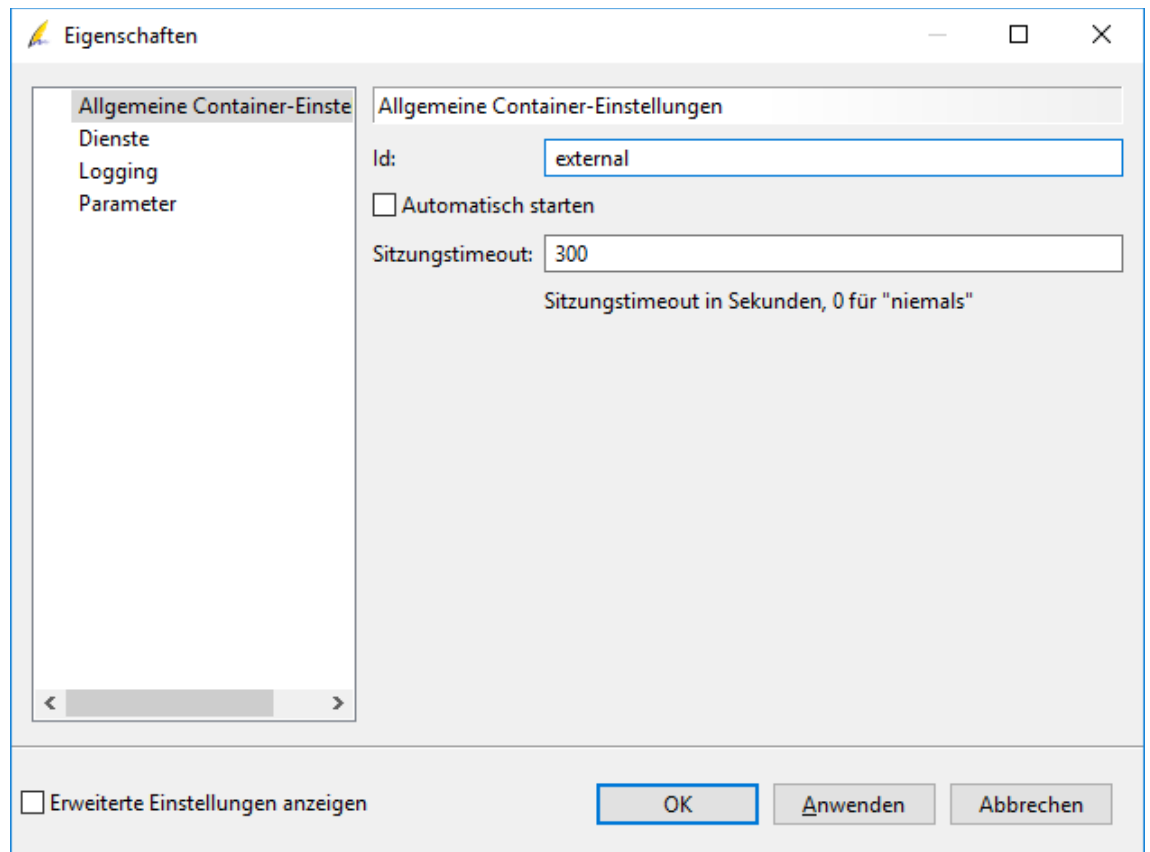
7.2 Adding a local container

For C-style calls you can use the "Local container" - it is protocol agnostic and is used in a variety of combinations.

You start like in the previous example by adding a container.



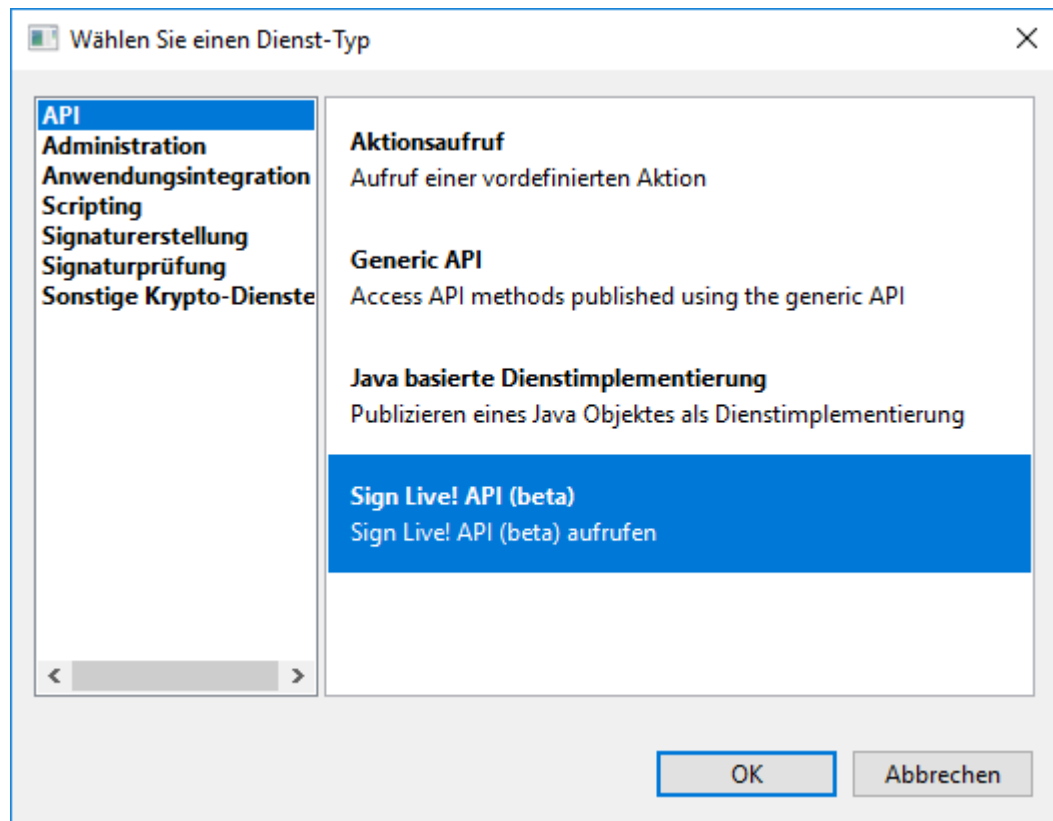
Select "Local service container".



There are not many settings for this container, but at least give it a meaningful name like "external".

7.3 Adding a service

Again, we add a service - this time we give the predefined "SignLive API" a try.



This API summarizes some of the most common tasks delegated to the application like

- sign a single document
- sign a batch of documents
- validate a document

For details, you should consult the respective documentation.

For sure you can publish any service you like.

7.4 Start the container

As always, you must start the container to make it publish its services.

7.5 The C-Style client

7.5.1 Overview

The trickier part is on the client side this time.

The C-style call in to the service container is implemented in "stageci.dll". In fact, all our DLLs use the same code base and method signatures, they only differ in the way the calls are marshalled and forwarded to some application implementation.

Here a summary of the IDE prerequisites:

- Our header files can be found in the "sdk" folders "IJLauncher" or "SharedLib". As you must use dynamic loading via "LoadLibrary" you should not need them anyway
- Your C-Code must reference our "stageci.dll" (which in turn loads ijlauncher.dll)

7.5.2 Launching

Your C-code must first launch the application before you can perform a "call in".

This is done in the "launch" function

```
result = launch("-launch interactive noblock resident -nowakeonstart");
```

You can add individual launch parameters (see "Operators Guide").

7.5.3 Calling

The C-style API is based on the well known "callArgs(name, args)" style. Only this time we have to deal with indexed arguments and the arguments are expressed using the C-type **ij_value_t**.

7.5.4 Method name

The method name is an URI with the following components

- //
- Container ID
- /
- Service path
- /
- Method name

Example

```
//external/method
```

7.5.5 Arguments

To make the service container call in as straightforward as possible, the "stageci.dll" implementation assumes a single string argument which contains the marshalled name/value pairs that you have already encountered in the other examples.

```

ij_value_t args[1];
ij_value_t result;
//...
args[0].type = TYPE_STRING;
args[0].value.string_value = "" ;
result = synchCallArgs("//external/connect", 1, args);
args[0].type = TYPE_STRING;
args[0].value.string_value = "document.path=c:\temp\foo.pdf;foo=bar;" ;
result = synchCallArgs("//external/signDocument", 1, args);
args[0].type = TYPE_STRING;
args[0].value.string_value = "" ;
result = synchCallArgs("//external/disconnect", 1, args);

```

You just concatenate "key=value" pairs with a ";" separator. The separator may be followed by whitespace (linebreaks). If value contains the separator, it must be quoted.

7.5.6 Results

The result values are serialized in the same format, as key/value pairs separated by semicolons.

Example

```

signatures.0.path=c:/data/out/templ.pdf;
signatures.1.path=c:/data/out/templ.pdf;

```

If the result is not a composite structure, but a primitive value, the result is marshalled into a result structure with the property "_wrapped".

Example

```

_wrapped=12

```

7.5.7 Exceptions

If the call throws an exception, the result is marshalled into a result structure with the property "_exception".

Example

```

_exception._class=de.intarsys.stage.signlive.api.SignLiveAPIException
_exception.message=that did not work
_exception.cause._class=java.lang.RuntimeException
_exception.cause.message=because the argument is rotten

```

7.6 Using the demo code

Even so the demo code is linked against the "stagesl.dll" (which has different argument conventions), you can easily reuse it to call your services.

The marshalling/unmarshalling protocol is configured in the stagesl.dll.lcnf (respective the stageci.dll.lcnf). By simply copying the content from one to the other, you can reuse the demo client.

1. Copy the contents from stageci.dll.lcnf to stagesl.dll.lcnf
2. start "stagesldemo.exe"

3. launch the application
4. call a method of your service

The result should be displayed in a simple dialog.

7.7 Using individual launch arguments

7.7.1 nowakeonstart

The commandline option "-nowakeonstart" gives you the ability to launch the application without the main window.

For more information see "Operators Guide".

7.7.2 profile

The commandline option "-profile <dir>" gives you the ability to define the profile directory.

For more information see "Operators Guide".

7.7.3 itdirs

The commandline option "-itdirs" gives you the ability to define instrument directories in addition to the standard application instruments.

This is useful, if you have individual extensions that you keep separate from the application or that are only deployed together with the C-code.

For more information see "Operators Guide".

7.8 Signature example

The installation includes an example "sdk/SharedLib/demo/ServiceContainer". Here you find

- An instrument definition that installs a local service container
- A TestApp that loads our DLL and executes functions in the service container.

7.8.1 Service container instrument

The instrument file defines a service container "external" with a service entry point at "functor".

Calling a method in this container has the form

```
//external/functor/<method>
```

You can deploy the instrument either in a standard Sign Live directory – this means you do not explicitly reference the instrument when launching, but you have to tweak the installation. Alternatively you can

deploy the instrument along with your own code and reference it when launching the application.

7.8.2 Signature method

The signature method

```
com.cabaret.security.document.signing.MultiSignerFactory
```

provides a multi purpose entry point for signing.

7.8.3 Launch application

```
result = launch("-launch interactive noblock resident -nowakeonstart -itdirs  
c:\temp\external\instruments");
```

Here you see an explicit example for launching where the instrument is provided explicitly via parameter.

7.8.4 Single document

Argument example

```
document.path=c:/temp/data/tags.pdf;  
documentSigner.factory=com.cabaret.security.method.pdf.signing.PDFDocumentSignerFactory;  
documentSigner.args.digestSigner.factory=de.intarsys.stage.security.device.smartcard.sig  
nature.SmartcardDigestSigner;  
documentSigner.args.digestSigner.args.signerSelectionInteractive=true;  
documentSigner.args.digestSigner.args.signerIdentifier=usage=qualified_signature;
```

These parameters define

- A document to be signed
- The PDF signature method
- A smartcard signature device
- A flag to support manual selection when the key material is not unique
- A filter to select only QES keys

Call example

```
args[0].type = TYPE_STRING;  
args[0].value.string_value =  
"document.path=...; ";  
result =  
synchCallArgs("//external/functor/com.cabaret.security.document.signing.MultiSignerFacto  
ry", 1, args);
```

As a result, you will receive

```
signature.path=c:/temp/data/tags.pdf;
```

7.8.5 Multi document

```
documents.0.path=l11.pdf;  
documents.1.path=ffff.pdf;  
documentSigner.factory=com.cabaret.security.method.pdf.signing.PDFDocumentSignerFactory;  
documentSigner.args.digestSigner.factory=de.intarsys.stage.security.device.smartcard.sig  
nature.SmartcardDigestSigner;  
documentSigner.args.digestSigner.args.signerSelectionInteractive=true;  
documentSigner.args.digestSigner.args.signerIdentifier=usage=qualified_signature;
```

Result

```
signatures.0.path=l11.pdf;  
signatures.1.path=ffff.pdf;
```

7.8.6 Document integrity

To ensure document integrity, you should add a hash value. This way Sign Live can check if the document that was collected from the referenced path matches with the one you have provided at the API.

The best way to add a document hash is to create an ASN.1 DER encoded hash of the complete document content using some of your preferred platform libraries and add it as an argument (base64 encoded).

Example

```
document.path=c:/temp/external/tags.pdf;document.der=e6e6e6...e6e6e;
```

8. Advanced signature scenarios

8.1 Overview

While in the first chapters we switched some technical and cryptographic details, this section is dedicated to investigating typical scenarios on the document level, e.g.

- visible signatures
- timestamps

8.2 Visible signatures

When signing PDF documents, one of the shining features is the possibility to add visual appearance to the signature.

The simplest way to use this feature is to add the arguments to your client.

```
nvps.add(new BasicNameValuePair("document.content", encodedDoc));
nvps.add(new BasicNameValuePair("document.name", inName));
/*
 * The PDF field we want to create
 */
nvps.add(new BasicNameValuePair("field.create", "true"));
nvps.add(new BasicNameValuePair("field.position", "50*50"));
nvps.add(new BasicNameValuePair("field.size", "200*100"));
nvps.add(new BasicNameValuePair("field.pageRange", "first"));
/*
 * The content for the field
 */
nvps.add(new BasicNameValuePair("decorator.factory",
    "de.intarsys.security.document.type.pdf.signature.ExtendedDecoratorFactory"));
nvps.add(new BasicNameValuePair("decorator.args.text", "A signature..."));
nvps.add(new BasicNameValuePair("decorator.args.textScaleWhen", "always"));
nvps.add(new BasicNameValuePair("decorator.args.textScaleProportional", "true"));
nvps.add(new BasicNameValuePair("decorator.args.textVAlign", "bottom"));
nvps.add(new BasicNameValuePair("decorator.args.textHAlign", "center"));
nvps.add(new BasicNameValuePair("decorator.args.icon.name", "signature.png"));
nvps.add(new BasicNameValuePair("decorator.args.icon.content", "<base64 encoded>"));
nvps.add(new BasicNameValuePair("decorator.args.iconScaleWhen", "always"));
nvps.add(new BasicNameValuePair("decorator.args.iconScaleProportional", "true"));
nvps.add(new BasicNameValuePair("decorator.args.iconVAlign", "center"));
nvps.add(new BasicNameValuePair("decorator.args.iconHAlign", "center"));
nvps.add(new BasicNameValuePair("decorator.args.layout", "overlay"));
```

There are two information chunks we need. First, the "physical field" in the PDF document we want to create and second the algorithm that creates the visible field content.

You can find detailed information for the arguments in the "Security Applications Developers Guide".

8.3 Visible signatures with dynamic field position and size

One of the trickier requirements is adding per document arguments or even positioning a signature field at a dynamic position, dependent on the flow of a dynamically created document.

In the application this solved in a very flexible way (that can even be used in lots of other scenarios, so keep on reading).

By the way, this scenario is already implemented in the "Cosima" workflow. You only need to follow this guide if you need the feature elsewhere.

8.3.1 Embedded tags

The first step is adding "tags" to a document instance. A "tag" is a simple key/value pair that is attached to a document and can be accessed via an API.

Attaching a tag to a PDF document is especially easy. You simply add text in the document, enclosed in the special chars "@@".

```
@@foo=bar@@
```

This would add the tag "foo" with value "bar". Well, in itself this may not serve any purpose., so let's go on.

Now we define a special convention, copying all tags that start with "args." to an argument set. There's a component to accomplish this, "de.intarsys.tools.tag.TagsToArgsFunctor".

A script snippet that will merge the tags in your arguments will look like this

```
var idoc = Processor.callArgs(
    'DocumentLoaderFactory', {
        locator: document
    });

var viewer = Processor.callArgs(
    'DocumentViewerFactory', {
        document: idoc
    });

/*
 * detect key=value style tags in document
 */
Processor.callArgs("com.cabaret.pdf.processor.tagdetector.PDFContentTagDetectorFactory",
{
    document: idoc,
    syntax: "separated"
});

/*
 * the static literal java object for the document signer
 */
var documentSignerArgsObj = {
    digestSigner: {
        factory: "com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",
        args: {
            signerIdentifier: 'SerialNumber:8139571262270123122;',
            signerPassword: 'password'
        }
    },
    timestampServiceName: 'MyTimestampServer'
};

/*
 * the mixed up version, copying "args." tags from idoc
 */
var documentSignerArgs = Functor.callArgs("de.intarsys.tools.tag.TagsToArgsFunctor", {
    args: documentSignerArgsObj,
    target: idoc
});

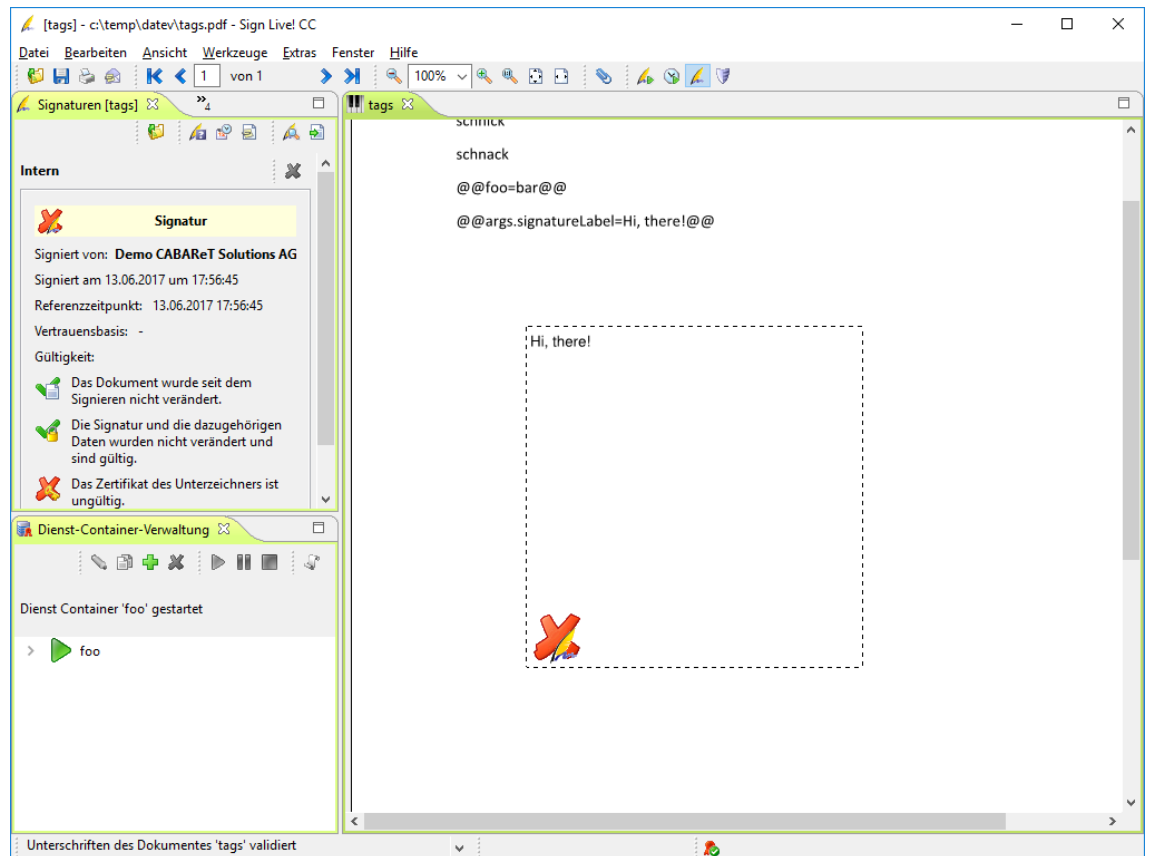
/*
 * call the wizard or processor here...
 */
var signature = Wizard.callArgs(
    'com.cabaret.security.document.signing.ui.DocumentSignerWizardFactory', {
        document: idoc,
        documentSigner: {
            factory: "com.cabaret.security.method.pdf.signing.PDFDocumentSignerFactory",
            args: documentSignerArgs
        }
    });

idoc.release();
```


If we now put any tag in the document that starts with "args.", the trailer of the tag will be merged into the "documentSignerArgs", for example, putting

```
@@args.signatureLabel=Hi, there!@@
```

in the document will result in a signature like this:



8.3.2 Embedded dynamic tags

The job is a little bit more sophisticated when it gets dynamic. If you have a plain letter, you may not know where to put a signature field in advance (remember a field has to be defined by a page and coordinates in the API).

This can be solved by reverting to "meta tags", that is tags holding information about tags. If you use PDF embedded tags, 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

Advanced signature scenarios

Visible signatures with dynamic field position and size

```
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. If you add a tag like

```
@@signhere=-----@@
```

In reality not only a tag "signhere=-----" is created, but something like

```
meta.signhere.llx=95  
meta.signhere.lly=304  
meta.signhere.urx=205  
meta.signhere.ury=324  
meta.signhere.width=110  
meta.signhere.height=20  
meta.signhere.page=0
```

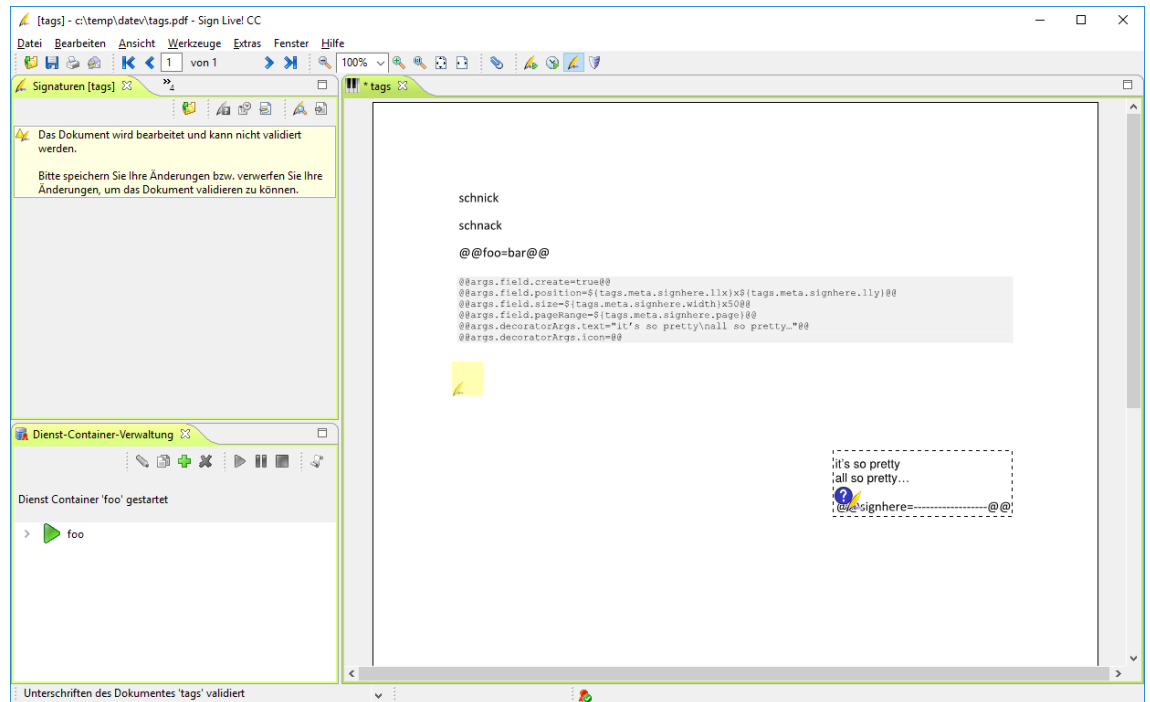
You can use this information now in your argument tags with string expansion like this

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

after this little addition to your script that takes care that the respective arguments are expanded before starting the wizard:

```
// creating the args is here before  
// ..  
  
/*  
 * make "tags" present for field expansion  
 */  
var resolver = new Packages.de.intarsys.tools.expression.TagResolver(idoc);  
resolver = Packages.de.intarsys.tools.expression.MapResolver.create("tags", resolver);  
Packages.de.intarsys.tools.expression.ThreadContextAwareResolver.attach(resolver);  
  
/*  
 * expand on "field" argument  
 */  
Functor.callArgs("de.intarsys.tools.functor.ArgsCruncher", {  
    args: documentSignerArgs,  
    expand: {  
        field: true  
    }  
});  
  
// ..  
// calling the wizard hereafter
```

And here the result



8.4 Timestamps

8.4.1 Overview

Adding a timestamp adds a proof by some third-party instance, that a certain event (most often a signature) has taken place at a certain moment in time.

8.4.2 Select a timestamp service

You have different options to create a timestamp - the simplest one is addressing a well-known, RFC3161 compliant server.

While for certain use cases you may need another quality of service (such as a qualified timestamp) for our purpose we simply use one of the numerous free timestamp services around. If this one does not fit/work, you can simply find another via internet search.

We are going to use

<https://freetsa.org/tsr>

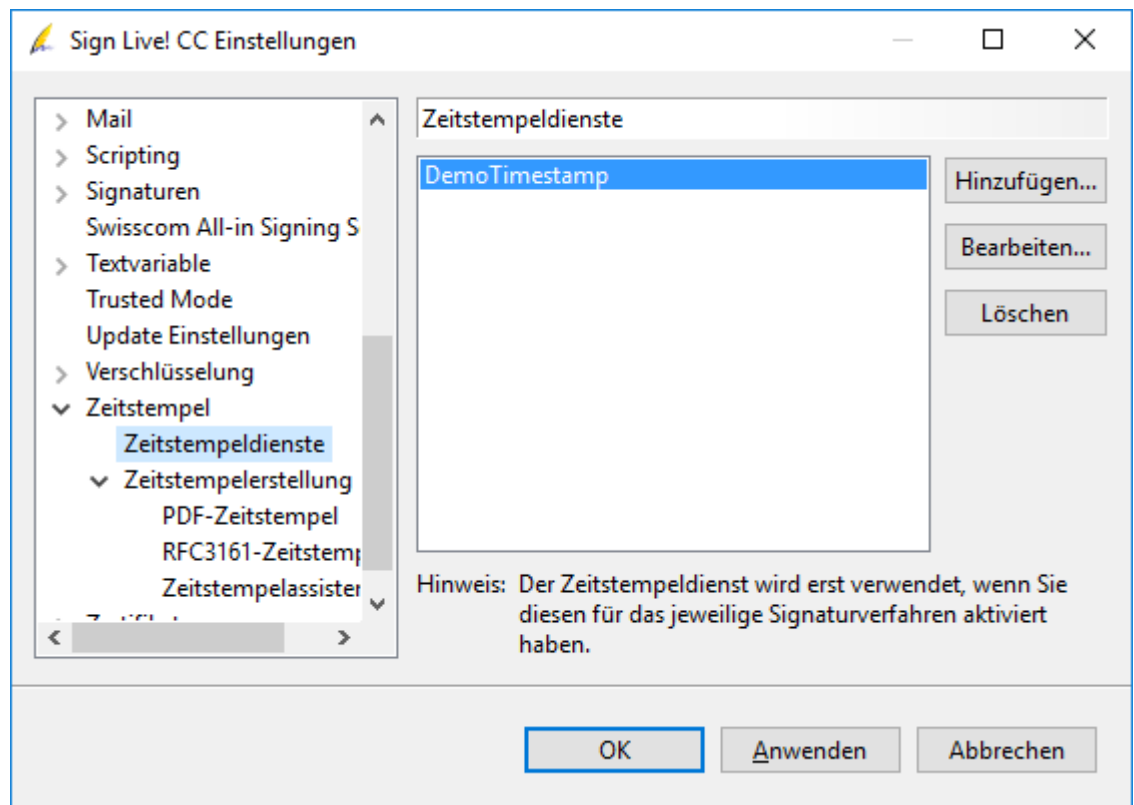
for this example.

When selecting a timestamp service, keep in mind that only HTTP Basic Authentication is supported at the moment.

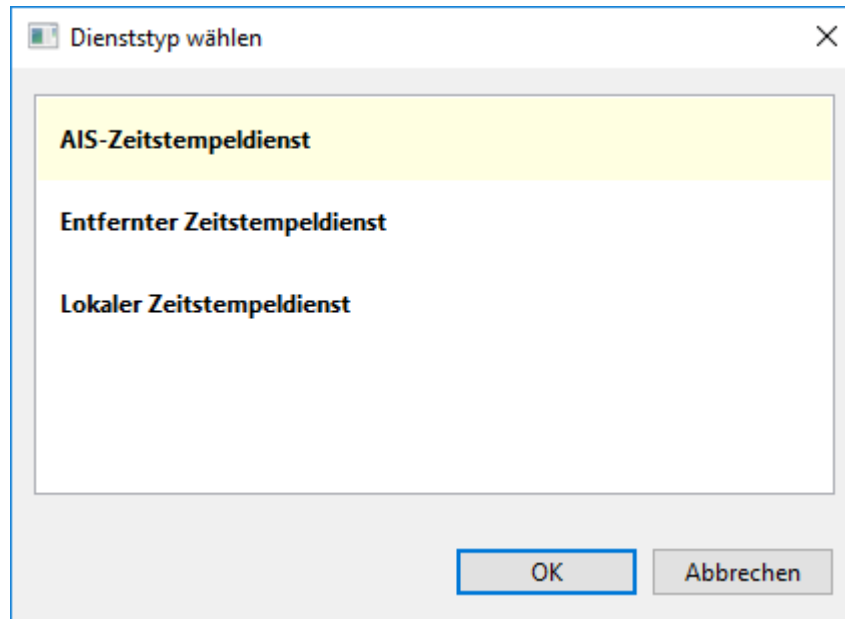
8.4.3 Use a pre-registered timestamp service

8.4.3.1 Register a timestamp service

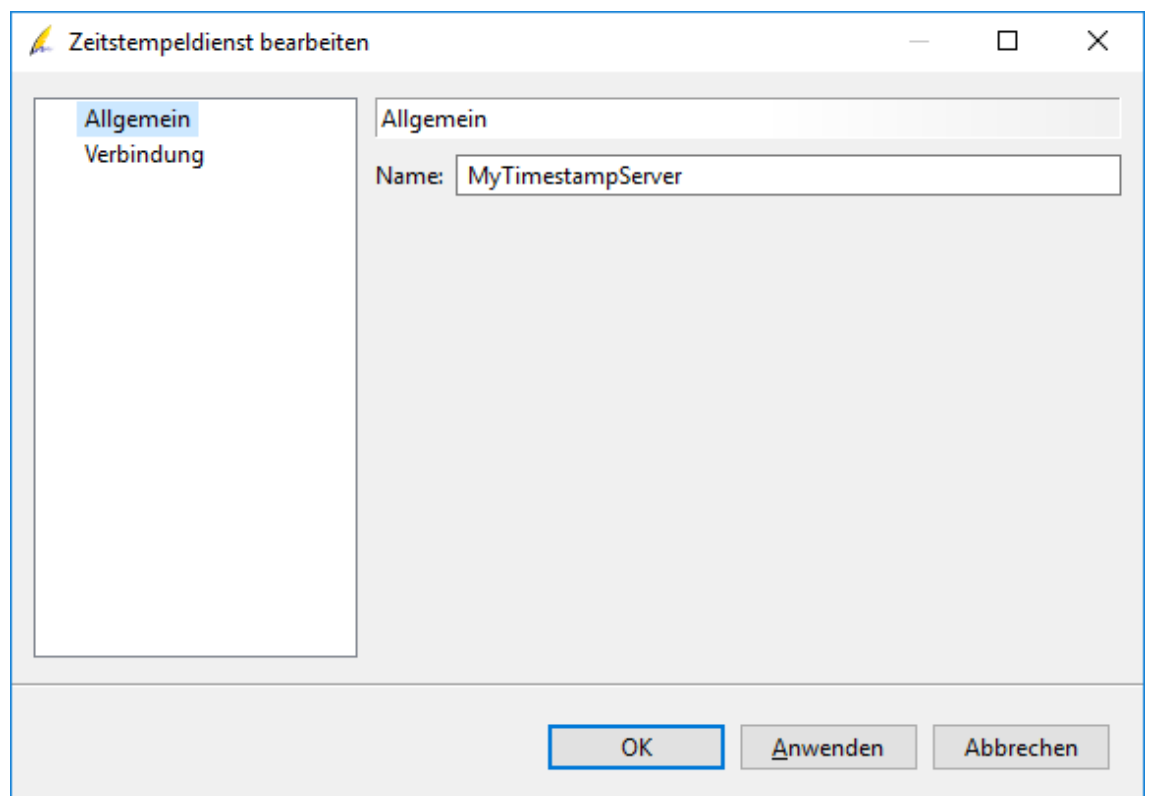
You can register a timestamp service in the settings dialog of the application, on the tab "Timestamp services".



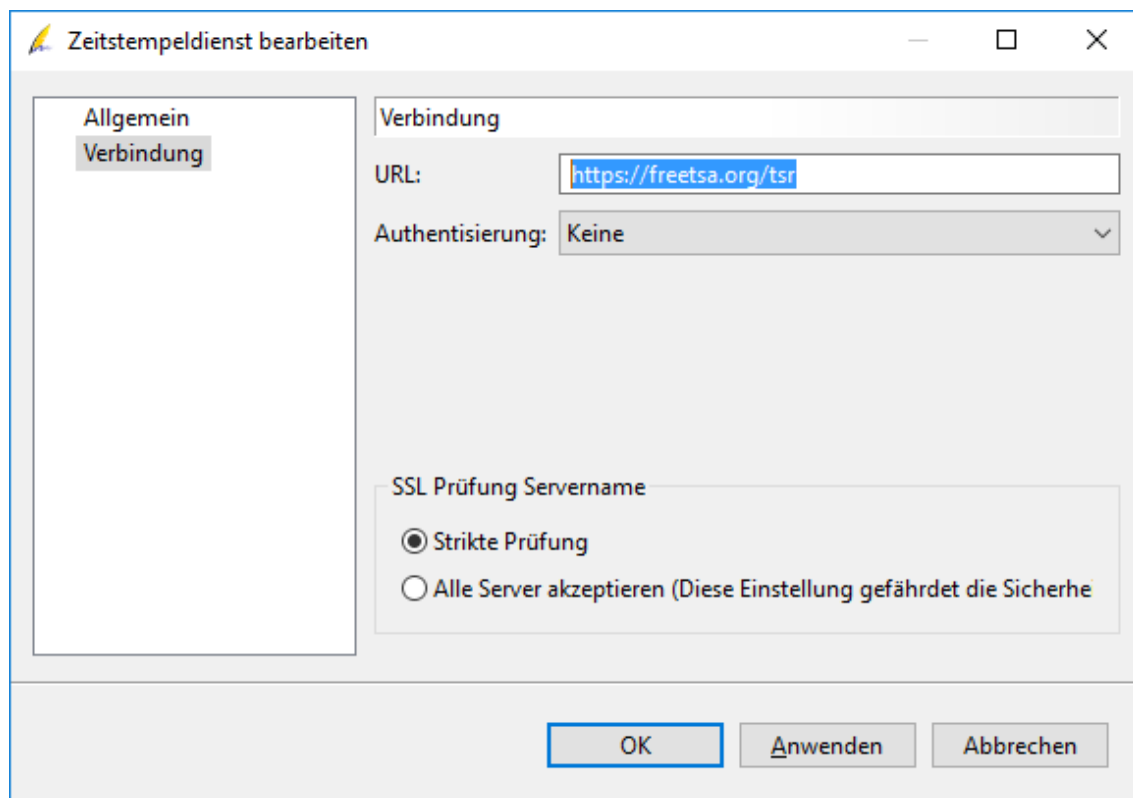
Pressing "Add..." will provide you with some choices, depending on your current installation. You should select "Remote service" here to connect to a standard remote timestamp service.



You must give a unique name to the service. This name is used later on to access the service when creating a signature.



Then you need to associate a server, in our case simply <https://freetsa.org/tsr> a free, non-authenticated, TSL based and standard compliant server. Feel free to use the server of your choice.



Now you have timestamp service you can use when creating a signature.

8.4.3.2 Reference a registered timestamp service when signing

You can add a timestamp to your signature by referencing the newly created service.

Example JavaScript Wizard call

```
var signature = Wizard.callArgs(  
    'com.cabaret.security.document.signing.ui.DocumentSignerWizardFactory', {  
        document: idoc,  
        documentSigner: {  
            factory: "com.cabaret.security.method.pdf.signing.PDFDocumentSignerFactory",  
            args: {  
                digestSigner: {  
                    factory:  
"com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",  
                    args: {  
                        signerIdentifier: 'SerialNumber:8139571262270123122;',  
                        signerPassword: 'password'  
                    }  
                },  
                timestampServiceName: 'MyTimestampServer'  
            }  
        }  
    }  
});
```

8.4.4 Use an on-the-fly timestamp service

8.4.4.1 Reference an on-the-fly timestamp service when signing

It is possible to describe the timestamp server directly without prior registration for HTTP based timestamps.

```
var signature = Wizard.callArgs(  
    'com.cabaret.security.document.signing.ui.DocumentSignerWizardFactory', {  
        document: idoc,  
        documentSigner: {  
            factory: "com.cabaret.security.method.pdf.signing.PDFDocumentSignerFactory",  
            args: {  
                digestSigner: {  
                    factory:  
"com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory",  
                    args: {  
                        signerIdentifier: 'SerialNumber:8139571262270123122;',  
                        signerPassword: 'password'  
                    }  
                },  
                timestampDevice: {  
                    factory:  
"de.intarsys.security.device.httptimestamp.device.HttpTimestampDeviceProvider",  
                    args: {  
                        url: "https://freetsa.org/tsr"  
                    }  
                }  
            }  
        }  
    }  
});
```