

# Sign Live! CC Operators Guide

Februar 2022



intarsys GmbH

## Sign Live! CC Operators Guide

Version 7.1

Installing and configuring Sign Live! CC

intarsys GmbH  
Sign Live! CC Operators Guide  
Version 7.1

All rights reserved  
© 2021 intarsys GmbH  
[www.intarsys.de](http://www.intarsys.de)

# Preface

---

- 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

- Who should read this book

This book is intended for operators and system integrators intending to enhance, customize, enhance or integrate the product.

If you simply want to add some buttons to the GUI, you may be better off reading the “Scripting Tutorial” and mimicking some of the examples. You don’t need the information included here. ...but, still, you may get inspired. There are more possibilities than you might imagine at the moment.

The basic concepts and APIs available to the developer are presented, as well as complete examples that you can use as templates.

You will not need this book if you simply wish to use the product or work with simple scripting features. You will find information thereto in the tutorials and the online help.

- Other documentation

The Online Help includes information about every feature that can be done with the desktop application (and some more). It is installed and available with every distribution.

“Operator’s Guide” is the book about installation and configuration of the product.

Now we come to the fun part, do a little programming – “Scripting Tutorial” shows a fast path to customizing the product and some useful tips for scripting it.

“Developer’s Guide” is the book about programming the product. This is about the architecture, the basic concepts and generic APIs.

On some special topics there is additional documentation, presenting APIs and examples for specific business tasks.

One of the most important is “Security Applications Developer’s Guide”, the book about external access to the security applications in the product.

### ■ 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](mailto:support@intarsys.de)

Website

[www.intarsys.de](http://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 arising from the information contained in this book, or from the use of the disks or programs that may accompany it.

# Contents

---

Preface	5
▪ Trademarks	5
▪ Who should read this book	5
▪ Other documentation	5
▪ Reviews and comments	6
▪ Disclaimer	6
Contents	7
Introduction	11
1. Installation	13
1.1 Overview	13
1.2 Microsoft Windows Platform	13
1.3 Unix Platform	13
2. Configuration	17
2.1 Starting the application	17
2.1.1 Overview	17
2.1.2 Start with the Launcher	17
2.1.3 VM Options File	17
2.2 The Configuration File	18
2.2.1 Description	18
2.2.2 Syntax	19
2.3 The JNI Callin Library Configuration Files	21
2.3.1 Description	21
2.3.2 Syntax	21
2.4 Preferences	30
2.4.1 Overview	30
2.4.2 Export preferences	31
2.4.3 Import preferences	31
2.4.4 Tip	31
2.5 Licenses	31
2.5.1 Overview	31
2.5.2 Importing licenses	31

# Contents

2.6	Extension Points	32
2.6.1	Overview	32
2.6.2	Logging	32
2.7	Monitor Configuration	34
2.7.1	Monitor	35
3.	Variables	39
3.1	Introduction	39
3.2	Concepts	39
3.3	Configuration	39
3.4	Editing	40
3.4.1	Edit the extension point	40
3.4.2	Declare preferences	41
3.4.3	Use the variable GUI editor	41
3.4.4	Import a preferences file	42
3.5	String expansion	42
3.6	Script access	43
4.	Automation	45
4.1	Commandline API	45
4.2	ActiveX	45
4.3	HTTP	45
4.4	Web Services	45
4.5	File monitoring	46
4.6	Scheduler	46
4.7	Printing	46
4.8	Message queues	46
4.9	Batch	46
5.	Commandline Interface	47
5.1	Commandline Interface (CLI)	47
5.1.1	Overview	47
5.1.2	Return Values	47
5.1.3	Error Handling	47
5.1.4	Scripting Integration	47
5.1.5	Option chaining	48
5.1.6	CLI Context	48
5.1.7	Hiding option values	48
5.2	CLI Platform Options	49
5.2.1	Profile directory	49
5.2.2	Search Path for Instruments	49
5.2.3	Strict instrument loading	50
5.2.4	Log configuration	50
5.2.5	Configuration file	51
5.2.6	Launch Mode	51
5.2.7	Disable application launch	52
5.2.8	Window creation & activation	53
5.2.9	Command Line File	54
5.3	CLI Application Options	55



5.3.1	Overview	55
5.3.2	Options	56
5.3.3	Working Stack	56
5.3.4	Standard Application Options	57
5.3.5	PDF Options	63
5.3.6	Exchange Options	64
5.3.7	Scripting Options	66
6.	Service framework	71
6.1	Overview	71
6.2	Basic concepts	71
6.3	Quick walkthrough	71
6.4	Service container	78
6.4.1	Common properties	78
6.4.2	File system container	79
6.4.3	Jetty Container	86
6.4.4	Virtual printer container	90
6.4.5	Scheduler container	91
6.4.6	Message queue (JMS) container	92
6.5	Services	93
6.5.1	Overview	93
6.5.2	Service Call	93
6.5.3	Standard Properties	94
6.5.4	JavaScript	96
6.5.5	External script	98
6.5.6	Java based script	100
6.6	Decorators	101
6.6.1	Overview	101
6.6.2	Logging	102
6.6.3	Audit Log	102
6.7	Advanced service definition	107
6.7.1	Scripted service factory	107
6.7.2	Java based service factory	110
6.7.3	Accessing context	110
6.8	Session handling	111
6.8.1	File system container	111
6.8.2	Jetty	111
6.8.3	Virtual printer	112
6.8.4	Scheduler	112
6.8.5	Message queue	112
6.8.6	Explicit session handling	112
6.9	String expansion	114
7.	Batch	119
7.1	Overview	119
7.2	Batch Creation	119
7.3	Define a Batch	119
7.3.1	The Toolbar	120

## Contents

7.3.2	The Lifecycle	120
7.3.3	Batch Settings	120
7.3.4	Batch Source Templates	120
7.3.5	Batch Destination Templates	121
7.3.6	Batch Actions	122
7.4	String Expansion	123
7.4.1	Overview	123
7.5	Advanced Features	123
7.5.1	Attachments	124
8.	Appendix	125
8.1	String Expansion	125
8.1.1	Basics	125
8.1.2	Namespaces	128
8.1.3	Special Namespaces	134
8.1.4	Formatting	136
8.1.5	Special Topics	144
8.2	Installation of PDF Printer	145
8.2.1	PDF Printer	145
8.2.2	Installation	145

# Introduction

---

This document is about installation, customization and integration of Sign Live! CC. If you're looking for information on how to work with the application you will find all the answers in the online help. This is not the right book for you.

Some of the basic features explored here are

- You can **integrate** the product using one of the many APIs provided
  - Commandline
  - Shared library
  - ActiveX
  - Native Java
  - Services
    - File System monitoring
    - XML RPC
    - Native HTTP
    - SOAP
- You can **extend** its features using **Instruments**. These extensions range from simply adding an additional icon in the toolbar, giving you a shortcut for a daily task to automatic backend integration, a database, interfacing archiving systems and so on.

The concept of an application built on a flexible platform is today widely known as a “rich client platform”. It is comparable to the concepts and services provided, for example, by Eclipse RCP. This product is based on our implementation of this concept, named **claptz** (which is not necessarily a meaningful acronym).

## Introduction

Here is a list of our primary design goals:

- **Lightweight**  
**claptz** should be lightweight in all dimensions. Small jar, few dependencies, few resources. All the bells and whistles can be attached, but are not included.
- **Simple**  
Insofar as a framework of this type can be called “simple”, this one should fulfill these criteria. The learning curve is not as steep as, for example, in eclipse.
- **Ad Hoc Customization**  
The main goal here is to create applications that can be customized upon installation by the user. The customization has to be accomplished with tools typically available at this environment and with easily acquired knowledge. You can watch this feature in action by reading the “Scripting Tutorial”. Almost everything you can do with a Java IDE you can also do with a text editor and a basic understanding of JavaScript. This point cannot be stressed enough - the integration of scripting languages into the framework at a very early point is a key concern and other goals may be subordinate to this one.
- **Performant**  
The target applications are desktop tools. These tools have to be up and running fast, typically running concurrently with many others. The platform should not eat up CPU and memory resources.

# 1. Installation

---

## 1.1 Overview

The standard installation of Sign Live! CC includes both the client application - accessible in the start menu - and the server application components. This means after successfully completing the installation you can use both the CLI and the API within your processes.

Most of these functions are controlled with licenses. Please read the EULA for more information. For commercial use an appropriate license is required.

## 1.2 Microsoft Windows Platform

Sign Live! CC is packaged with a setup program, which will guide you through the installation process.

## 1.3 Unix Platform

For the Unix systems Sign Live! CC is delivered as a tarball (tar.gz). Just unpack the contents in a folder of your choice.

You will need a Java 7 Runtime Environment from Oracle in order to run the product. For some Unix or Linux distributions this comes preinstalled, on others you will need to install it using the delivered installation files or the online repositories for your specific version.

If you have different Java distribution you must download and install the Java Runtime Environment from Oracle. It's very probable that you will need to adjust links and environment variables after the installation of the JRE. Please refer to the installation guides from Oracle.

You can start Sign Live! CC with the binary file *signlivecc*. For this binary to work it needs the Java Runtime Environment location. There are several ways to provide this information, listed in order of preference:

- A subdirectory called **jre** within the Sign Live! CC folder.  
If there is a Java Runtime Environment in this subdirectory, it will be used. You can copy your existing Java Runtime Environment, put a link to it here or install it directly.

To link to your existing Java Runtime Environment (example):

```
ln -s /usr/lib/jvm/java-1.7.0-sun /usr/local/<installationpath>/jre
```

- Environment variable **CABARET\_JAVA\_HOME**.  
If there is a Java Runtime Environment in the directory denoted with this environment variable it will be used.

To use this (example):

```
export CABARET_JAVA_HOME=/usr/lib/jvm/java-1.7.0-sun
```

- Environment variable **JAVA\_HOME**.  
If there is a Java Runtime Environment in the directory denoted with this environment variable it will be used. This is the most common way to determine a Java Runtime Environment and it is recommended.

To use this (example):

```
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-sun
```

- A file **/etc/jvm**.  
Each line of this file denotes a potential Java Runtime Environment folder. If there is a Java Runtime Environment in the folders denoted within this file it will be used. This file exists on Debian based systems, on all others you will need to create it. This is the recommended way on Debian based systems and on all other systems where this file already exists.

Example content:

```
/usr/lib/jvm/jre-1.7.0-sun
```

The paths in the examples need to be replaced with your actual path names. If all four options fail in determining a Java Runtime Environment, the binary cannot be started.

You can then try to use the shell script *signlivecc.sh*. This script uses the java binary found in the system paths.

Finally, you can create your own starter file for Sign Live! CC . For example, if you have unpacked to */usr/local*:

```
[Desktop Entry]
Version=1.0
Encoding=UTF-8
```





## 2. Configuration

---

### 2.1 Starting the application

#### 2.1.1 Overview

Sign Live! CC is a Java application, therefore you can configure the Java VM as usual. For example, you can modify system properties of the VM or change the dedicated memory assignments.

For more details and specifications for the commandline parameters for Java VM, please refer to <http://java.sun.com>

#### 2.1.2 Start with the Launcher

Sign Live! CC is delivered with a native launcher executable. You find this executable in the “bin” folder of your installation under the name

- `SignLiveCC.exe`

This launcher starts and configures the Java VM. The configuration uses preset defaults.

If you want to start with a different configuration you can achieve this by starting the Java VM directly without the launcher by calling *java.exe* with the appropriate options. This call is somewhat complicated though, so it is not recommended.

#### 2.1.3 VM Options File

##### 2.1.3.1 Overview

Instead of starting *java.exe* we recommend customization using the *vmoptions* file. You can edit the Java VM options in a file called *SignLiveCC.exe.vmoptions*. Each option is in a separate line. They are passed to the VM upon starting the launcher. This *vmoptions* file must be in the same folder as the launcher executable.

Example content of a vmoptions file that will increase the memory available to the VM.

```
-Xms64m  
-Xmx256m
```

### 2.1.3.2 Language Options

The language used for the application is determined by the local settings of your operating system.

If you want to use a different language than the one set by your OS you can use the Java VM option *user.language* in your vmoptions file. German, for example, is defined by *de*, english by *en*. You must prefix the setting with *-D*.

Example:

```
-Duser.language=en
```

The application now uses English.

### 2.1.3.3 Memory Options

The Java VM uses 64 MB by default. Under certain circumstances this is not enough for the application to run smoothly, for example, when processing large documents or documents with memory hogs, like large images. You will then get the Java VM error message *Out of Memory*.

You can, however, assign more memory to the Java VM in your vmoptions file using the options below.

Example:

```
-Xms64m  
-Xmx512m
```

The application will now use at least 64 MB memory and up to 512 MB.

## 2.2 The Configuration File

### 2.2.1 Description

The configuration file allows you to customize the behavior and runtime environment of Sign Live! CC .

By default Sign Live! CC loads a configuration file located under *config/stage.config*. If no such file is found, the application will start with the default options.

You can also specify the configuration file using the command line option *-config*.

## 2.2.2 Syntax

**STAGE**

Element	Content	
stage	Configuration element for the application	
.	Attribute	Content
	name	Application name
	basedir	The base directory of the application. All related paths relate to here.  The installation directory is defaulted.
	profiledir	The base directory for user specific files. For example, preferences and log files are stored here.  The default is the subdirectory with the suffix <code>.SignLiveCC_&lt;version&gt;</code> in the OS specific user directory.  For example, in Windows <code>C:\Documents and Settings\&lt;user&gt;\.SignLiveCC_&lt;version&gt;</code> .
.	Element	Content
	instruments	Contains the instruments to be loaded.

**INSTRUMENTS**

The application loads a number of instruments upon startup. This can be customized in the configuration file. If no *instruments* entry is present, the application will load all instruments in the folder “*instruments*” in your installation.

Element	Content
---------	---------

---

**instruments**    The instruments to be loaded. If instruments are specified here, no instruments from the folder *instruments* in your installation will be loaded.

---

	Attribute	Content
	Element	Content

---

load            Load directive for one or more instruments.

---



---

## LOAD

The specified instrument(s) will be loaded.

Element	Content
---------	---------

---

load            Load directive for one or more instruments.

---

	Attribute	Content
--	-----------	---------

---

path            The path to the instrument definition or the path to a folder with several instrument definitions, which are to be loaded.

---

	Element	Content
--	---------	---------

---

Here is an example of a configuration file loading an XML entity with the instrument load declarations for a more modular approach to configuration files.

```
<?xml version="1.0" standalone="yes" ?>

<stage>
  <instruments>
    <load path="instruments"/>
    <load path="instruments.sample"/>
    <load path="/mytestapplication/instruments"/>
  </instruments>
</stage>
```

## 2.3 The JNI Callin Library Configuration Files

### 2.3.1 Description

The Windows Version of the application comes with an ActiveX component that allows you to embed Sign Live! CC into other Windows applications. The ActiveX component works via a JNI callin library. The library file for ActiveX is called `signliveax.dll`, the name of the JNI callin library is `ijlauncher.dll`. The corresponding configuration files have the same name as the library with `.lcnf` appended to it. If you want to use a different configuration for a specific container you can do so by putting the container's executable name between the library name and the suffix (example: `signliveax.dll.iexplore.exe.lcnf`). The contents of the configuration file are in XML format.

The library will internally set a "base directory" to the directory containing the library file. If the library file is in a directory called `bin` the "base directory" will be set to the parent directory.

Relative filenames in the configuration will be treated as relative to the base directory.

### 2.3.2 Syntax

#### LAUNCHCONFIG

Element	Content		
launchconfig	Required root element		
		Attribute	Content
		Element	Content

?	java	Configuration element for launching the Java Virtual Machine.
---	------	---

?	context	Configuration element for initializing an application context.
---	---------	--

## JAVA

Element	Content
---------	---------

java Configuration element for launching the Java Virtual Machine. This configuration element takes effect only in the *ijlauncher.dll.lcnf*.

.	Attribute	Content
---	-----------	---------

.	Element	Content
---	---------	---------

1 searchSequence Configuration element for the methods to search for a JVM library.

?	init	Configuration element for settings pertaining to VM initialization.
---	------	---

? nativeLibraryDirectories Configuration element for the composition of the Java native library directories

## SEARCHSEQUENCE

Element	Content
---------	---------

searchSequence Configuration element for the methods to search for a JVM library. Child entries are processed in sequence. The first JVM library found will be used to launch the JVM.

.	Attribute	Content
---	-----------	---------

	Element	Content
	directory	Configuration element to search the specified directory for a <i>jvm.dll</i> file.
	registry	Configuration element to search Windows registry entries for the location of the <i>jvm.dll</i> file.

**DIRECTORY (IN SEARCHSEQUENCE)**

Element	Content
directory	Configuration element to search the specified directory for a <i>jvm.dll</i> file. Specify the home directory of a Java Runtime Environment installation here (the one that contains subdirectories <i>bin</i> and <i>lib</i> ).

	Attribute	Content
1	name	The directory name

**REGISTRY**

Element	Content
registry	Configuration element to search Windows registry entries for the location of the <i>jvm.dll</i> file.

**INIT (IN JAVA)**

Element	Content
init	Configuration element for settings pertaining to VM initialization. Java class path and VM options go here.

·	Attribute	Content
	Element	Content
	classpath	Configuration element for the composition of the Java class path.
*	vmoption	Configuration element for an option to pass in JVM creation.

**CLASSPATH (IN JAVA.INIT)**

Element	Content																		
classpath	<p>Configuration element for the composition of the Java class path. Sets the system property <i>java.class.path</i>. Child entries are processed in sequence and the resulting strings are appended to the property value.</p> <p>If this configuration element is omitted a default classpath will be created. If this configuration element is present it replaces the default class path (to only add entries to the default class path use the <i>default</i> child entry together with the additional child entries).</p>																		
	<table><tr><th></th><th>Attribute</th><th>Content</th></tr><tr><td></td><th>Element</th><th>Content</th></tr><tr><td>?</td><td>archive</td><td>Single archive file to append to the class path.</td></tr><tr><td>?</td><td>default</td><td>The class path as it would have been had the whole <i>classpath</i> configuration element been omitted.</td></tr><tr><td>?</td><td>directory</td><td>Single directory to append to the class path.</td></tr><tr><td>?</td><td>scanDirectory</td><td>Directory to scan for archive files (<i>*.jar</i>) to append to the class path.</td></tr></table>		Attribute	Content		Element	Content	?	archive	Single archive file to append to the class path.	?	default	The class path as it would have been had the whole <i>classpath</i> configuration element been omitted.	?	directory	Single directory to append to the class path.	?	scanDirectory	Directory to scan for archive files ( <i>*.jar</i> ) to append to the class path.
	Attribute	Content																	
	Element	Content																	
?	archive	Single archive file to append to the class path.																	
?	default	The class path as it would have been had the whole <i>classpath</i> configuration element been omitted.																	
?	directory	Single directory to append to the class path.																	
?	scanDirectory	Directory to scan for archive files ( <i>*.jar</i> ) to append to the class path.																	



**ARCHIVE**

Element	Content						
archive	Single archive file to append to the class path.						
.	<table><tr><th></th><th>Attribute</th><th>Content</th></tr><tr><td>1</td><td>name</td><td>The archive file name</td></tr></table>		Attribute	Content	1	name	The archive file name
	Attribute	Content					
1	name	The archive file name					

**DEFAULT**

Element	Content
default	The class path as it would have been had the whole <i>classpath</i> configuration element been omitted.

**DIRECTORY (IN CLASSPATH)**

Element	Content		
directory	Single directory to append to the class path.		
.		Attribute	Content
1	name	The directory name	

**SCANDIRECTORY**

Element	Content
scanDirectory	Directory to scan for archive files ( <i>*.jar</i> ) to append to the class path. All files with the extension <i>jar</i> in the directory will be appended to the class path.

		Attribute	Content
--	--	-----------	---------

1	name	The directory name
---	------	--------------------

#### NATIVELIBRARYDIRECTORIES

Element	Content
---------	---------

nativeLibraryDirectories Configuration element for the composition of the Java class path. Sets the system property *java.library.path*. Child entries are processed in sequence and the resulting strings are appended to the property value.

		Attribute	Content
		Element	Content

directory	Single directory to append to the native library path
-----------	---

#### DIRECTORY (IN NATIVELIBRARYDIRECTORIES)

Element	Content
---------	---------

directory	Single directory to append to the native library path.
-----------	--

		Attribute	Content
--	--	-----------	---------

1	name	The directory name
---	------	--------------------

#### VMOPTION

Element	Content
---------	---------

vmoption	Configuration element for an option to pass in JVM initialization.	
----------	--	--

.		<b>Attribute</b>	<b>Content</b>
---	--	------------------	----------------

? value The VM option value.

.		<b>Element</b>	<b>Content</b>
---	--	----------------	----------------

#### CONTEXT

<b>Element</b>	<b>Content</b>
----------------	----------------

context Configuration element for initializing an application context. This configuration element represents a common context in the *ijlauncher.dll.lcnf* and a special context derived from the common context in the configurations of all libraries using *ijlauncher*. Each context will get its own class loader. If you use native libraries, make sure the corresponding java code is in the class path of the common context.

.		<b>Attribute</b>	<b>Content</b>
---	--	------------------	----------------

.		<b>Element</b>	<b>Content</b>
---	--	----------------	----------------

? init Configuration element for settings pertaining to context initialization.

? splashScreen Configuration element for splash screen display.

#### INIT (IN CONTEXT)

<b>Element</b>	<b>Content</b>
----------------	----------------

init	Configuration element for settings pertaining to context initialization. Entries for context class loader and startup arguments go here.
------	--

.	Element	Content
*	arg	Configuration element for an argument to pass to context startup.
	classpath	Configuration element for the composition of the context class path.

#### CLASSPATH (IN CONTEXT.INIT)

Element	Content
---------	---------

classpath	Configuration element for the composition of the context class path. Configures a context's class loader. Child entries are processed in sequence and the resulting strings are appended to the class loader's URLs to search.
-----------	--

If this configuration element is omitted a default class path will be created. If this configuration element is present it replaces the default class path (to only add entries to the default class path use the *default* child entry together with the additional child entries).

.	Attribute	Content
---	-----------	---------

.	Element	Content
---	---------	---------

?	archive	Single archive file to append to the class path.
---	---------	--

?	default	The class path as it would have been had the whole <i>classpath</i> configuration element been omitted.
---	---------	---

?	directory	Single directory to append to the class path.
---	-----------	---

---

?	scanDirectory	Directory to scan for archive files (*.jar) to append to the class path.
---	---------------	--

---

**ARG**

Element	Content
arg	Configuration element for an argument to pass to context startup. The arguments are passed in the same order as they occur in the configuration file.

---

.	Attribute	Content
1	value	The argument value.

---

.	Element	Content
---	---------	---------

---

**SPLASHSCREEN**

Element	Content
splashScreen	Configuration element for splash screen display

---

.	Attribute	Content
	bitmapFile	Name of an image file to display where appropriate.

---

Sample files:

```

<?xml version="1.0" encoding="utf-8"?>
<launchconfig>
  <java>
    <init>
      <searchSequence>
        <directory name="jre" />
        <registry />
      </searchSequence>
      <classPath>
        <scanDirectory name="lib" />
      </classPath>
      <nativeLibraryDirectories>
        <directory name="bin" />
      </nativeLibraryDirectories>
      <vmoption value="-
Djava.system.class.loader=de.intarsys.jnicallin.SystemClassLoader" />
    </init>
  </java>
  <context>
    <init>
      <classPath>
        <archive name="jna.jar" />
        <archive name="swt.jar" />
      </classPath>
    </init>
  </context>
</launchconfig>
<?xml version="1.0" encoding="utf-8"?>
<launchconfig>
  <context>
    <init>
      <classPath>
        <scanDirectory name="lib" />
      </classPath>
      <arg value="-launch" />
      <arg value="activedoc" />
    </init>
    <splashScreen bitmapFile="splash.bmp" />
  </context>
</launchconfig>

```

## 2.4 Preferences

### 2.4.1 Overview

The preferences are used for storing system and user defined settings for various application functions, like window positions, last loaded documents or even server settings.

The preferences are loaded in two separate steps:

- **System preferences** System preferences are stored in the folder *preferences* in the *basedir* of the application. The *basedir* is the installation folder, but can be changed through the configuration file (see above).

- **User preferences** User preferences are stored in the folder *preferences* in the *profiledir* of the application. The *profiledir* is usually the OS specific user folder, for example under Windows *.../Documents and Settings/<user>/SignLiveCC\_<application version>*. This can be changed through the configuration file (see above).

User preferences take precedence over system preferences.

## 2.4.2 Export preferences

There is no way to export preferences as they already reside in *exported* form as *\*.prefs* files on your system. You can select the preferences you are interested in transporting and move them to a different system using the following import methods.

## 2.4.3 Import preferences

The simplest way to import preferences is to drag a *\*.prefs* file into the application GUI. You can also open it using the “Open” menu or using the CLI to invoke Sign Live! CC with appropriate commands.

Upon importing, the new preferences are merged with the existing preferences and immediately written back. They are used the next time the application is started.

## 2.4.4 Tip

Under certain circumstances a preference can cause malfunctions during startup or while working with the application. You can delete the folder *../preferences* in your *profiledir* in such a case - the application will use defaults from now on and can you resume working normally.

# 2.5 Licenses

## 2.5.1 Overview

Some functional parts of Sign Live! CC are only usable with valid licenses. These licenses are delivered as *\*.lic* files. You can drag&drop the file on your application or open them with the “File->Open” menu. After restarting the corresponding functions are enabled.

## 2.5.2 Importing licenses

The simplest way to import licenses is to drag a *\*.lic* file into the application GUI. You can also open it using the “File->Open” menu or using the CLI to invoke the application with appropriate commands.

The license file is stored in the license import folder, which is related to the application base directory. If the user doesn't have write access, the license file is stored in the profile directory of the user instead. If this

happens, the license is loaded individually per user and does not work for other users.

## 2.6 Extension Points

### 2.6.1 Overview

The application is built on the platform (claptz) and enhanced by several instruments. Each of these instruments has a specific function, like digitally signing documents, viewing or handling licenses. The platform bundles all these functions together.

The configuration of the platform is therefore defined by adding, removing or changing instruments. The result is the functional capacity of the application.

Most instruments have so called **Extension Points**, which help you to extend their functionality.

By changing these Extension Points in existing instruments or by defining new Extension Points in your instruments you are able to customize the resulting application.

The two chapters *Logging* and *Monitor* describe two common tasks where instruments are extended. In the chapter above you have already written your own instrument using an extension point to load license files.

### 2.6.2 Logging

#### 2.6.2.1 Overview

The application is instrumented to provide detailed logging information. In case of problems you can lookup information in the log file and send it to our support along with your request.

Logging can be enabled, disabled and configured in a variety of ways. First of all, for you comfortable with the standard Java logging facilities: the application uses the standard Java logging facilities...

#### 2.6.2.2 Default configuration

The application uses a hardwired, built in default configuration. This configuration will create a rolling log file in your user profile directory. All events at level "INFO" will be visible there by default.

See Configuration for more information about the *profiledir*.

#### 2.6.2.3 Instrument based customizing

You can create an extension to customize the various logging properties. The configuration entries match those of *java.util.logging*. See the Java documentation for details.

For example you can use the following settings to raise the logging level to FINE for file output.



```

<extension point="de.intarsys.claptz.beans">
  <object class="com.cabaret.claptz.stage.main.LogInstaller">
    <property name="properties">
      <value>
.level=INFO
org.apache.commons.digester.Digester.level=WARNING
org.jcp.xml.dsig.internal.level=WARNING
handlers=java.util.logging.ConsoleHandler
de.intarsys.tools.logging.FileHandler
java.util.logging.ConsoleHandler.level=WARNING
java.util.logging.ConsoleHandler.formatter=de.intarsys.tools.logging.S
impleFormatter
de.intarsys.tools.logging.FileHandler.level=FINE
de.intarsys.tools.logging.FileHandler.formatter=de.intarsys.tools.logg
ing.SimpleFormatter
de.intarsys.tools.logging.FileHandler.limit=100000
de.intarsys.tools.logging.FileHandler.count=10
de.intarsys.tools.logging.FileHandler.pattern=${environment.profiledir
}/stage.vmid_%u.index_%g.log
      </value>
    </property>
  </object>
</extension>

```

Note that you must explicitly add an entity to denote end of line!

#### 2.6.2.4 Plain Java logging

In some circumstances you may want to disable the built in logging configuration, for example when you have set up your VM with a general logging convention. To suppress standard logging configuration simply use the commandline switch `-nologconfig`. Now the application will leave the logging alone, allowing you to revert to the standard Java logging administration.

Example:

```
SignLiveCC.exe -nologconfig
```

#### 2.6.2.5 Notes on Java logging

While the best source of information on logging is the Java documentation, we will provide a short compendium for your convenience here.

The following properties are supported in the logging configuration description in the standard Java properties format.

*handlers* specifies a comma separated list of log handler classes. These handlers will be installed during VM startup. Note that these classes must be on the system classpath.

Example:

```
handlers= java.util.logging.ConsoleHandler
```

Typical log handlers are:

- `java.util.logging.ConsoleHandler`
- `java.util.logging.FileHandler`

*level* specifies which kinds of events are logged. This property can be appended to a handler name or a logger name to limit the output to the defined level.

This example limits console output to warnings but does a very detailed logging in a file.

```
java.util.logging.ConsoleHandler.level=WARNING
java.util.logging.FileHandler.level=FINE
```

To fine tune logging, you can set a level for any logger (category) that is used by the application itself. The logger instances are organized in a hierarchy and child loggers inherit the level of their parents.

The behavior of the root logger is defined simply by appending level to nothing...

```
.level=WARNING
```

The possible level values, in order of level of detail, are:

SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST

#### 2.6.2.6 Standard configuration protocol

If you don't use the built-in logging configuration (`-nologconfig`), the system reverts to the standard logging protocol.

The default logging properties are read from the file "lib/logging.properties" in your Java runtime installation directory. Changes you make here apply to all Java applications.

To direct the application to a special configuration file, use the property `java.util.logging.config.file` when launching the VM.

Example:

```
<exe> -nologconfig -
Djava.util.logging.config.file=c:/temp/logging.properties
```

Do not forget to switch off the internal handling with `-nologconfig`

## 2.7 Monitor Configuration

## 2.7.1 Monitor

### 2.7.1.1 Overview

A useful addition to the logging system is the use of *monitor* objects. These monitor objects are able to trace runtime events for performance logging.

### 2.7.1.2 API

The monitor concepts are similar to those of the basic logging. You can create a monitor with the following API call:

```
monitor = MonitorFactory.getDefault().lookupMonitor(MONITOR_ID);
```

The factory creates a new monitor object according to the configuration provided. If there is no configuration a `NullMonitor` is returned, which has no impact on the runtime.

When you want to start monitoring, use:

```
monitor.start();
```

This starts a `MonitorTrace`. You can add information to this trace later.

```
monitor.getCurrentTrace().tag("user", "me");  
...  
monitor.getCurrentTrace().sample("about to foo");  
...  
monitor.getCurrentTrace().sample("about to bar");  
...
```

If you are finished tracing, you need to stop the trace with:

```
monitor.getCurrentTrace().stop();
```

At this point the trace is written in the log file. The log category is the name of the monitor. A typical output looks like this:

```
[29.3.2007-13:4:56:40]  
[FINE]  
[com.cabaret.platform.cli.monitor]  
[SWT GUI Thread]  
[Trace=com.cabaret.platform.cli.monitor]  
[start=29.3.2007-13:4:56:40;stop=0]  
[total=0;concurrent=0]
```

You can aggregate all traces created by the monitor very easily by creating a table as a `String`:

```
stats = monitor.toString();
```

The result looks like this:

```
com.cabaret.application.pdf.common.PDFCreatorFactory
```

description	count	total	total %	min	max	avg
started	7	0	0.0	0	0	0
stop	7	641	100.0	15	531	91
	7	641	100	15	531	91

As you see, the monitor is an easy to use tool for giving your code the capability of creating traces. You can start these monitor traces by providing configurations, GUI actions or JMX access. The application provides several monitor types and configurations. A standard configuration for a time measurement (TimeMonitor) is described below.

#### 2.7.1.3 Logging

The monitor creates log output at the end of one trace or aggregation. These are written on the logger associated with the monitors name by default. You can adapt this specific logger using the configurations. The log level is *FINE* per default.

You can also adjust the amount of aggregation to be done. *logCycle* defines the number of traces after which an aggregation is created.

#### 2.7.1.4 Configuration

A monitor is defined using the basic extension point *de.intarsys.claptz.beans*. The *object* element defines a single monitor within this extension point.

```
<extension point="de.intarsys.claptz.beans">
  <object
    class="de.intarsys.tools.monitor.TimeMonitor"
    bean-id="BatchMonitor"/>
</extension>
```

### MONITOR

Element	Content		
monitor	The monitor configuration attributes.		
.	<table> <tr> <th>Attribute</th><th>Content</th></tr> </table>	Attribute	Content
Attribute	Content		

class	The monitor type to be used, typically <i>de.intarsys.tools.monitor.TimeMonitor</i> .
name	The logical name of the monitor.
logger	The log category on which output is created. The default is the name of the monitor.
level	The log level to be used by this monitor's output.
logcycle	The number of traces needed to start an aggregation. The default is 100.

#### 2.7.1.5 Standard monitors

The application has several predefined monitors. These can be activated using the above configuration to create detailed runtime information.

The following monitor objects are predefined:

- *de.intarsys.tools.cli.processor*  
Traces CLI calls.
- *<Processor Factory ID>*  
All processors carry their own monitor with their factory ID.



## 3. Variables

---

### 3.1 Introduction

Sign Live! CC supports the definition of variables for the ease of configuration and customization. A variable is a simple mapping from a name to a string value. For example if you have the value “c:\temp”, denoting a directory, you can define a variable “tempDir” and set it to this value. Using the variable will give you the value stored in it, “c:\temp”.

The benefit of this indirection is that from now on you can simply change “tempDir” at a single point in your configuration and you don’t need to adapt scripts or definitions.

Variables are tightly integrated with other parts of the application, enabling you to store them in preferences, edit them in preference pages and use them in **string variable expansion** or scripting.

For more information about the programming API for variables, see the “Developer’s Guide”.

### 3.2 Concepts

A **variable** is a simple placeholder for a string value. A variable can be assigned and read.

Variables are organized in **namespaces** to allow for modularization. A namespace is simply a named collection of variables.

### 3.3 Configuration

Variables and their namespaces are defined using the extension point *de.intarsys.claptz.variables*. This extension point is defined in the instrument *com.cabaret.claptz.common*, so the minimum requirement if you define an instrument using this is

```
<requires>  
  <prerequisite instrument="com.cabaret.claptz.common"/>  
</requires>
```

This extension point allows for the definition of namespaces and the definition of variables contained in them. The namespaces are defined along with a reference to a preferences path and scope. This way you can define where to store (and later on reload) changes made to the variables using the preferences pages or some other means.

This example introduces the namespace “mycompany”. Persistent values are stored in the path “/com.mycompany.prefs/variables” on a per user scope.

```
<extension point="de.intarsys.claptz.variables">  
  <namespace id="mycompany"  
    preferences="/com.mycompany.prefs/variables" preferencesscope="USER"/>  
</extension>
```

A variable is defined as belonging to a namespace and having a key / value pair.

This example defines the variable “webserver” in the above namespace with the initial value “www.mycompany.com”.

```
<extension point="de.intarsys.claptz.variables">  
  <variable namespace="mycompany" name="webserver"  
    value="www.mycompany.com"/>  
</extension>
```

When the variable is read for the first time, “www.mycompany.com” is returned. If later by some interaction, for example in the preferences page for variables, the value is set to “www.mycompany.org”, this value is available from now on and after each restart.

## 3.4 Editing

### 3.4.1 Edit the extension point

You can easily edit the extension point *de.intarsys.claptz.variables* to add or change variable values. Setting a variable that already exists in another definition simply overwrites its value.

Most of the times you should not change the original file to allow for easy upgrade of your software. Simply define a new instrument, make the instrument with the definition you want to change a required one and redefine the variable (take care of using the correct scope).



### 3.4.2 Declare preferences

As variables are mapped to preferences, setting values in the correct preferences tree will show up as variable values.

Preferences are configured using the extension point *com.cabaret.claptz.preferences.declarations*.

This example will set “foo” to “bar” in the above defined namespace.

```
<node name="/com.mycompany.prefs">
  <node name="variables">
    <property name="foo" value="bar"/>
  </node>
</node>
```

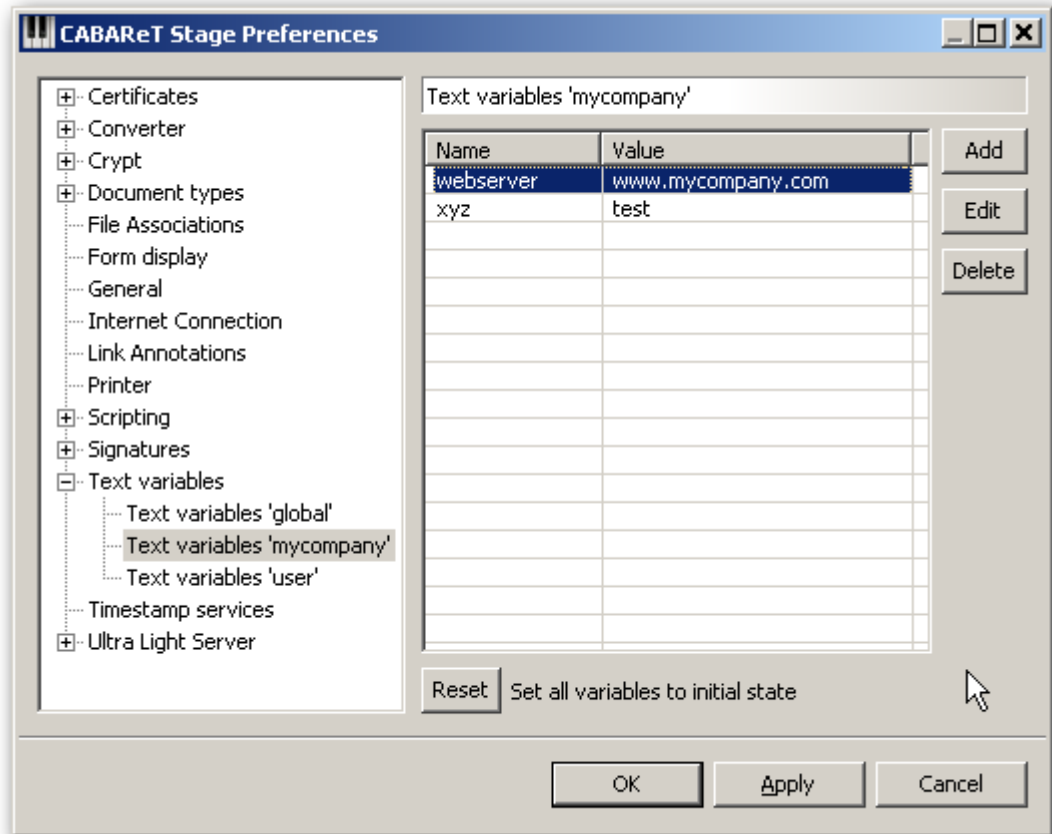
### 3.4.3 Use the variable GUI editor

For interactive editing of your variables you can add a page to the Sign Live! CC preferences. Your namespace will show up in a subpage of “Text variables” in the applications settings dialog.

The declaration for a new preferences page is easy, you don’t even need new required instruments

```
<extension point="com.cabaret.claptz.preferences.pagefactories">
  <pagefactory
    class="com.cabaret.claptz.common.preferences.GenericPreferencesPageFactory"
    pageclass="com.cabaret.claptz.common.variables.swt.VariablesNamespacePreferences"
    id="mycompany"
    namespace="mycompany"
    parent="com.cabaret.claptz.common.variables.swt.VariablesRootPreferences"/>
</extension>
```

Now in the settings dialog for the application you will find a new subpage for your variable namespace “mycompany”.



Here you can Add new variable mappings or Edit and Delete existing ones. All your changes are made persistent in the corresponding preferences.

If you need to “forget” all your persistent changes, you can Reset to the initial state. This means simply that all the associated preferences are cleared.

#### 3.4.4 Import a preferences file

Another easy way for editing and defining variables is simply take an existing preferences file and import it in your system. Importing is done by simply opening a “.prefs” file using the file menu or drag&drop.

### 3.5 String expansion

One of the most common uses of variables is in string expansion. The chapter “String Expansion” will bring you more detailed information on the topic later on.

Here we will look at a typical example. Suppose you want to add a new standard location to look for PDF stamps in your company. Your operator will supply this on a network drive “z:/mycompany/stamps”. You will add a new “Stamp Library” pointing to this location

```
<extension point="com.cabaret.markup.stamp.libraries">
  <load path="z:/mycompany/stamps" recursive="true"/>
</extension>
```

Now, if some workgroup needs to redirect this to an offline copy, it has to add a declaration or change the existing one. You can leverage variables here:

```
<extension point="de.intarsys.claptz.variables">
  <variable namespace="mycompany" name="stamps"
value="z:/mycompany/stamps"/>
</extension>

<extension point="com.cabaret.markup.stamp.libraries">
  <load path="${variables.mycompany.stamps}" recursive="true"/>
</extension>
```

You introduce an indirection that allows for changing the path in the preferences of the application instead of the hardcoded *instrument.xml* file. Still you need to restart to have your changes active in this example! Be sure to define your new variable before you use it!

## 3.6 Script access

You can access variables in the scripting environment of Sign Live! CC. More information on this you will find in the “Developer’s Guide” chapter on „Scripting“.

The global variable holding all variables namespaces is *jVariables*.

Reading a variable value in JavaScript can be done this way

```
...
var tempValue =
jVariables.getNamespace("mycompany").getVariable("webserver");
...
```

This flavor will return the *defaultValue* in case the variable is not available

```
...
var tempValue =
jVariables.getNamespace("mycompany").getVariable("webserver",
"www.defaultsite.com");
...
```

This example will write to the variable

```
...  
jVariables.getNamespace("mycompany").putVariable("myvar", "been  
here");  
...
```

## 4. Automation

---

### 4.1 Commandline API

The Commandline API of Sign Live! CC allows for easy integration into existing processes. All of the platform's functions are accessible with OS commands for other applications. You will not need Java or a development platform.

Advantages:

- Simple adhoc integration
- Access to all features using command line scripting

Disadvantages:

- No return values
- Asynchronous processing

### 4.2 ActiveX

Please note that this API is not available for all product variants.

For the Windows platform we provide a native integration using the COM standard. You can customize nearly any scenario and publish it yourself via the generic ActiveX API.

### 4.3 HTTP

You can integrate Sign Live! CC using the service framework. Internally based on the popular Jetty container engine you can publish predefined or custom services via plain HTTP or XMLRPC.

### 4.4 Web Services

To integrate Sign Live! CC in existing SOA Architectures a “SOAP” connector is available in the service framework. It is again based on Jetty and the Apache CXF libraries.

## 4.5 File monitoring

A synchronous coupling is often not wanted or impossible. A scenario often seen then is based on file system monitoring. The originating system produces a file output to a directory monitored by Sign Live! CC. After detecting the file, it is forwarded to the configured workflow in the service framework.

## 4.6 Scheduler

For time triggered activities a scheduler is included in the service framework

## 4.7 Printing

To create workflows based on printed documents ( a typical ERP scenario), a “Network Printer” is included in the service framework.

## 4.8 Message queues

You can integrate Sign Live! CC in message queues using the JMS connector for the service framework. This is tested for ActiveMQ only.

## 4.9 Batch

A Batch allows the execution of defined actions on a defined number of input elements, for example, on a folder, a list of files or on the currently open documents.

The actions can be predefined functions (from instruments), like signature creation or scripts you created yourself using the scripting framework. You can use the full API of Sign Live! CC within scripts for batch processing.

A batch can be created by handcrafting XML definitions with a text editor or using the built in GUI batch editor within Sign Live! CC .

## 5. Commandline Interface

### 5.1 Commandline Interface (CLI)

#### 5.1.1 Overview

Sign Live! CC can be started using the command line with a list of parameters. By using CLI parameters you can define everything from environment settings, viewing, printing and processing options to chaining scripts and processors.

If you use the native launcher to start Sign Live! CC, only one instance can be running at any given time. This means other CLI calls are delegated to the currently running instance.

The application distinguishes between two option sets: Platform options and Application options.

#### 5.1.2 Return Values

The application does not return state values for the processed operations. If you want to integrate Sign Live! CC and rely on states, you may want to look at other integration options like the API or the service framework.

This is a limitation of the currently used launcher and may change in future releases.

#### 5.1.3 Error Handling

CLI processing stops when an error is encountered. The order for processing the options is as follows:

- platform options
- application options in order of input

If the CLI processing stops, Sign Live! CC is not necessarily shut down. You can consult the log file for the problem encountered and the state of the application.

#### 5.1.4 Scripting Integration

A particularly powerful feature is the option to execute scripts with the CLI. You can do this by referencing a script file or by putting the actual code in the CLI call. Using this you can access all the functions of the application and its API.

The scripting framework has its own guide and a scripting tutorial, which also explains calls from the command line.

### 5.1.5 Option chaining

You can chain CLI options. Using this method you can launch more complex processes. The order of the options is relevant.

Example:

Load a file, import data, print it, save it, don't view.

```
<exe>
-f c:\demo\Document.pdf
-im c:\demo\Document.xfdf
-p
-s c:\demo\Dokcment_with_Data.pdf
-pop
```

### 5.1.6 CLI Context

The CLI has no knowledge of previous processing calls, which means a CLI call must describe the complete processing task.

CLI chains like this will not work.

```
<exe> -f c:\demo\document.pdf
<exe> -p
```

### 5.1.7 Hiding option values

The commandline used to call into Sign Live! CC is included in the log for the ease of monitoring and debugging. But often there is information not intended to be logged in plain text. In these cases just use the equivalents of - and - - that hide the option values: ~ and ~ ~.

The commandline call

```
<exe> ... -password foo ...
```

will create this log entry

```
[...] IMain.launch (... -password foo...)
```

The commandline call

```
<exe> ... ~password foo ...
```

will hide the password value



```
[...] IMain.launch (... -password *** ..)
```

## 5.2 CLI Platform Options

The platform options configure the startup and runtime behavior of the application. These options are parsed independent from their position in the CLI chain.

### 5.2.1 Profile directory

Sign Live! CC may use a dedicated directory for storing user specific information. By default this is the folder “.SignLiveCC” in the user’s home directory. You can change this directory using the profile option. The option value is expanded before use.

<b>Option (short)</b>	-profile
<b>Option (long)</b>	--profile
<b>Description</b>	A directory for storing user specific information.
<b>Argument</b>	Directory name
<b>Example</b>	-profile “c:/temp/preferences”

### 5.2.2 Search Path for Instruments

Sign Live! CC allows instruments to be loaded from several path locations at system startup.

The default instrument paths are always loaded in addition to the directories named here.

<b>Option (short)</b>	-itdirs
<b>Option (long)</b>	--itdirs
<b>Description</b>	Semicolon (;) separated list of folders to be searched for instrument definitions.

<b>Argument</b>	List of folders
<b>Example</b>	-itdirs "c:\myinstruments;x:\standardinstruments"

### 5.2.3 Strict instrument loading

You can disable the loading of the default instrument search paths with the option "itstrict".

The default instrument search paths are:

- The *instruments* folder in the base dir of Sign Live! CC
- The *instruments* folder in the profile dir of the user

<b>Option (short)</b>	-itstrict
<b>Option (long)</b>	--itstrict
<b>Description</b>	Disable loading default instruments
<b>Example</b>	-itdirs "c:\myinstruments;x:\standardinstruments" -itstrict

### 5.2.4 Log configuration

Sign Live! CC overrides the standard log configuration of the Java VM to allow for a useful default configuration and easy instrument based override. If this behavior is not what you want and you want to revert to the standard Java logging protocol, you can switch of the Sign Live! CC logging configuration.

<b>Option (short)</b>	-nologconfig
<b>Option (long)</b>	--nologconfig
<b>Description</b>	Switch of built in logging configuration
<b>Argument</b>	-

<b>Example</b>	<code>-nologconfig</code>
----------------	---------------------------

### 5.2.5 Configuration file

Sign Live! CC can be started using a configuration file. This is discussed in detail in the Configuration chapter.

If you do not provide a configuration file, the file found under *config/stage.config* will be used. If this file is not found, the default configuration will be used.

<b>Option (short)</b>	<code>-config</code>
<b>Option (long)</b>	<code>--config</code>
<b>Description</b>	A configuration file for the application
<b>Argument</b>	File name
<b>Example</b>	<code>-config "config/special.config"</code>

### 5.2.6 Launch Mode

Sign Live! CC can be started in different modes where every mode stands for a dedicated application scenario. The following modes are supported by default:

#### 5.2.6.1 Interactive (GUI) Mode

Sign Live! CC will be started in *interactive* mode with the GUI enabled. This is the default launch mode if no other is selected.

By default, the application will block and dispatch GUI events until it is closed. To disable this behavior when starting in embedded context where you control the event dispatching yourself, use the *noblock* option flag.

#### 5.2.6.2 Headless mode

The *headless* mode starts Sign Live! CC without the graphical user interface. For Unix systems an X Server or an equivalent solution is still needed. The application does not present a GUI but still needs to call X related functions for some tasks.

This option supports no interaction with the user. A typical use is executing CLI calls or running server based processes, like file monitoring.

You need a *Professional* license for this mode to be enabled.

By default, running “headless” mode will use blocking behavior, this means that the launch will not return until the application end.

You can fine tune this behavior using the “!block” or “noblock” option. If you choose non blocking behavior, the headless application will terminate immediately after executing the commandline. If you want the application to stay alive, you can add the “resident” option. .

#### 5.2.6.3 Active Document

This mode allows the embedded use of Sign Live! CC within an ActiveX document container. You should not use this mode from the commandline or embedding applications.

<b>Option (short)</b>	-launch
<b>Option (long)</b>	--launch
<b>Description</b>	Definition of the Sign Live! CC working mode
<b>Argument</b>	Name of the working mode: interactive [noblock] headless [block] activedoc
<b>Example</b>	-launch headless !block resident

#### 5.2.7 Disable application launch

In some embedding scenarios only the platform initialization is needed and you may want to disable the default application launch. You can do this using the “nolaunch” option.

<b>Option (short)</b>	-nolaunch
<b>Option (long)</b>	--nolaunch

<b>Description</b>	Disable application launching
<b>Example</b>	-nolaunch

### 5.2.8 Window creation & activation

Upon startup an initial window will be created and mapped to the display. In much the same way a new window will be created if none is available and a document viewer is opened. In some command line driven scenarios this may be undesirable, so here are options to control this behavior.

These options are evaluated for each command line call and remain in effect up to the next call.

Under windows this option can not prevent an existing window to be restored and activated. This is a “feature” of the native launcher that you can only workaround by not having a window at all and combine with the “nowakeonstart” to prevent creating one. This way you will not see a window come up upon each commandline call.

<b>Option (short)</b>	-nowakeonstart
<b>Option (long)</b>	--nowakeonstart
<b>Description</b>	Suppress the automatic creation of a window upon startup.
<b>Argument</b>	-
<b>Example</b>	-nowakeonstart
<b>Option (short)</b>	-nowakeonview
<b>Option (long)</b>	--nowakeonview
<b>Description</b>	Suppress the automatic creation of a window when a document viewer is requested.

<b>Argument</b>	-
<b>Example</b>	-nowakeonview

## 5.2.9 Command Line File

All the options available for the command line can make for a rather large call. Therefore there is another way to include these options, the *command line file*. You can specify all options in this file and invoke Sign Live! CC using this file instead of a string of options. If you use a command line file all other options on the CLI are ignored.

The syntax of the command line file is the same as the normal CLI options, but you can use line breaks and comments, preceded with the # character. Every line starting with # is ignored.

The contents of the command file are expanded before use, you can use the standard string expansion variables. This allows you to easily forward batch arguments to the commandfile.

Example:

```
<exe> -cf commands.cli
```

This will start the application and load the CLI options from the command line file *commands.cli*. This file may look like this (example):

```
# a sophisticated command line file
-f test.pdf          # load a document
  -p                 # then print it
  -pop               # and discard it
# now start something else...
```

<b>Option (short)</b>	-cf
<b>Option (long)</b>	--commandfile
<b>Description</b>	The name of a file containing CLI options.
<b>Argument</b>	filename [options]*

**Example****-cf** commands.cli

The option “-cf” is available for string expansions using “options”. You can access the options indexed, starting at 0 with the file itself, or name if you have specified the option in a “option=value” style.

This example is a batch file *batch.cmd*, using indexed options

```
<exe> -cli foo.cli "%1" "bar.pdf"
```

It provides the shell argument %1 and the string *bar.pdf*, using a numeric index. With this command file *foo.cli*

```
-f ${options.1}  
-s ${options.2}
```

calling the batch file *batch.cmd my.pdf* will expand to

```
-f my.pdf  
-s bar.pdf
```

In much the same way you can use named options. This example is a batch file *batch.cmd*, using named options:

```
<exe> -cli foo.cli "in=%1" "out=bar.pdf"
```

It provides the shell argument %1 and the string *bar.pdf*, using named variables. With this command file *foo.cli*

```
-f ${options.in}  
-s ${options.out}
```

calling the batch file *batch.cmd my.pdf* will expand to

```
-f my.pdf  
-s bar.pdf
```

## 5.3 CLI Application Options

### 5.3.1 Overview

After processing the platform options the application options provided by the CLI call are parsed. These options are parsed in order of

appearance in the CLI. You can therefore access results from previous options. You can chain these options together to create complex CLI calls.

This means

```
<exe> -f c:\demo\Document.pdf -p
```

is not the same as

```
<exe> -p -f c:\demo\Document.pdf
```

The availability of certain application options is dependent upon the instruments used. The option for importing data is, for example, only available if the import instrument has been loaded.

Some options are license protected and you will need the appropriate license to use them.

### 5.3.2 Options

The following options are available and described in their own chapters.

- Standard Application Options
- PDF Options
- Exchange Options
- Scripting Options

### 5.3.3 Working Stack

The processing of the CLI options operates on an internal working stack where all objects are stored. This stack works first-in/last-out, as some scientific calculators do.

Stack oriented processing has an impact on the order of processing and on the stored objects. Which objects are on the stack depends on your CLI options. Examples for stored objects are:

- documents
- fields

The objects on the stack are bound by type. This means you can only invoke a save command if the object on top of the stack is a document. If there is a field object on top of the stack an error will occur.

The command line



```
<exe> -f test.pdf -p -aff textfield -affsv test
```

opens a file, prints it and changes the value of the field *textfield*.

```
<exe> -f test.pdf -aff textfield -affsv test -p
```

This command line opens a file, changes the value of the field *textfield* and then prints.

### 5.3.4 Standard Application Options

#### 5.3.4.1 Stack operators

Remove the top element from the stack

<b>Option (short)</b>	-pop
-----------------------	------

<b>Option (long)</b>	--pop
----------------------	-------

<b>Description</b>	Remove the top element from the stack
--------------------	---------------------------------------

Remove all elements from the stack

<b>Option (short)</b>	-popall
-----------------------	---------

<b>Option (long)</b>	--popall
----------------------	----------

<b>Description</b>	Remove all elements from the stack
--------------------	------------------------------------

Copy the top element of the stack

<b>Option (short)</b>	-dup
-----------------------	------

<b>Option (long)</b>	--dup
----------------------	-------

<b>Description</b>	Copy the top element of the stack.
--------------------	------------------------------------

#### 5.3.4.2 Window and interaction control

##### 5.3.4.2.1 No user interaction

Sign Live! CC can be executed without user interaction in the GUI mode with the option *silent*. All user interactions, like messages boxes or file dialogs, are suppressed and use default values.

This option remains active until a change is made using the option *silent* again.

<b>Option (short)</b>	-silent
<b>Option (long)</b>	--silent
<b>Description</b>	Run with or without user interaction. The stack remains untouched.
<b>Argument</b>	[on off]
<b>Example</b>	-silent on

#### 5.3.4.2.2 Window control

You can control the current application window to improve the visual feedback for your user.

<b>Option (short)</b>	-window
<b>Option (long)</b>	--window
<b>Description</b>	Control the current application window. The stack remains untouched.
<b>Argument</b>	[maximize minimize restore activate close closeRequested]
<b>Example</b>	-silent on

The effect of the option argument is as follows:

- maximize  
Maximize the current application window.
- minimize  
Minimize the current application window.

- **restore**  
Restore the current application window to its normal size.
- **activate**  
Active the current application window (bring it to front).
- **close**  
Close the current application window. Depending on the number of windows and the current application settings this will close the application, too.
- **closeRequested**  
Request to close the current application window. The window will be closed if the close operation is not rejected. Depending on the number of windows and the current application settings this will close the application, too.

#### 5.3.4.3 Application control

##### 5.3.4.3.1 Exit the application Exit the target application.

<b>Option (short)</b>	-e
<b>Option (long)</b>	--exit
<b>Description</b>	Exit the application. If <b>request</b> argument is present, the exit may be cancelled by user interaction (for example when there are unsaved documents). By default, the application exits unconditionally. Be aware that if combined with “—silent”, the default for “save changes” is “false”.
<b>Argument</b>	["request"]
<b>Example</b>	--exit
<b>Example</b>	--exit request

#### 5.3.4.4 Document control

##### 5.3.4.4.1 Load a document Loading a file does not imply the viewing of this document. The file is loaded into the application and CLI context and put on the stack. The default behavior of Sign Live! CC is to open a viewer after all CLI options have been processed, this can be prohibited using the appropriate option.

<b>Option (short)</b>	-f
<b>Option (long)</b>	--file
<b>Description</b>	Loads a file and puts the resulting document on the stack.
<b>Argument</b>	Filename
<b>Example</b>	-f "temp/test.pdf"

To shorten standard CLI calls, the file option *-f* can be omitted.

```
<exe> -f "c:\demo\document.pdf"
```

is the same as

```
<exe> c:\demo\document.pdf
```

The file is loaded and opened in the viewer.

#### 5.3.4.4.2 Use the active document

The active document, opened in Sign Live! CC, meaning the one in the active viewer window, is put on the stack.

<b>Option (short)</b>	-cd
<b>Option (long)</b>	--currentdocument
<b>Description</b>	The active document is put on the stack.

#### 5.3.4.4.3 Lock the document

Locking a document prohibits write access to this file. For example, this forces the choice of a new save location upon saving.

<b>Option (short)</b>	-lock
-----------------------	-------

<b>Option (long)</b>	--lock
<b>Description</b>	The document on the stack is locked. The stack remains unchanged.
<b>Example</b>	-f "temp/test.pdf" -lock

#### 5.3.4.4.4 Print a document

<b>Option (short)</b>	-p
<b>Option (long)</b>	--print
<b>Description</b>	The document on the stack is printed. The stack remains unchanged.

The following additions can extend the *-print* options.

<b>Option (short)</b>	-pd
<b>Option (long)</b>	--printdialog
<b>Description</b>	Opens the printer dialog before printing.
<b>Example</b>	-f c:\demo\Dokument.pdf -p

The example loads the document, prints it without showing the printer dialog and opens it for viewing, since it is still on the stack.

Using this

```
<exe> -f c:\demo\Dokument.pdf -p -pop
```

the document will not be opened for viewing, since it is no longer on the stack.

```
<exe> -fc -p
```

This prints the active document without showing the printer dialog.

```
<exe> -f c:\demo\Dokument.pdf -p -pd
```

Finally, this loads the document and opens the printer dialog.

#### 5.3.4.4.5 Save a document

<b>Option (short)</b>	-s
<b>Option (long)</b>	--save
<b>Description</b>	The document on the stack is saved. The stack remains unchanged.
<b>Argument (optional)</b>	Filename used for saving

```
<exe> -fc -s
```

This saves the active document using its own filename.

```
<exe> -fc -s c:\demo\Document.pdf
```

This saves the active document using a new filename  
*c:\demo\document.pdf*.

#### 5.3.4.4.6 Save all open documents

<b>Option (short)</b>	-sa
<b>Option (long)</b>	--saveall
<b>Description</b>	Saves all open documents, not necessarily those on the stack. The stack remains unchanged.

### 5.3.5 PDF Options

#### 5.3.5.1 Overview

Sign Live! CC provides means for manipulating PDF forms with the CLI. The fields of the form can be selected and their value be changed. The PDF can then be saved.

For these functions the *Professional* license is required.

In order to do this you will first need to select the form field you are interested in, put it on the stack, and then perform the desired operation with it. The PDF needs to have a form, otherwise an error will occur. The field names within a PDF form can be hierarchical, pay attention to the proper naming notation, like *adress.street*.

#### 5.3.5.2 Select a form field

<b>Option (short)</b>	-aff
<b>Option (long)</b>	--afffield
<b>Description</b>	Selects the given form field and puts it on the stack.
<b>Argument</b>	Name of the form field

#### 5.3.5.3 Change the value of a form field

<b>Option (short)</b>	-affsv
<b>Option (long)</b>	--offsetvalue
<b>Description</b>	Change the value of the form field on the stack to the provided value. The field is removed from the stack.
<b>Argument</b>	New literal value

#### 5.3.5.4 Change the value of a form field with file contents

<b>Option (short)</b>	-affsvr
-----------------------	---------

<b>Option (long)</b>	--offsetvalueref
<b>Description</b>	Change the value of the form field on the stack to the provided content of the provided file. The field is removed from the stack.
<b>Argument</b>	Name of a file whose content is to be used as the new field value.

## 5.3.6 Exchange Options

### 5.3.6.1 Overview

You can import data into PDF forms using Sign Live! CC, create new PDF documents or export data from existing filled PDF forms. Several exchange file formats are supported

You will need the *Professional* license for these options.

The following file formats are supported:

- FDF
- XFDF
- CSV
- HTML

Availability of the given format is dependent on the appropriate instrument being loaded.

### 5.3.6.2 Import

Data can be imported into PDF forms.

You can provide the file name of a PDF document or you can use the referenced document, which can be enclosed in the data. If no file name is used, the active document will be used as the target.

You can provide the file format (import filter). If omitted, the import uses the file's extension to determine the file format.

<b>Option (short)</b>	-im
<b>Option (long)</b>	--import
<b>Description</b>	Starts the import of a file
<b>Argument</b>	The file name to be imported



Several options can extend *import*.

<b>Option (short)</b>	-it
<b>Option (long)</b>	--importtype
<b>Description</b>	Specify the import filter
<b>Argument</b>	Name of the import filter (see list above)
<b>Option (short)</b>	-idata
<b>Option (long)</b>	--idata
<b>Description</b>	Import form data
<b>Option (short)</b>	-iannot
<b>Option (long)</b>	--iannotations
<b>Description</b>	Import annotations

With data file and target document provided:

```
<exe> -f c:\demo\Document.pdf -im c:\demo\Document.xfdf
```

With data file provided, the target document is determined by the import data:

```
<exe> -im c:\demo\Document.xfdf
```

or (omit the *-im*)

```
<exe> c:\demo\Dokument.xfdf
```

### 5.3.6.3 Export

Data contained in a form can be exported. The source document is always the active document.

You can specify the type of export filter (file format). If omitted, the type is determined by the extension of the export file name.

<b>Option (short)</b>	-ex
<b>Option (long)</b>	--export
<b>Description</b>	Starts the export of data in a file
<b>Argument</b>	Target file

Several options can extend *export*.

<b>Option (short)</b>	-et
<b>Option (long)</b>	--exporttype
<b>Description</b>	Specify the export filter
<b>Argument</b>	Name of the export filter (see list above)

```
<exe> -f c:\demo\Document.pdf -ex c:\demo\Document.xfdf
```

Loads the PDF file and exports its data into the specified target.

## 5.3.7 Scripting Options

### 5.3.7.1 Overview

With scripting options you gain access to the full scripting framework of Sign Live! CC. Because of the versatility of the scripts, these options provide a powerful means for the customization and integration of the platform.

The range of actions starts with simple user interactions and the starting of Sign Live! CC processors and does not end with full fledged batch scripts, which are executed in Sign Live! CC.

You will find the options for using the scripting framework from the CLI in this document. To unleash the full potential of the scripting framework, please refer to the “Scripting Tutorial”. You will find a tutorial and all specifications in there.

#### 5.3.7.2 Call a script

<b>Option (short)</b>	-perform
<b>Option (long)</b>	--perform
<b>Description</b>	This calls a script, which is further defined in other options. The result of the script processing is put on the stack.

You need to provide the script code using one of the following options, as actual code or code in an external file. You can do this using:

- actual code in the command line
- providing a file containing the script code
- providing a script name which is located in the standard paths

The following options describe the details of the script call.

<b>Option (short)</b>	-ps
<b>Option (long)</b>	--performsource
<b>Description</b>	Defines the code for the script call. The code type is dependent on the option <i>performtype</i> .
<b>Argument</b>	Script source code
<b>Option (short)</b>	-pt
<b>Option (long)</b>	--performtype

<b>Description</b>	Defines the type of script code
<b>Argument</b>	The script type

The script type can be one of the following:

- **JavaScript**  
A script written in JavaScript
- **Script**  
A script call, which is to be looked up by Sign Live! CC in its search context.
- **ScriptFile**  
A script file denoted by the file name.

#### 5.3.7.3 Parameters

Each call can be accompanied by several parameters. These can then be used with the scripting framework for further processing. The name *event* is not allowed for a parameter name.

You can provide parameters using one of the following methods:

- **Literal**  
The actual value of the parameter in the CLI.
- **Refer to objects on the CLI stack**  
The value will be read from the CLI stack. Using this you can provide created objects to process.
- **Refer to the argument list of the CLI stack**  
The argument list will be read from the CLI stack. Using this you can provide created objects to process.

<b>Option (short)</b>	-pa
<b>Option (long)</b>	--performarg
<b>Description</b>	Defines a parameter to be provided for the script execution. The syntax to be used is "name=value". The <i>value</i> will be provided as a String with the name <i>name</i> for the script.
<b>Argument</b>	name = value
<b>Option (short)</b>	-ppoparg

<b>Option (long)</b>	--performpoparg
<b>Description</b>	Defines a parameter to be provided for the script execution by taking it from the stack. The syntax to be used is “name”. The top element from the stack will be provided under the name <i>name</i> and removed from the stack.
<b>Argument</b>	name
<b>Option (short)</b>	-ppopargs
<b>Option (long)</b>	--performpopargs
<b>Description</b>	Extends the argument list for the script call by the argument list already on the stack. There is no argument for this option, the top element from the stack will be taken as an argument list and removed from the stack.
<b>Argument</b>	none

#### 5.3.7.4 Execution semantics

The execution is forwarded to the CodeExit framework. You will find more information on CodeExit in the “Developer’s Guide”.

The implementation is called with the following named arguments:

- **event**  
The restricted description of the invocation cause.
- **jEvent**  
The Java (LiveConnect) representation of the invocation cause.
- **jCli**  
The commandline processor itself.

Via *jEvent* you have access to

- **target**  
A wrapper processor that gives access to a document on the cli stack (if one available). This is introduced mainly to support the same syntax within scripts to access document arguments regardless of system context.

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

After execution the result from the CodeExit is pushed on the commandline stack.

#### 5.3.7.5 Return values

The return values of the script call will be put on the stack for further processing. You need to use *pop* if the return values are no longer needed for processing.

#### 5.3.7.6 Examples

```
-perform -pt JavaScript -ps "app.alert(var1);" -pa "var1=hi, I am  
var1"  
-e
```

Sign Live! CC is started and a message box appears with the value “hi, I am var1”. This is the content of the parameter *var1* as defined by the option *-pa*. After the message is confirmed, Sign Live! CC is shut down.

```
-perform -pt ScriptFile -ps "demo/scripts/helloworld.js"  
-e
```

Sign Live! CC is started and the script *helloworld.js* is executed. After processing the script, Sign Live! CC is shut down.

```
-f test/cli/batch/docs/1.pdf  
-perform -pt JavaScript -ps "app.alert(event.target.path);" -q  
-e
```

Sign Live! CC is started and opens the file *1.pdf*. A message box will open and show the OS specific name of the document. After the message is confirmed, Sign Live! CC is shut down.

## 6. Service framework

### 6.1 Overview

The service framework is a very powerful automation concept. The primary use is the easy implementation of flexible API calls for fast integration in a great variety of processing scenarios.

It is provided by the application platform and allows for automatic processing, triggered and parameterized from various communication channels.

This implementation supersedes and includes all previous features based on “ULS” and the like. These are no longer supported. While existing configuration do not have to be migrated, we strongly suggest to no longer use the older approach. New features and fixes are only applied to the service framework.

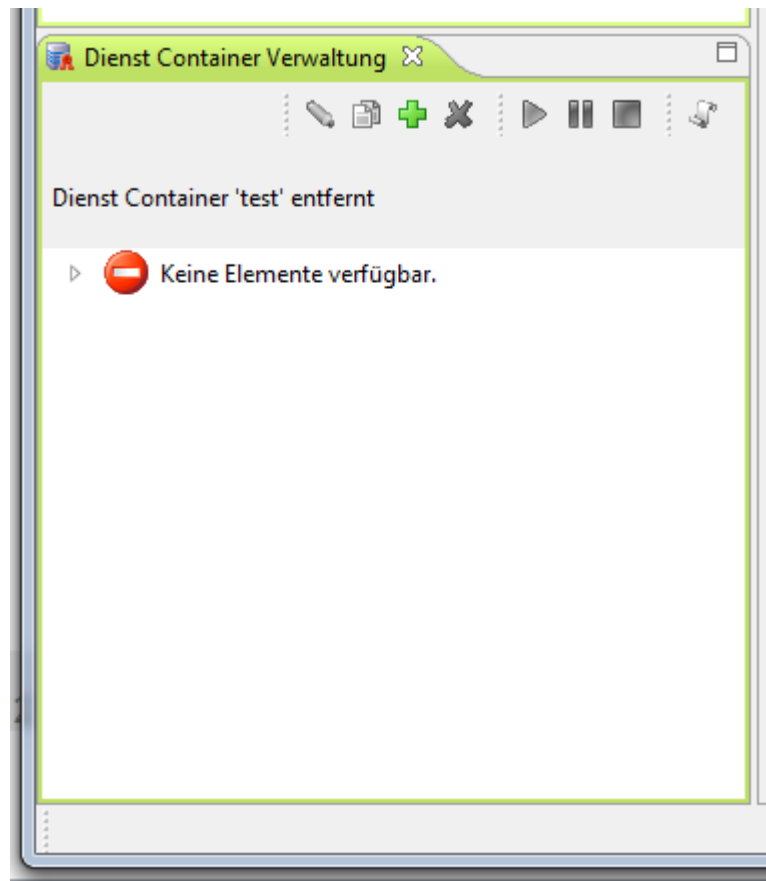
### 6.2 Basic concepts

The basic idea is the decoupling of the “trigger” and the “action”. A trigger is for example a HTTP request or the occurrence of a file in a directory. This trigger, along with dynamic and static argument values is mapped to an action executed by a service, where the service is for example a document signature process or a JavaScript execution.

The definition of the listener that receives a trigger and the service that performs the action is the heart of the service framework.

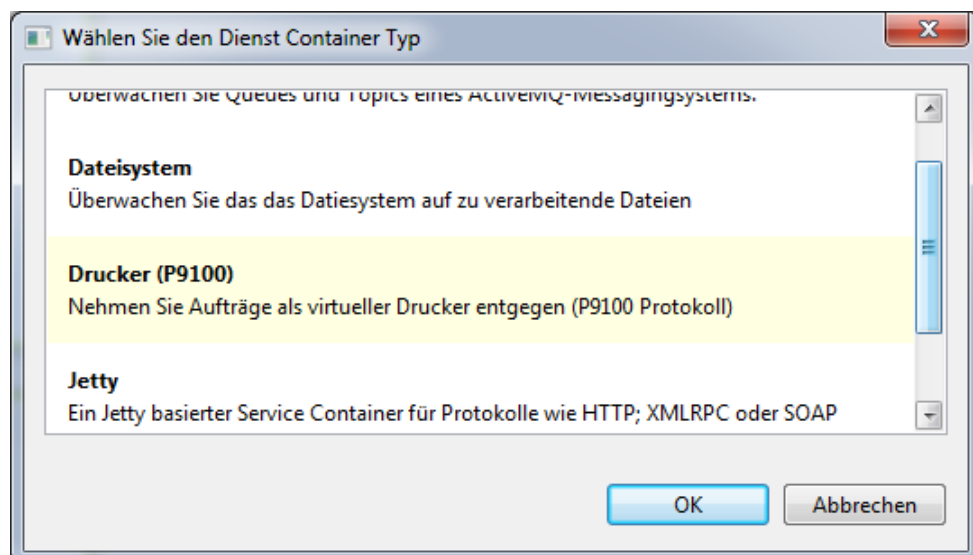
### 6.3 Quick walkthrough

In the application you can find the service container user interface in the menu “Extras -> Services -> Service container console”. This will launch a console in the application sidebar where you can manage you service containers.



Here you can edit, copy, add and remove new containers. If you have selected a specific container you can start, pause or stop it.

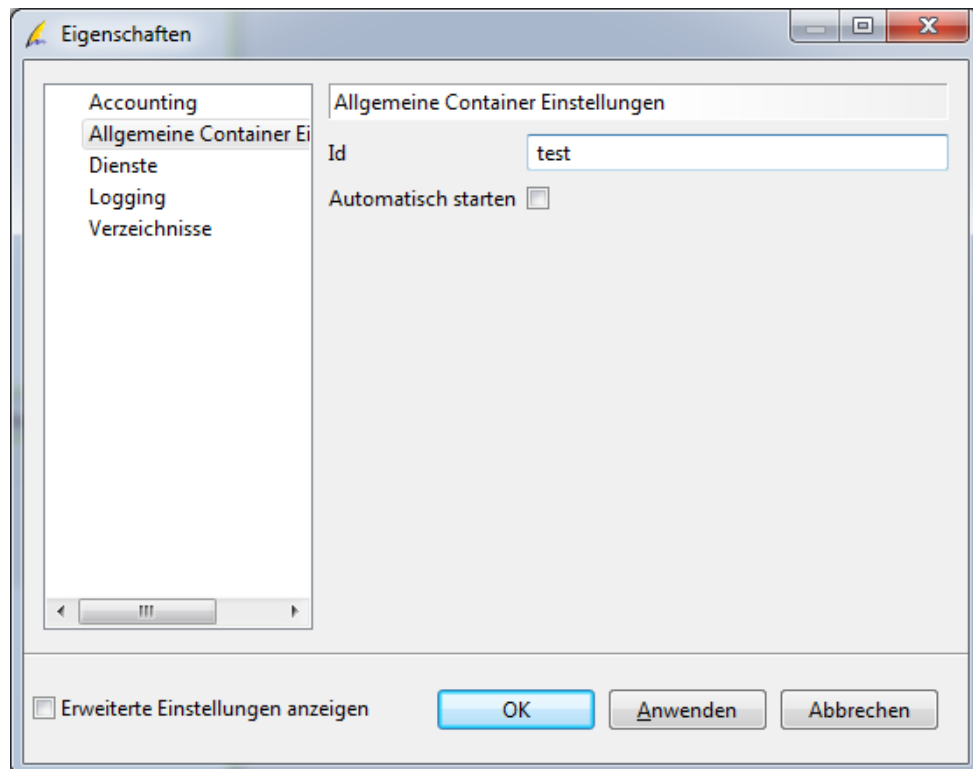
So, let's press the "Add" icon to create a new container.



Here you can select one of the available service containers. Each one will be explained in a later chapter, so don't care now and select "File System".

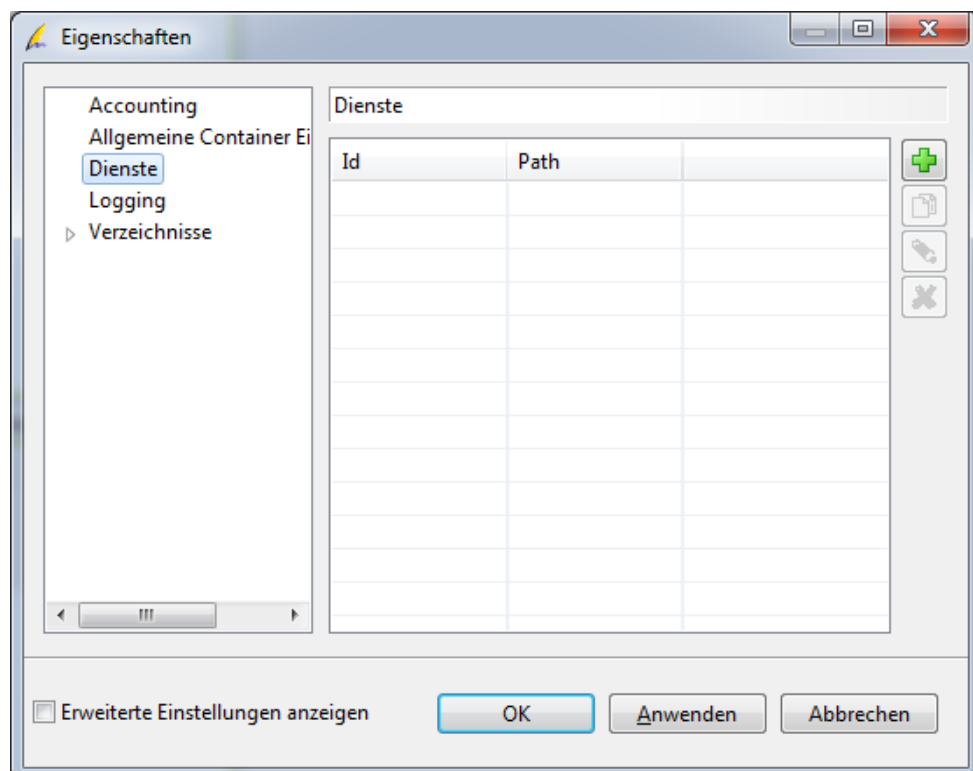
After this the properties page for your new service container shows up.





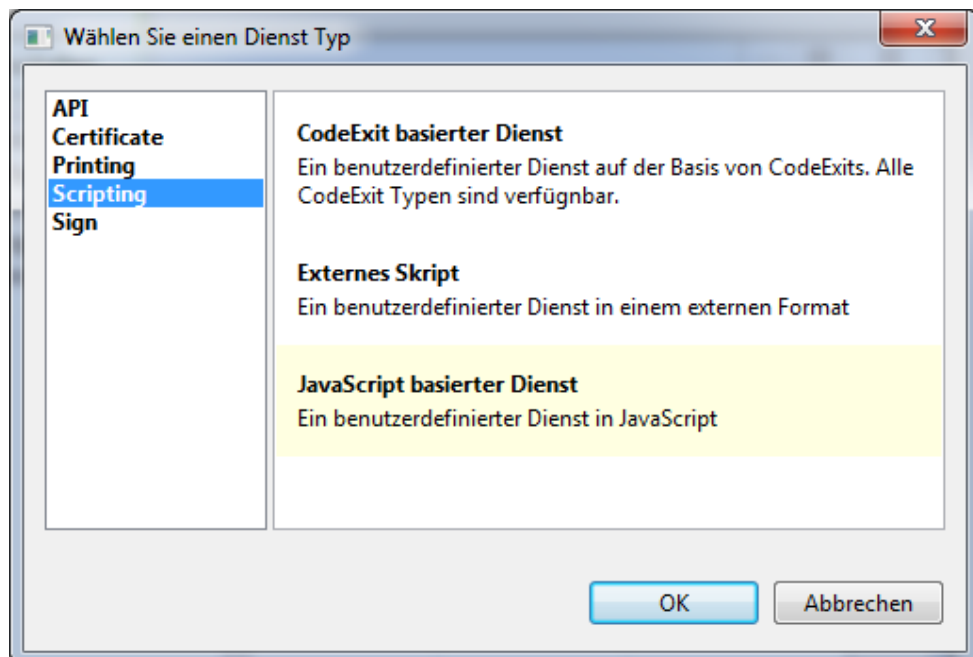
You can enter a lot of information here and some of the pages are different for the service container types. They are explained in the chapters to come.

Simply type “test” for the id and let’s go on. The next step would be to associate a service with this container. Select the “Service” page on the left hand side.



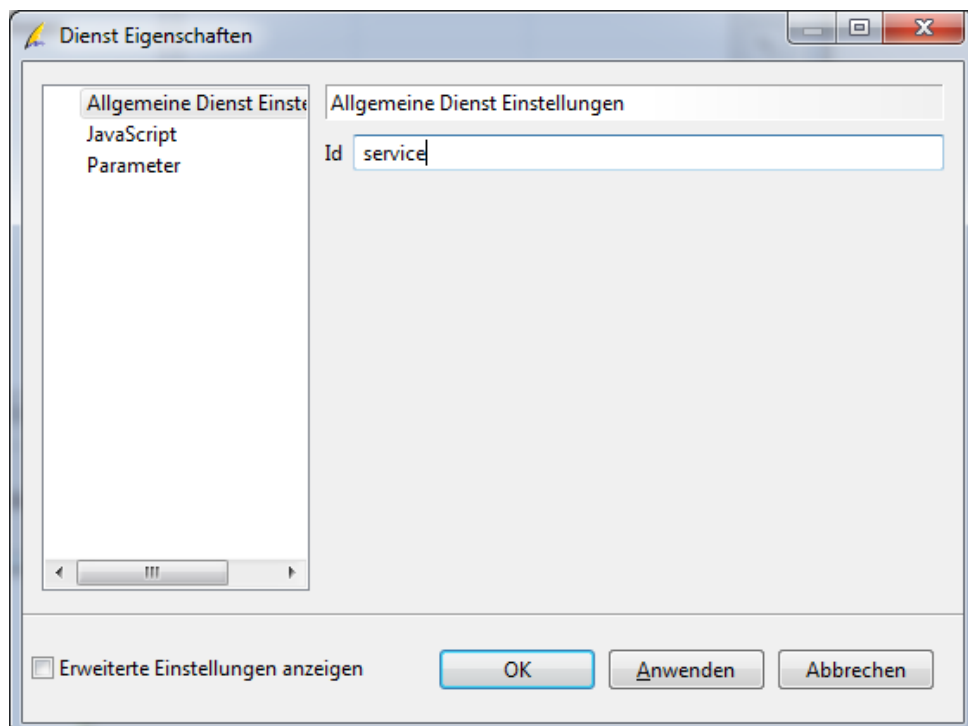
Here you define the services that will handle the triggers from the container. Again, you can add, copy, edit and remove.

Simply “add” a service now.



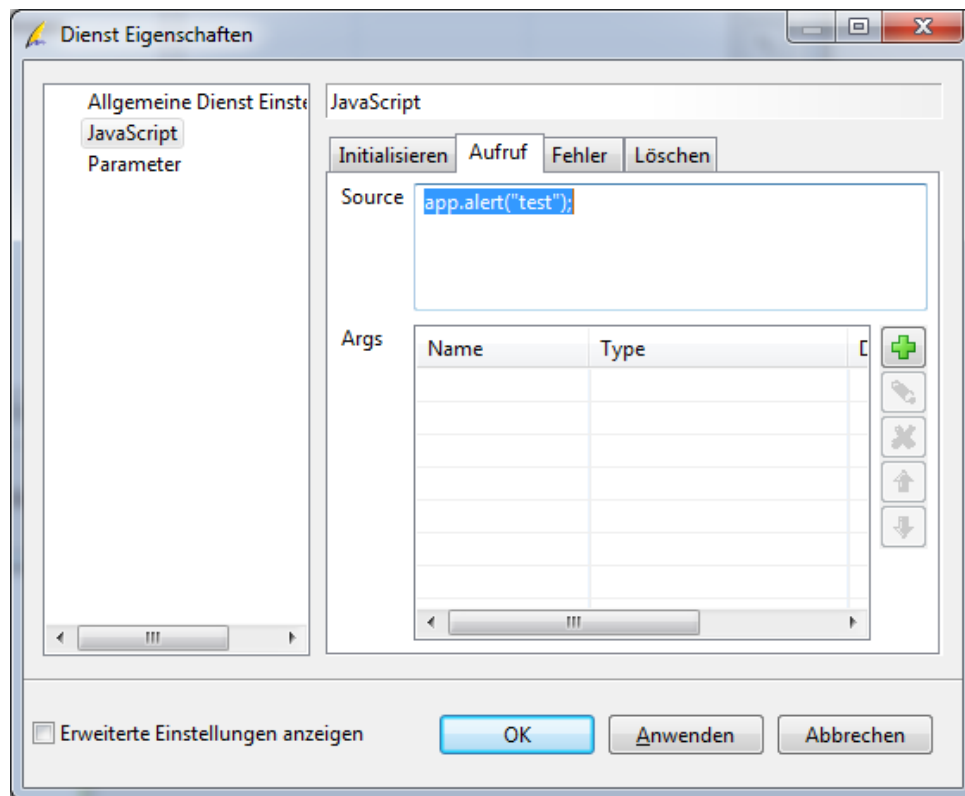
Here you can select from a variety of predefined and scriptable services. For our walkthrough, select the “JavaScript” entry and go on.

The final step now allows the definition of the service itself.



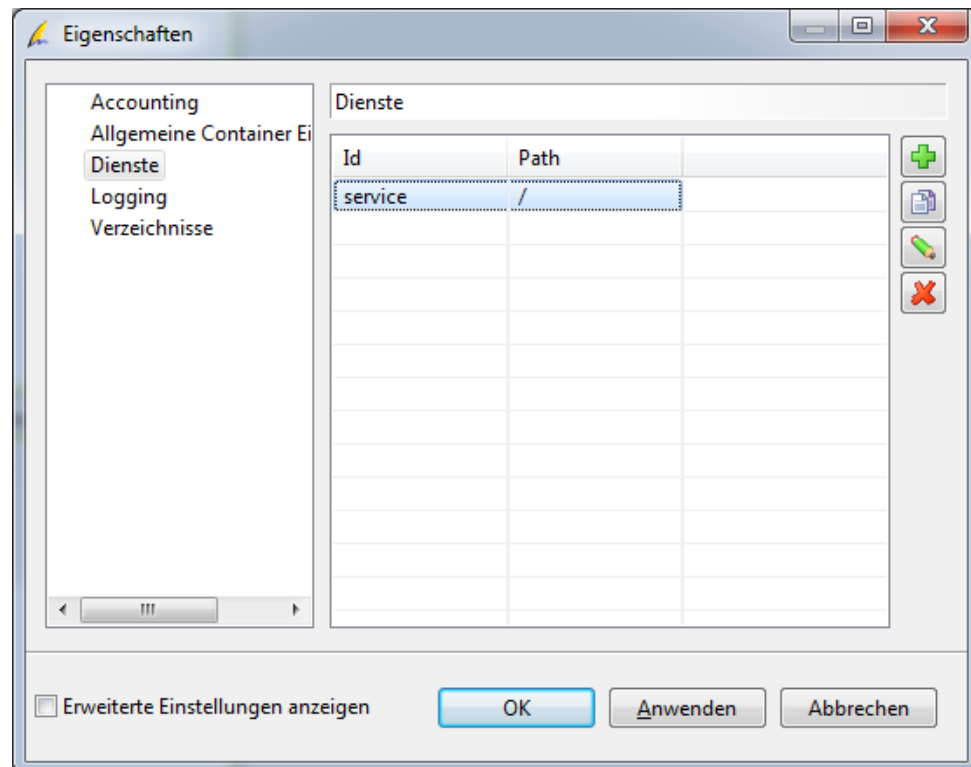
Again, dependent on the service you selected you may see more pages and properties to define. We simply type “service” for the id and are eager to write a small script... so select “JavaScript” on the left hand.

You will see some tabs that allow you to enter JavaScript code and define arguments.



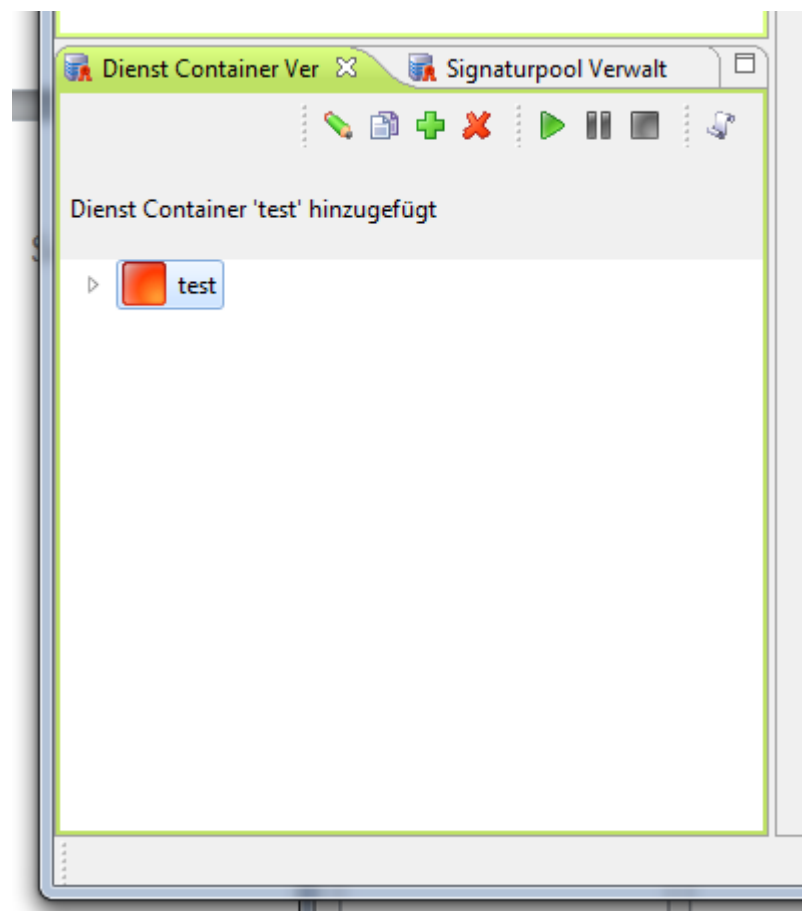
For now, simply select the “Invoke” tab and type “`app.alert("test");`”. Say “OK”.

You return to the service container dialog where the new service now shows up.



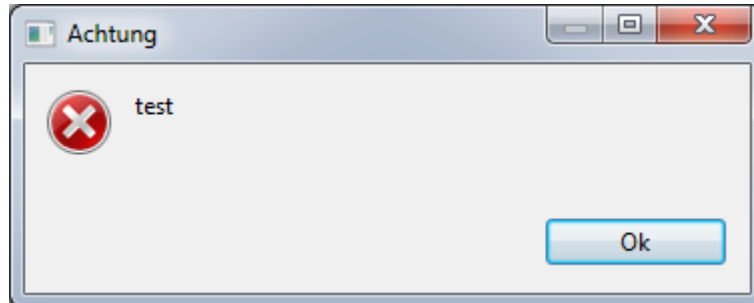
Again, say "OK".

Now, in the service container console the newly defined service shows up.



If you start the service with the green “start” button, the file system monitor starts to monitor the directory “<user>/.<application>\_<version>/<containerid>/input” within your user profile directory. If you followed the instructions, “containerid” should be “test”.

Throw in a file and shortly after a message box like this will show up.

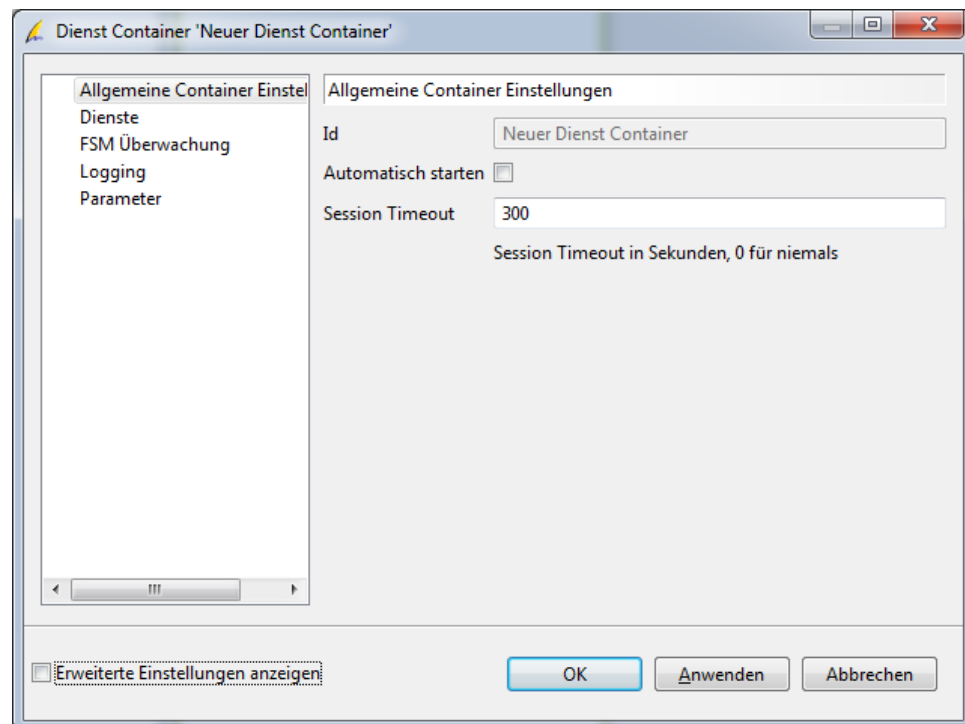


Gratulation, the first service definition using the service container console is done.

## 6.4 Service container

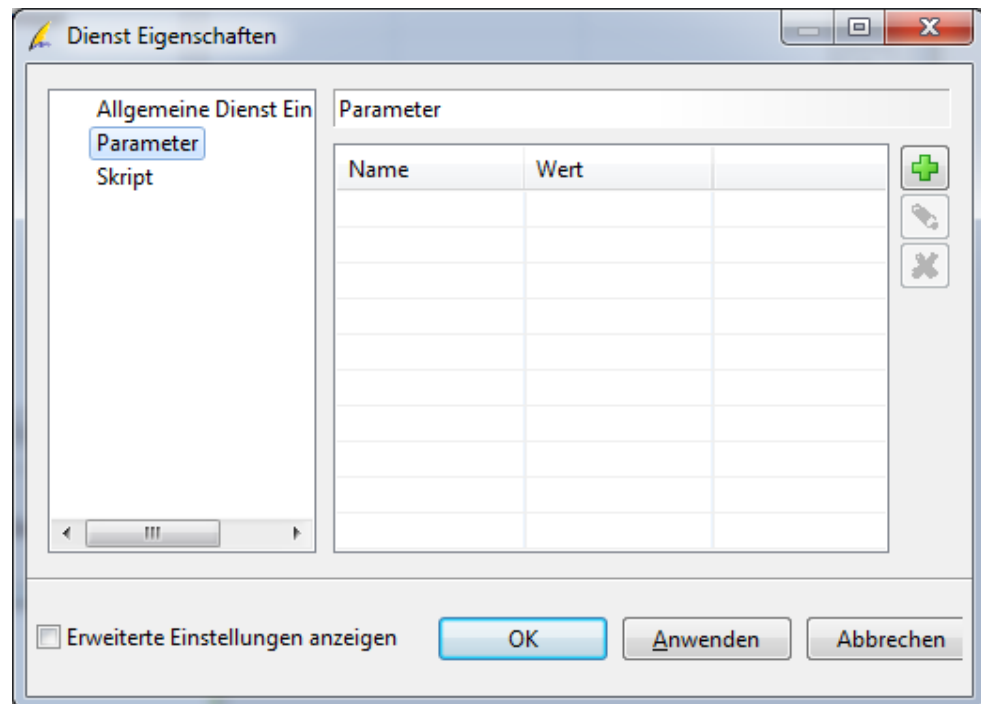
### 6.4.1 Common properties

#### 6.4.1.1 Common container properties



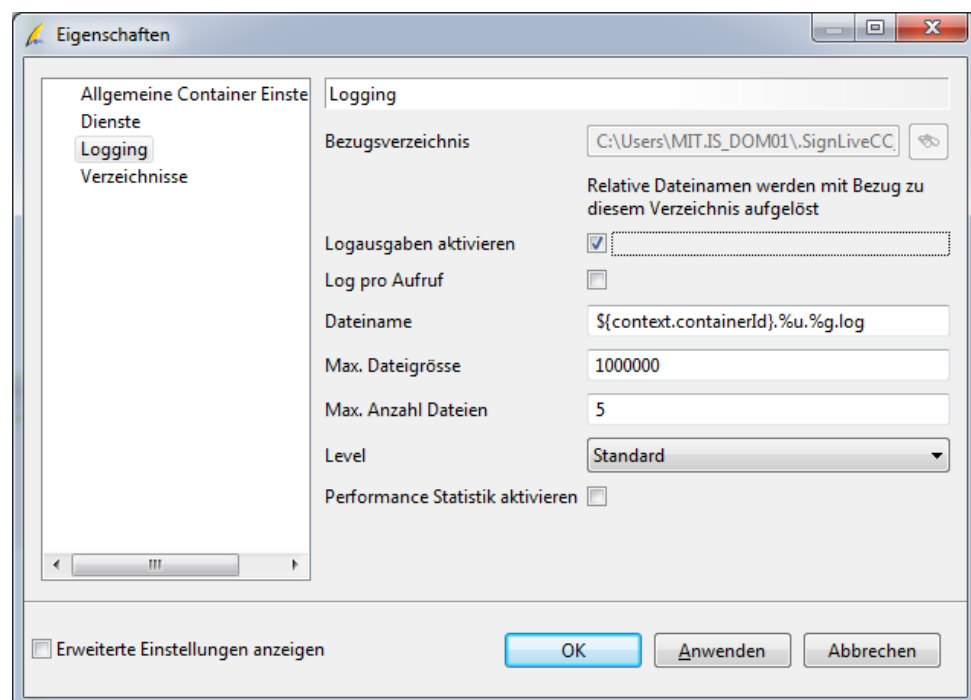
#### 6.4.1.2 Parameter

The “Parameter” page allows you to define optional parameters to the service container itself. Parameters can be accessed programmatically or using string expansion.



#### 6.4.1.3 Logging

Set up logging here.

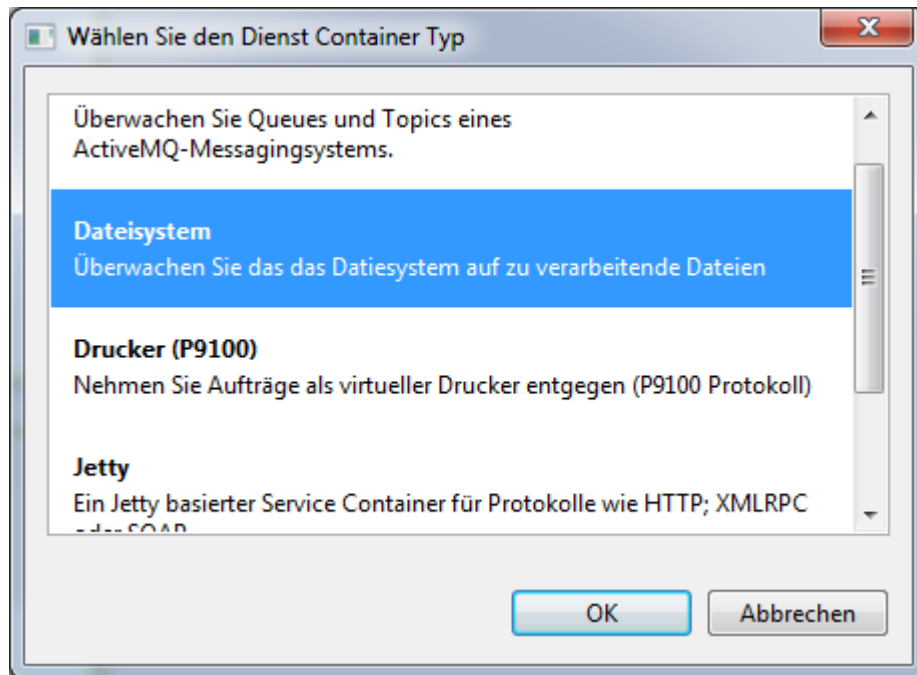


#### 6.4.2 File system container

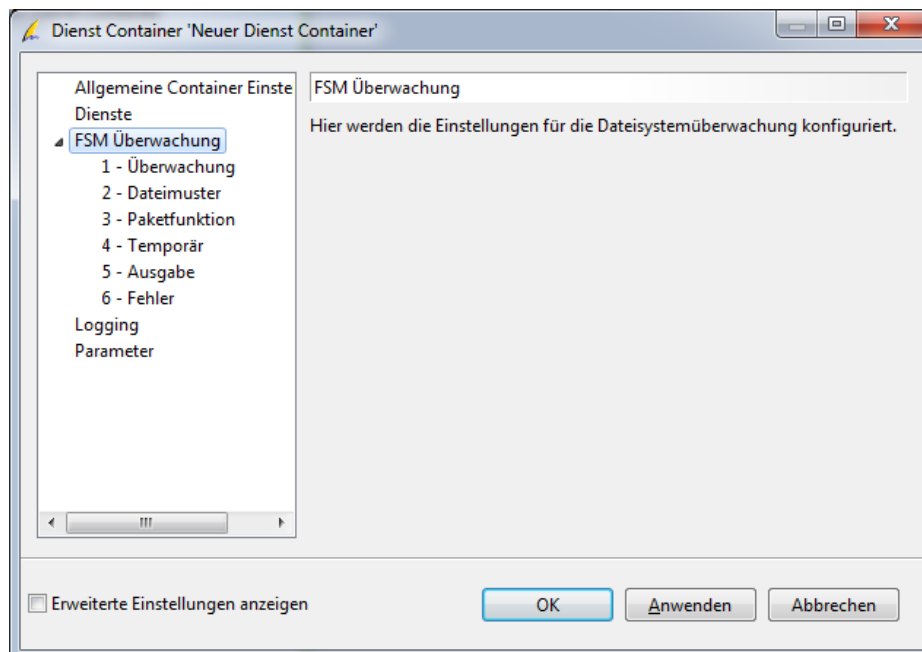
The file system container is a powerful tool for loose system coupling. It monitors one or more directories for the occurrence of a file – with this file an action is triggered.

The files are moved automatically in this process from input to output or error directories.

To create a file system container select “File System” in the selection dialog.



The service container properties dialog shows up. The pages specific for file monitoring are visible under the node “Directories”:



You can configure:

- Monitor This is the directory where the input is expected
- Pattern Define rules for the files to be selected
- Packet You can use an optional “packet” feature, giving a context to multiple single files.



- **Temporary** The input can be moved optionally to a temporary directory
- **Output** This is the place where the output will be written
- **Error** In case of errors while processing, output will be redirected here.

In the following tables all parameters to the monitor are explained

---

#### MONITOR PAGE

Property	Content
Monitor Directory	<p>The folder to be observed.</p> <p>Default <code>\${environment.profiledir}/\${container.id}/input</code></p>
Scan Subdirectories	<p><i>true</i> if sub directories of the input directory should be monitored too.</p> <p>Default <i>true</i>.</p>
Delete Subdirectories after processing	<p><i>true</i> if empty sub directories are deleted after processing.</p> <p>Default <i>false</i>.</p>
Delete Immediately	<p>Flag if files should be deleted from the input or temporary directory immediately after processing.</p> <p>This can be used if the process performed on the file is asynchronous and you don't get information about its outcome (for example when opening an editor). Then you can opt for a "lazy" clean up by setting this property to <i>false</i> and set "Delete After" to some value appropriate for your scenario.</p> <p>If the process on the file is synchronous, you can set this property to <i>true</i>, ensuring that the files processed are deleted immediately from the input or temporary directory.</p> <p>Default <i>false</i>.</p>
Monitor Delay	<p>The delay in seconds before the monitor polls again.</p>

---

Test Delay	<p>After detecting a file the listener looks for changes on that file as the writing process may still be active. This is the delay in milliseconds the file monitor waits between two successive tests for the current file size. If the file size or last access timestamp has changed, the file is not yet ready for processing.</p> <p>If the value is <i>0</i>, no check is made.</p> <p>Default is <i>0</i>.</p>
Retry Delay	<p>The service may fail for some reason. If indicated that the service failure is not fatal, the action can be rescheduled after “retry delay” milliseconds</p>

## PATTERN PAGE

Property	Content
Monitor Extensions	<p>A list of specific file extensions separated by “;”, like <i>.pdf;.txt</i>. Only files whose names end <b>exactly</b> with one of these extensions are filtered for the listener.</p> <p>Please keep in mind that a file must be matched by both filters defined by Monitor Extensions and Monitor Patterns.</p>
Monitor Patterns	<p>A list of specific file patterns, like <i>*.pdf;*.txt</i>. This filter allows for “*” and “?” wildcards anywhere in the pattern. Only files that match the given filter are selected.</p> <p>Please keep in mind that a file must be matched by both filters defined by Monitor Extensions and Monitor Patterns.</p>
Attachment Extensions	<p>A list of specific file extensions separated by “;”, like <i>.p7s;.pkcs7;.pkcs#7</i>.</p> <p>The listener processes the files defined by Monitor Extensions and Monitor Patterns, ignoring the files defined by Attachment Extensions. For every accepted input it looks for attachments by adding the specified attachment extension(s) to the input file name resp. to its name without extension.</p> <p>Attachments are available for processing (scripting) and moved along with the input file while processing.</p>

Transparent file handling	<p>If you want to use “transparent pass through”, you must activate this flag.</p> <p>Transparent files are moved by the file system container, but no actions are performed on them. This is useful when we are only a part in a multistep process and the files are targeted at another system.</p> <p>Most times you can achieve similar results using the attachments feature.</p>
Transparent Files Extensions	<p>A list of specific file extensions separated by “;”, like <i>.pdf;.txt</i>. Only files whose names end <b>exactly</b> with one of these extensions are filtered for the listener.</p> <p>Please keep in mind that a file must be matched by both filters defined by ~Extensions and ~Patterns.</p>
Transparent Files Patterns	<p>A list of specific file patterns, like <i>*.pdf;*.txt</i>. This filter allows for “*” and “?” wildcards anywhere in the pattern. Only files that match the given filter are selected.</p> <p>Please keep in mind that a file must be matched by both filters defined by ~Extensions and ~Patterns.</p>

## PACKET PAGE

Property	Content
Marker File	If a file is specified here, processing will not start before this file shows up. This may improve synchronization for processing multiple files (attachments).
Log File	If present, all log entries with regard to this packet are appended to this file. The log file is moved along with the packet itself.
Global error handling	If this flag is set, errors when handling a single file will invalidate the complete packet. This means, either all files will make in to the output or all will end in the error directory.

## TEMPORARY PAGE

Property	Content
Temp Directory	<p>If defined, a file found in the input is moved to this temporary directory before processing. This is especially useful if the processing is asynchronous and the files must be moved to keep from re-processing multiple times.</p> <p>In most cases you will not need to define a temp directory, processing files directly from the input.</p> <p>The default is <i>empty</i>.</p>
Delete After	<p>The delay in <b>hours</b> after which the files in the temporary directory are deleted. This is useful only in combination with Delete Immediately=<i>false</i>.</p>

## OUTPUT PAGE

Property	Content
Output Directory	<p>If this property is defined, the input file is moved to this directory after processing. The filename used can be configured with Output Filename.</p> <p>Default <code>\${environment.profiledir}/\${container.id}/output</code></p>
Output Filename	<p>If Output Directory is defined, the input file is moved. The filename can be configured using this property.</p> <p>Default  <code>\${event.params.RELATIVEPATH}\${event.params.INPUTFILENAME}</code> </p>
Output Filename Collision	<p>If the file defined with Output Filename causes a collision, the monitor falls back to this definition.</p> <p>Default  <code>\${event.params.RELATIVEPATH}\${event.params.INPUTFILEBASENAME}.\${system.time:d}.\${event.params.INPUTFILEEXTENSION}</code> </p>
Discard Input	<p>If this flag is set, the input files are simply discarded, not moved to output.</p>

---

Keep File Date	If true, the file timestamp in the output is reset to the file timestamp of the input file.
----------------	---

---



---

**ERROR PAGE**


---

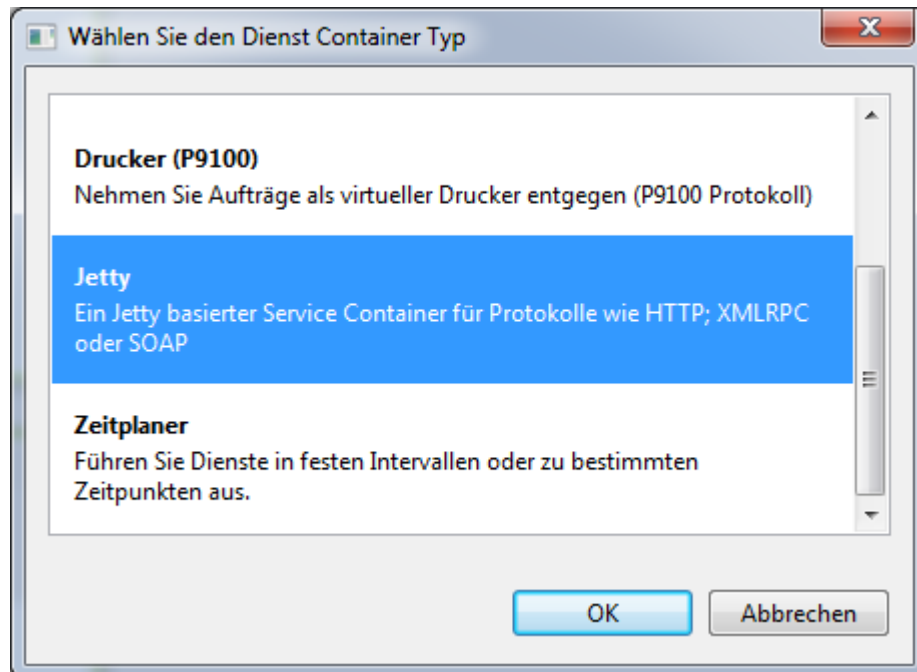
Property	Content
Error Directory	<p>If this property is defined, the complete input is moved to this directory if the processing encounters an error. The path structure from the input directories are preserved.</p> <p>Default <code>\${environment.profiledir}/\${container.id}/error</code></p>
Error Filename	<p>If Error Directory is defined, the input is moved in the case of an error. The filename can be configured using this property.</p> <p>Default  <code>\${event.params.RELATIVEPATH}\${event.params.INPUTFILENAME}</code></p>
Error Filename Collision	<p>If the file defined with Error Filename causes a collision, the monitor falls back to this definition.</p> <p>Default  <code>\${event.params.RELATIVEPATH}\${event.params.INPUTFILEBASENAME}.\${system.time:d}.\${event.params.INPUTFILEEXTENSION}</code></p>

---

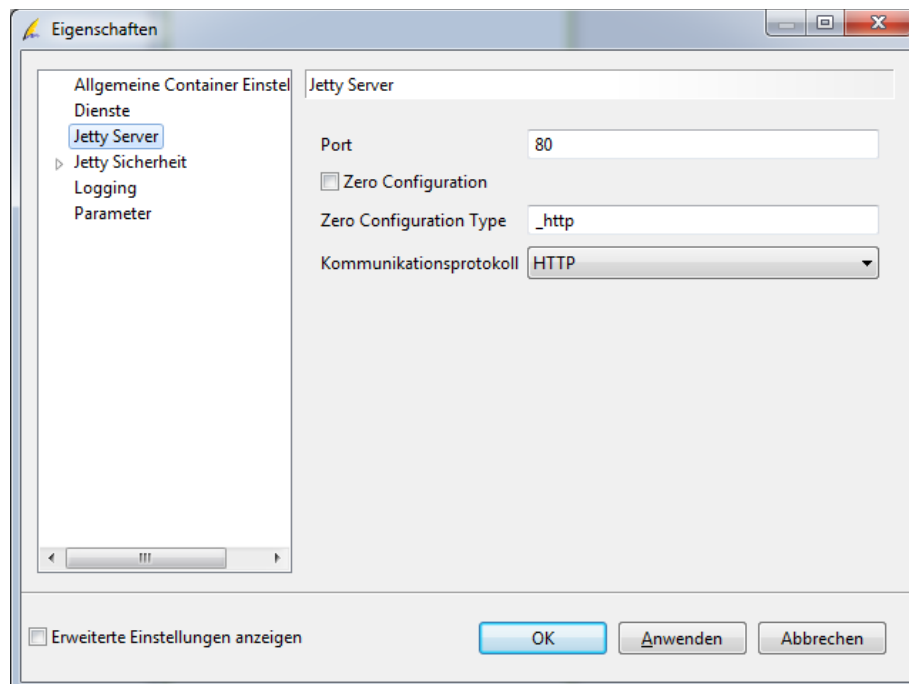
### 6.4.3 Jetty Container

Using the Jetty container, the application can interact with clients using HTTP based protocols. It can serve as an XMLRPC listener, for SOAP or just plain HTTP requests.

To create a Jetty container, select “Jetty”



Jetty defines several pages in the service property dialog to fine tune the behavior:



- Jetty Server Jetty HTTP server specific settings

- Jetty Security Common Common Security Settings
- Jetty Security Authentication Setup security realms
- Jetty Security SSL Setup the SSL environment to be used by Jetty

**SERVER PAGE**

Property	Content
Port	The port to listen to. Default is <i>80</i> .
Zero Configuration	This service container can publish its service using “Zeroconf” when setting this flag to true.  The service is published at “<type>_tcp.local.”
Zero configuration type	The “type” prefix used for publishing the service (see above) Default <i>_http</i>
Protocol	One of the supported protocols. You can find more on this in the respective chapters below. <ul style="list-style-type: none"> <li>• HTTP</li> <li>• XMLRPC</li> <li>• SOAP</li> </ul>

**SECURITY – COMMON PAGE**

Property	Content
Only local connections	If activated Jetty accepts connections only from local loop (localhost, 127.0.0.1)

**SECURITY – AUTHENTICATION PAGE**

Property	Content
----------	---------

---

**Client Authentication** Here you can activate Jetty Basic Authentication.

In the list user/password combinations are added.  
 Passwords are stored using the Jetty encryption facilities.

---



---

#### SECURITY – SSL PAGE

Property	Content
Use SSL	Activates SSL for this container
Keystore file	The file where the private key for the server is stored. This is a standard Java keystore file.
Keystore password	The password for the keystore file
Key password	The password for the private key in the file
Client authentication	Flag if client authentication is required
Truststore file	The file for trusted client identities
Truststore password	Optional password for the truststore file.

---

##### 6.4.3.1 Protocol HTTP Plain (Post)

Accepts “plain” HTTP Post requests and forwards to the service implementation.

The servlet acting as the gateway to your service (action) can handle a great variety of different marshaling scenarios.

##### 6.4.3.2 Protocol XMLRPC Protocol

Accepts XMLRPC encoded post requests.

##### 6.4.3.3 Protocol SOAP Protocol

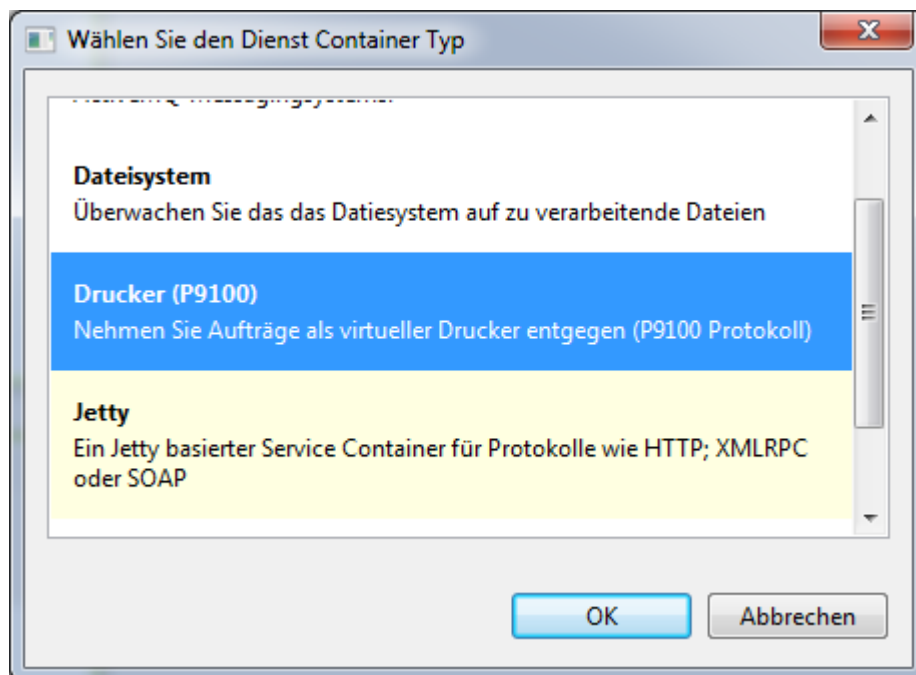
The web service container integration is a Java WS / SOAP compliant server with pluggable “service” configurations. The primary use for this container is the easy and fast integration in existing SOA engines or BPM scenarios.



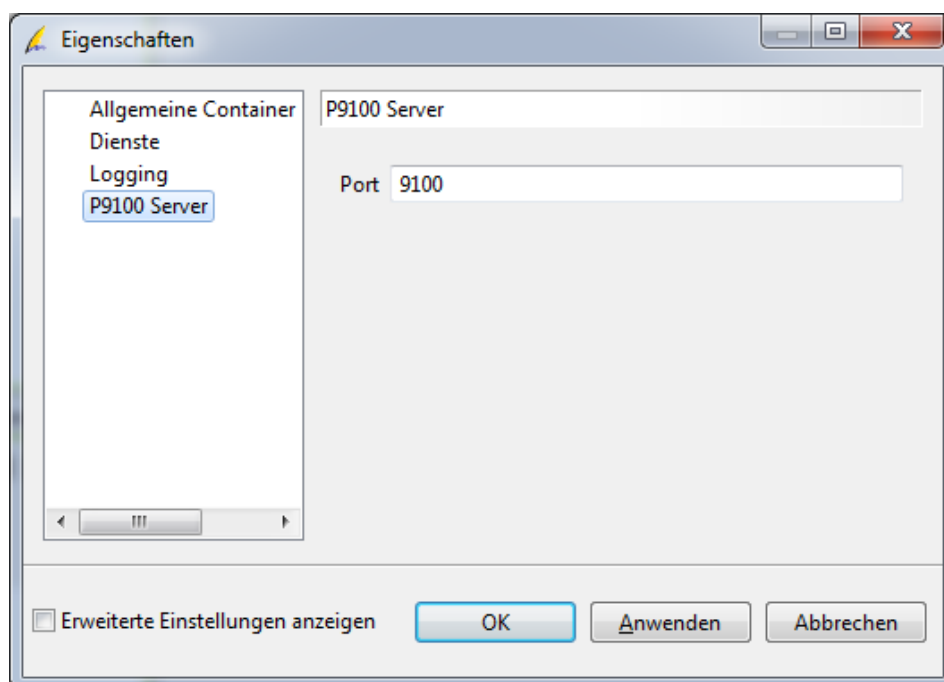
This feature is based on the popular Apache CXF web service framework. Not every feature of CXF is supported and available at this level. You can revert to plain Java WS or CXF if you need to.

#### 6.4.4 Virtual printer container

This is a very simple yet powerful coupling mechanism for processing documents from systems that produce print output. Install a virtual printer and accept the document directly in its printed form.



The properties configuration dialog for the P9100 virtual printer.



#### P9100 SERVER PAGE

Property	Content
----------	---------

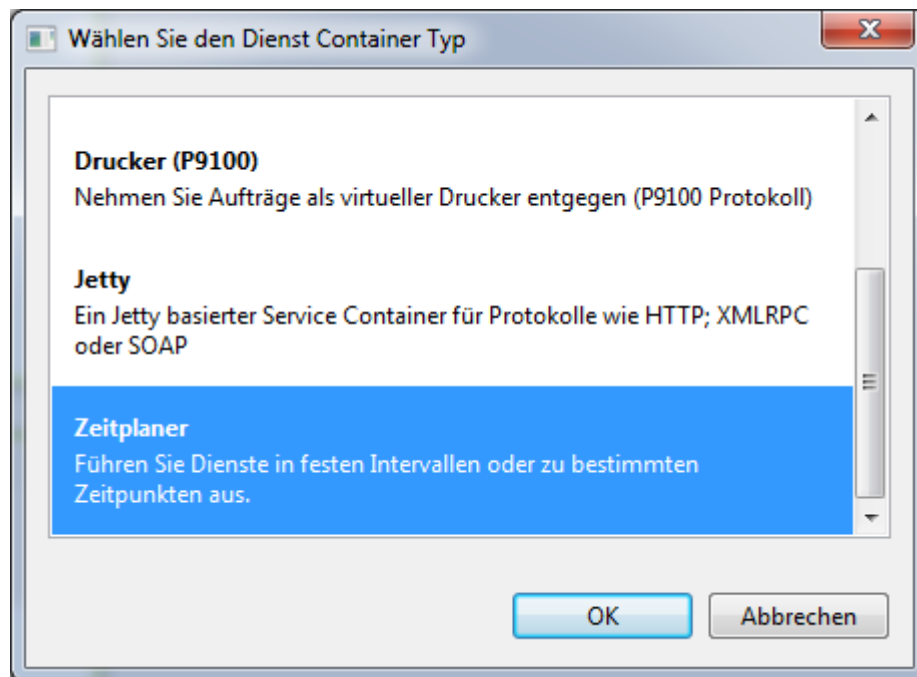
---

Port	The port to listen to
------	-----------------------

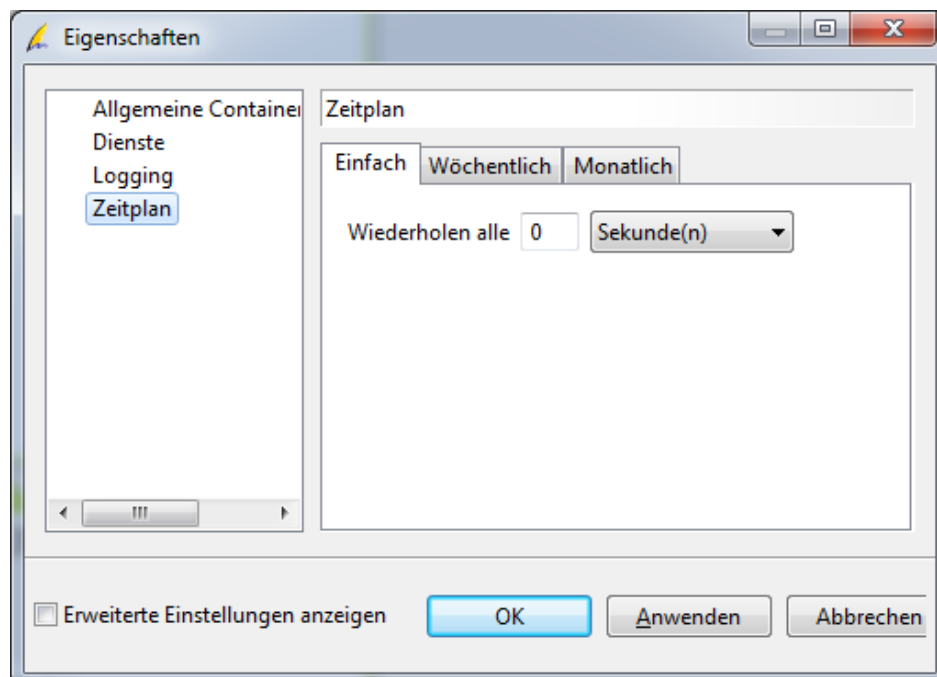
---

#### 6.4.5 Scheduler container

Here you can add scheduled services (like a cron job).



The service container configuration dialog



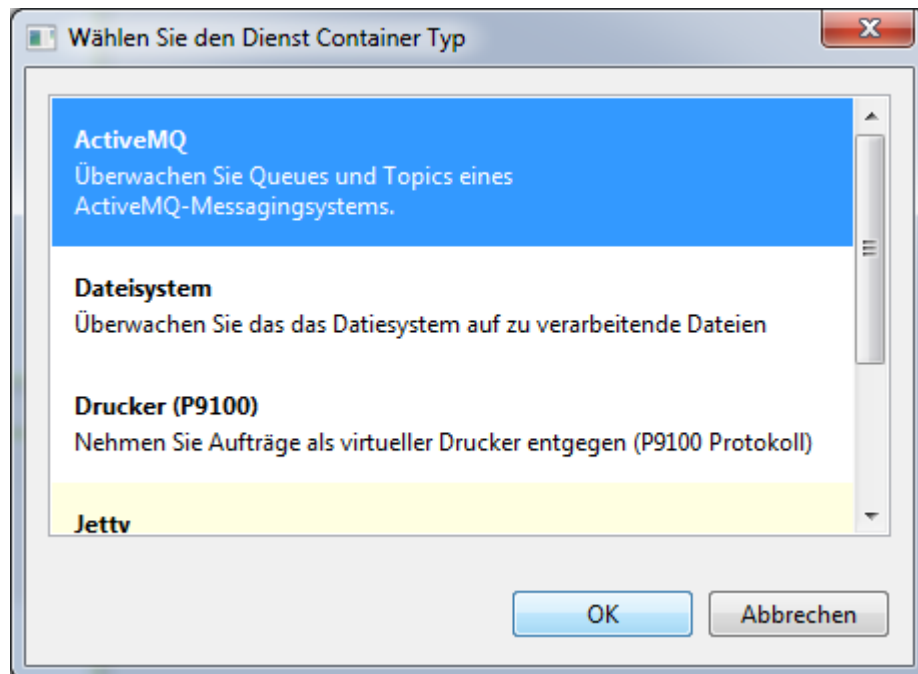
The implementation allows you to add

- Simple Repeat every “delay” seconds

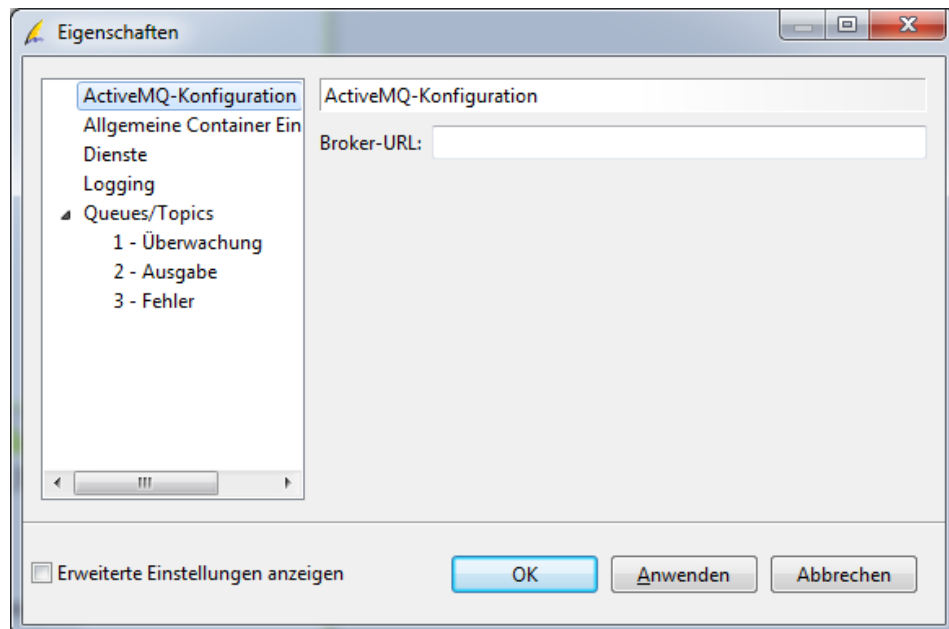
- Weekly Repeat on a weekday schedule
- Monthly Repeat on a monthly schedule

#### 6.4.6 Message queue (JMS) container

The service container can attach to a message queue.



The service configuration properties dialog:



You can set up the monitor itself and the queues used for communication.

## 6.5 Services

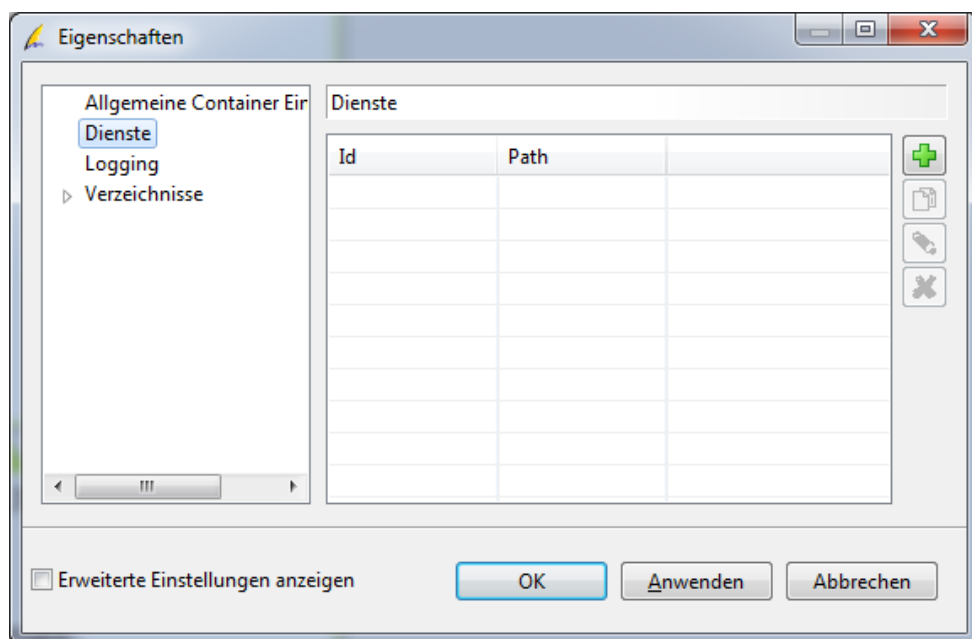
### 6.5.1 Overview

You can plug any service into the service container if compatible to the “de.intarsys.service.api.IService” interface of the framework.

To your convenience, many service implementations are preconfigured – some of them only need to be plugged in, some have sophisticated settings, some of them allow free scripting.

Here we give a short overview on the most important features of services.

Services are added to the service container on the “Services” page of the service container properties dialog.



Press “add” to install a new service.

### 6.5.2 Service Call

The service framework translates the “trigger” to an action, resulting in a service invocation. This invocation has a number of default arguments provided by the framework.

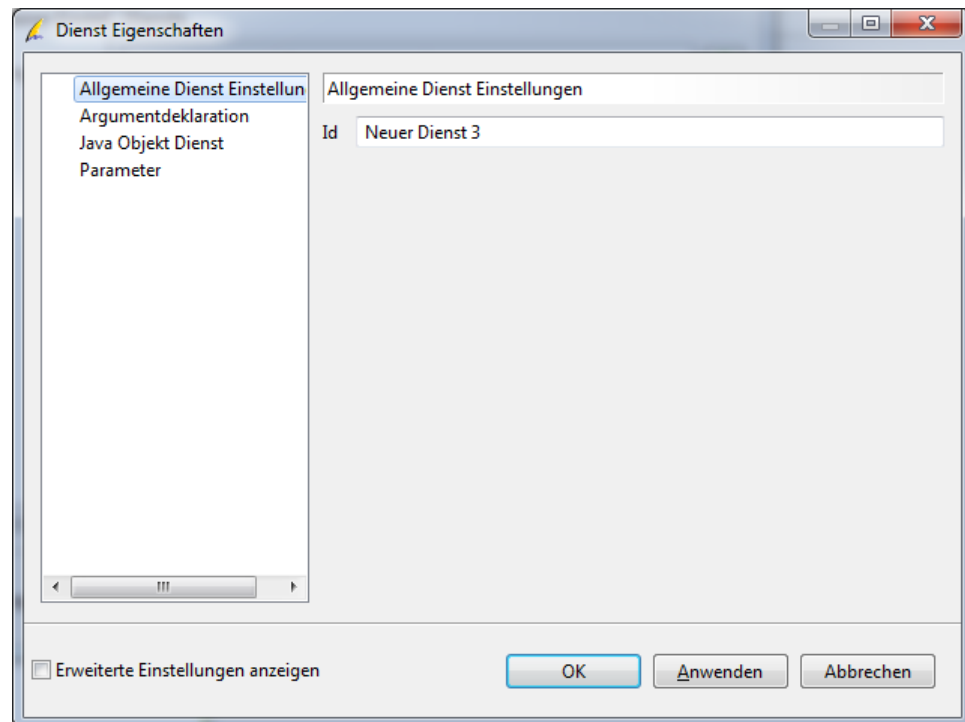
Argument	Meaning
_args	The service arguments structure.
_name	The name of the service that is called. The value provided here may depend on the service event source.

<argument 1..n>	An argument for every argument supplied by the caller

### 6.5.3 Standard Properties

#### 6.5.3.1 Common

Every service has a set of standard properties. First, you have the “Common service settings” page.

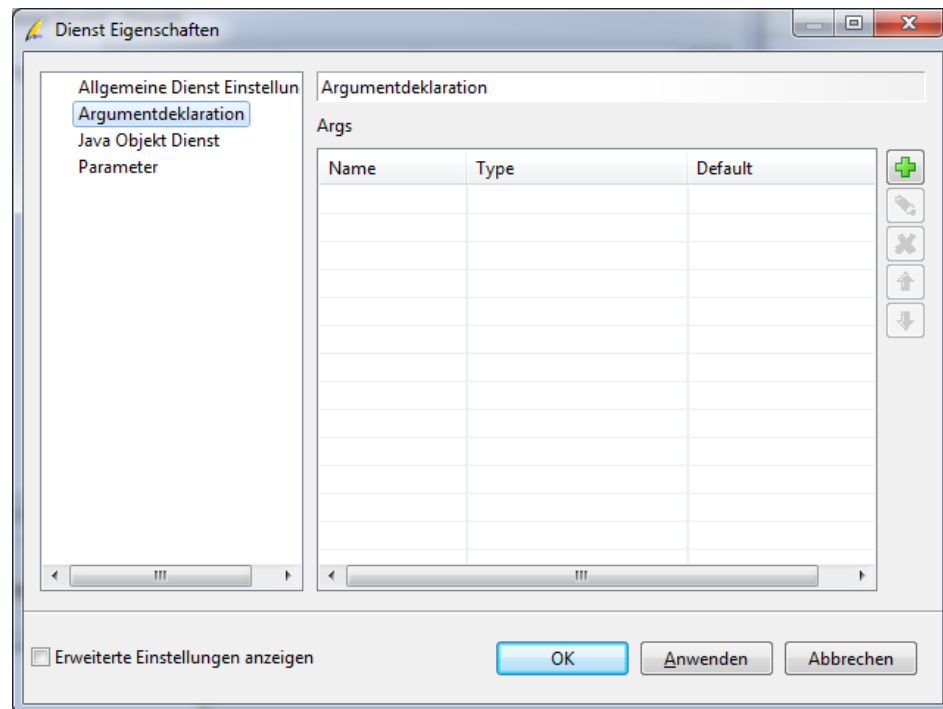


Here you can add a unique id for the service. Some implementations may use the id for defaulting internal settings, like directory names or zero configuration settings. The id is available in string expansion as “config.id”.

You cannot change the id of a service once it is defined.

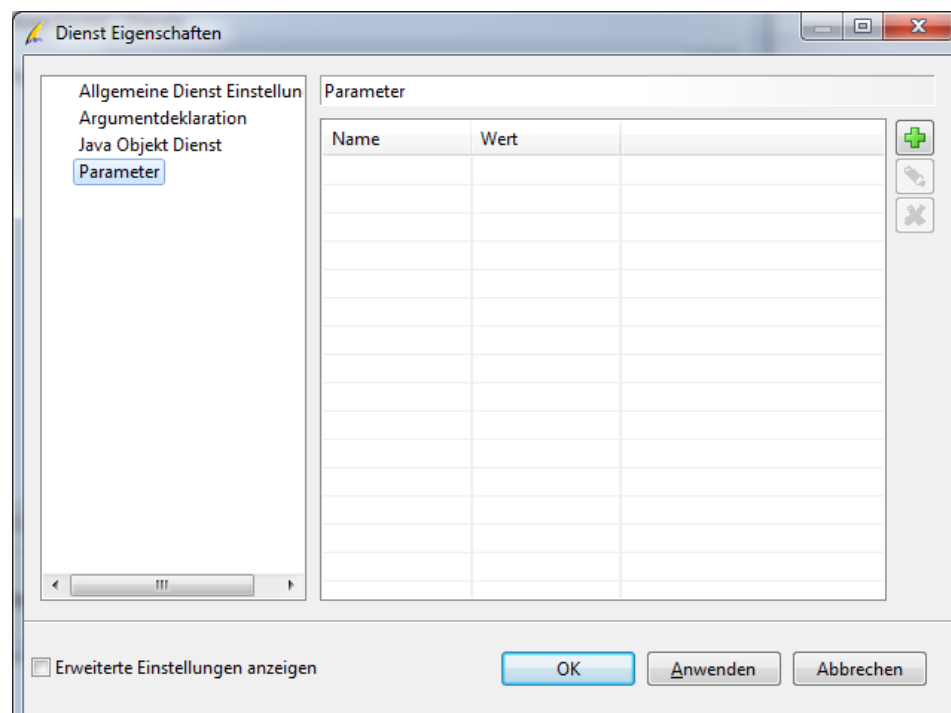
#### 6.5.3.2 Arguments

The “Arguments” page allows you to declare arguments and defaults to the service. These are “merged” with the concrete arguments upon “invoke”.



### 6.5.3.3 Parameter

The “Parameter” page allows you to define optional parameters to the service. It is up to the service to interpret these settings. Parameters can be accessed programmatically or using string expansion with the expression “`config.param.<name>`”.

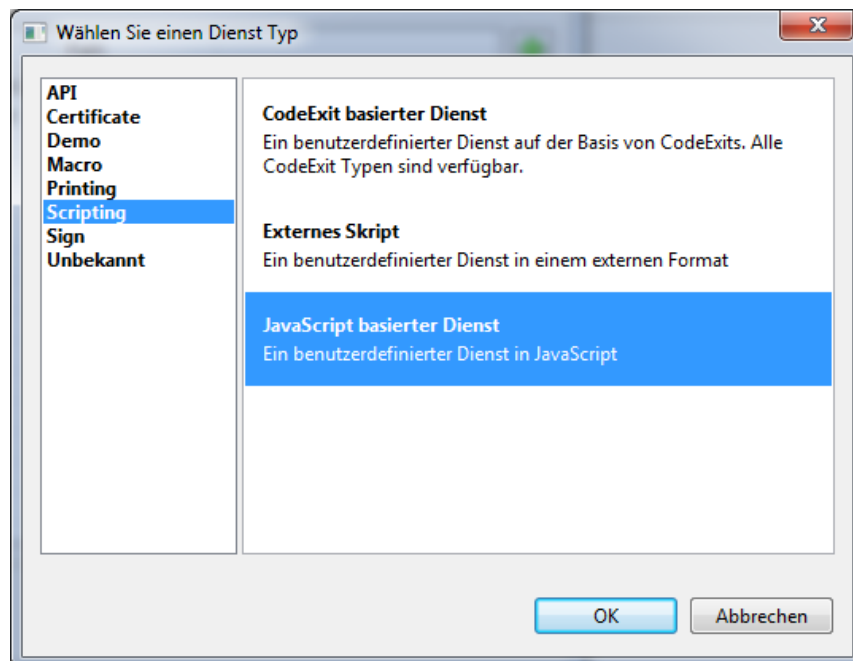


### 6.5.4 JavaScript

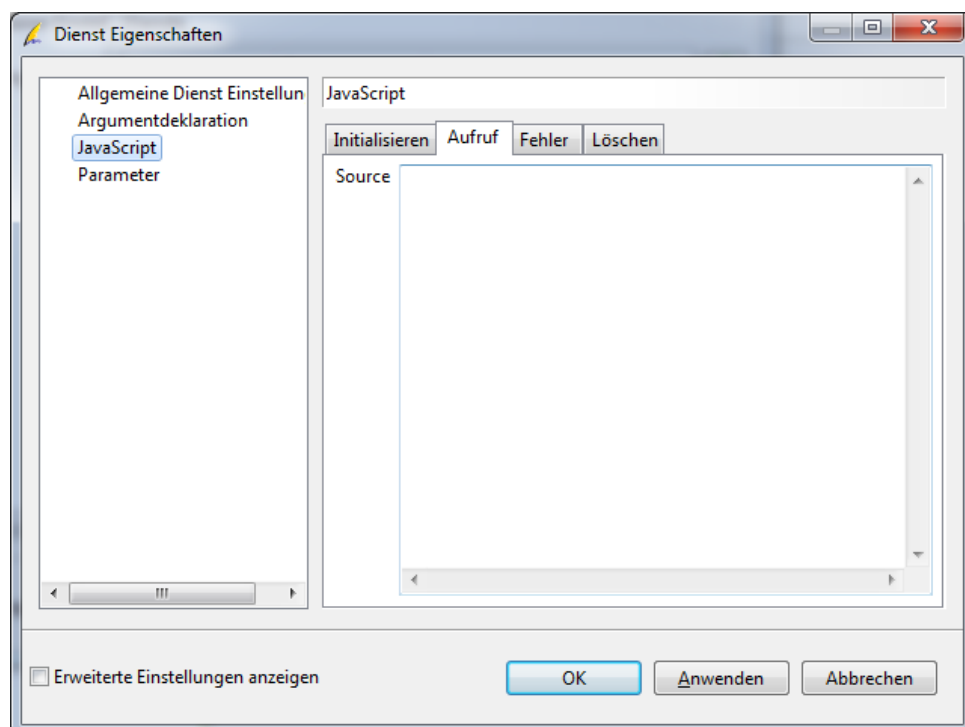
Java scripts use directly the JavaScript binding of the application.



JavaScript services can be created in the service type selection dialog by selecting the category “scripting” and “JavaScript”.



You will see the service properties dialog.



Besides the standard service configuration pages, the page “JavaScript” allows you to define literal JavaScripts for the service lifecycle

- Init
- Invoke
- Error

- Destroy

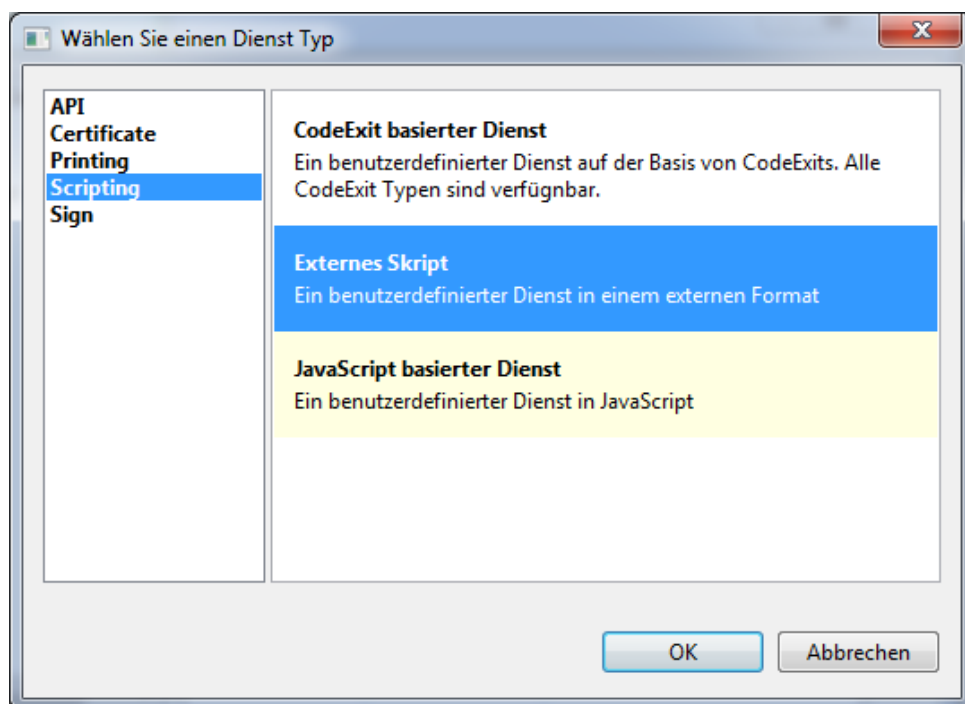
Examples for a JavaScript

```
app.alert('hello');
```

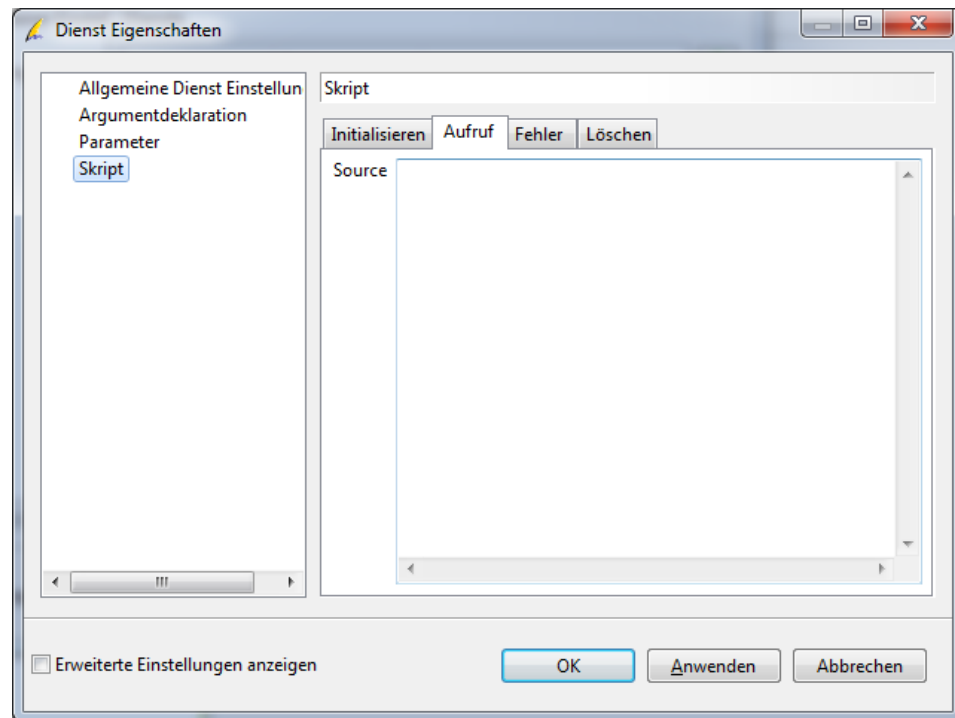
### 6.5.5 External script

External scripts use the applications scripting framework to call in user defined scripts, stored in the file system. By default JavaScript is supported in the application. Other script languages may be available.

External scripts can be created in the service type selection dialog by selecting the category “scripting” and the “External Script”.



You will see the service properties dialog.



Besides the standard service configuration pages, the page “Script” allows you to define scripted actions for the service lifecycle

- Init
- Invoke
- Error
- Destroy

The script “source” field references a script in the scripting framework. If the script name is relative, it is looked up in

- 1) The “basedir” of the service (this is the directory where the instrument was defined).
- 2) All search paths for the scripting framework.

The script name may be defined with templates. It is expanded once at service startup.

Examples for script source

```
test
```

Lookup the script “test.js” (the extension depends on the supported script languages) in the instrument directory and all search paths.

```
scripts/test
```

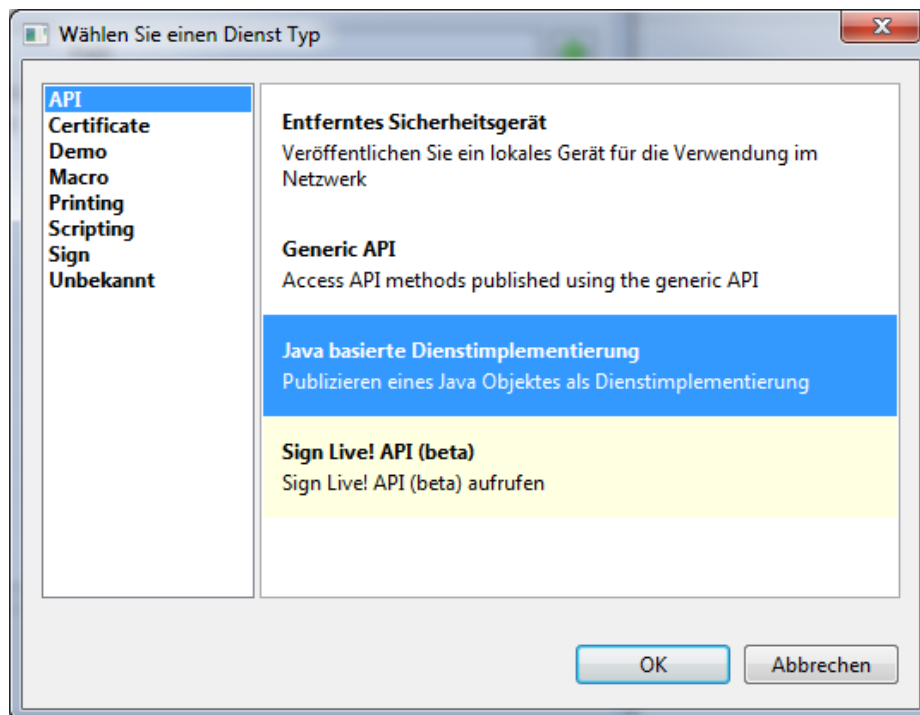
Lookup the script “scripts/test.js” (the extension depends on the supported script languages) in the instrument directory and all search paths.

```
${config.basedir}/test
```

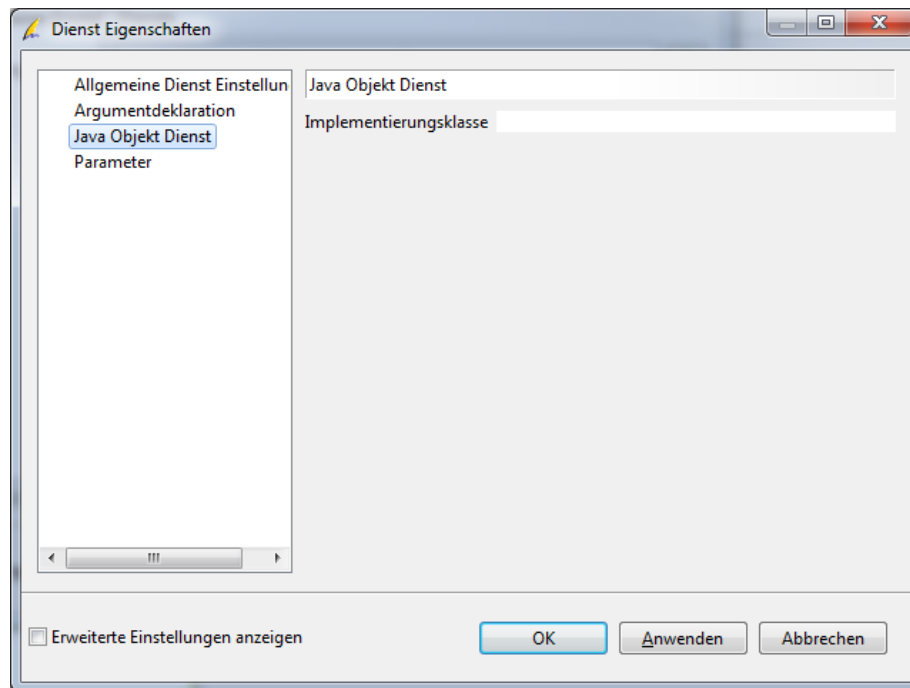
Lookup the script “test.js” (the extension depends on the supported script languages) only in the instrument basedir where the service was defined. This is because the template expands to a fully qualified path name.

### 6.5.6 Java based script

You can directly publish Java Pojo's via the service framework. Select the category “API” and “Java based service”.



You will see the service properties dialog.



Here you define the implementation class for your service.

The object must currently implement the following signature that will get called via reflection.

```
void onInit(IArgs args);  
void onDestroy(IArgs args);  
Object onService(IArgs args);
```

## 6.6 Decorators

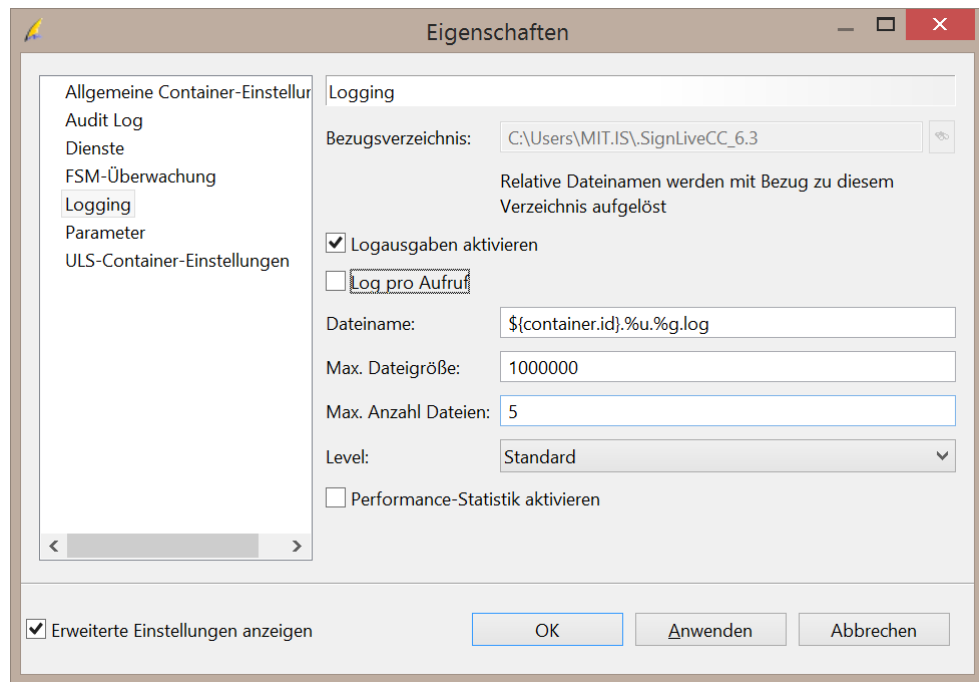
### 6.6.1 Overview

Decorators are features that can be switched on independent from the service, such as logging, authentication or auditing.

The service call is tunneled through the decorator pipeline before the service action is triggered.

To simplify administration, decorators are configured on the service container level, so every service gets decorated using the same decorators.

As an example, you can see the logging decorator settings on the service container definition page:



### 6.6.2 Logging

The logging decorator, if enabled, will add entries in a dedicated log for the service container.

In addition, a variable “logger” is attached to the service context to enable you to log service related information to the same log.

You can create a complete log for the service container or a dedicated log per request.

Last, you can enable a statistics monitor that adds statistic information about requests to the log.

### 6.6.3 Audit Log

#### 6.6.3.1 Overview

**Remark:** Audit logging is not available in every software edition.

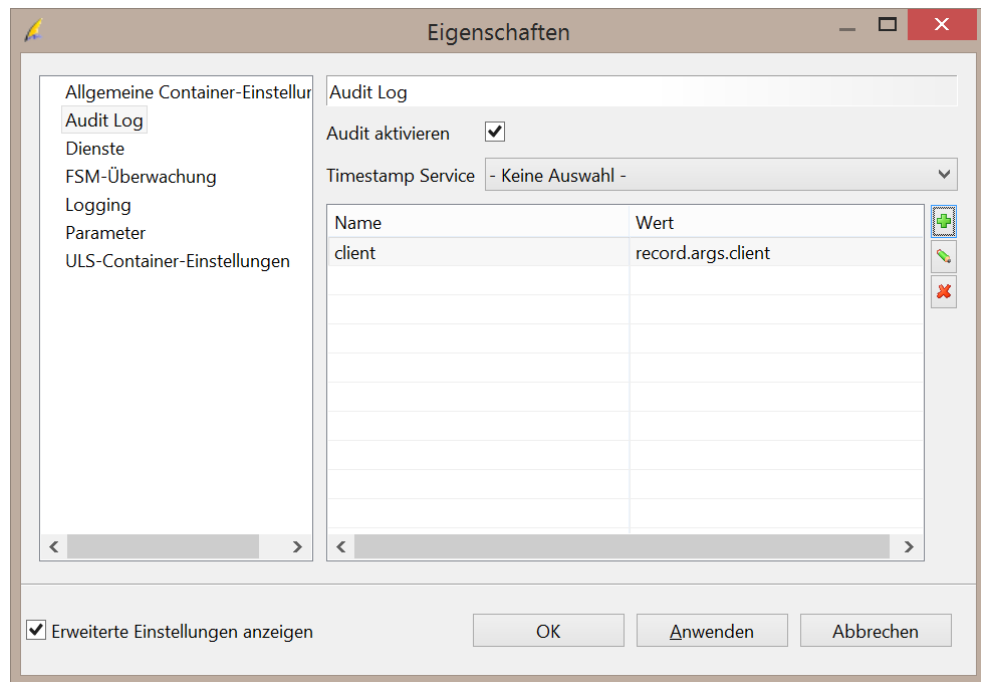
The audit log provides evidence for the service execution. A dedicated log is created that contains configurable information about each service request.

In addition to a standard log, the auditing adds tags to an event by cryptographic means that make it harder (but not impossible) to tamper with the audit data.

The audit log is created in a CSV format in a file named “audit.<containerId>.log”.

#### 6.6.3.2 Configuration

To enable auditing, you must check the respective box.

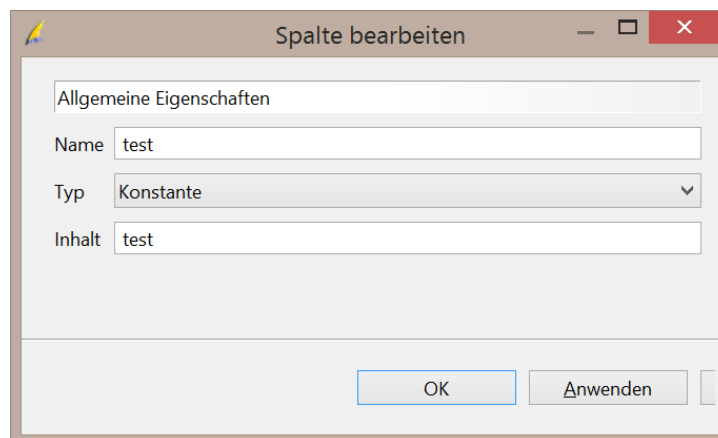


The first choice to make is the inclusion of a timestamp service. A timestamp service is provided by an independent third party and can provide evidence to the audit log of the real moment in time when certain events took place (Timestamp services configuration itself is described in another place).

After this the “business data” columns for the log are defined. Besides the “hard coded” audit columns

- auditCommand
- auditSequenceNumber
- auditTimestamp
- auditSeal

you can add any user defined column you want. To do this press the “+” button and define a new column.



You must enter a column name, type and content. The column type is either “Constant” or “Variable”. If you add a constant, every audit record

will get this content entered as its value for this column. This way you can separate log records that stem from different services.

Adding a variable column gives you dynamic access to all information available at the moment of service invocation.

The content is a string variable expression, any global namespace (see chapter “String Expansion”) can be used here. In addition, to get access to the service request context, the namespace “record” is available.

This namespace contains:

NAME	
<b>record</b>	
DESCRIPTION	
Dynamic access to request specific information.	
VARIABLES	
container	<p>Access the service container provided namespace (see below)</p> <p>Example:</p> <pre>record.container.id</pre> <p>will provide the container id you used upon service definition.</p>
service	<p>Access the service namespace, containing</p> <ul style="list-style-type: none"> <li>• config</li> <li>• context</li> <li>• container</li> </ul> <p>See below.</p>
success	“T” if call succeeded, “F” if call failed
args	<p>access all request arguments.</p> <p>Example:</p>



```
record.args.client
```

will provide the value of the “client” argument to the service.

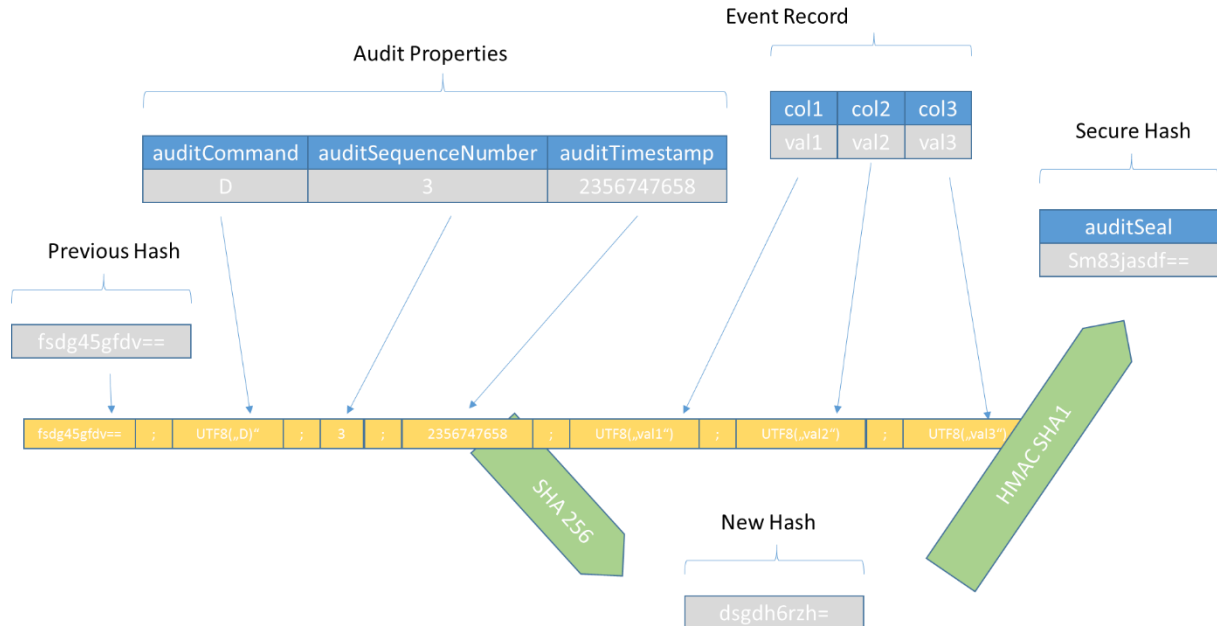
### 6.6.3.3 Audit mechanics

“Auditing” an event means adding properties that will render tampering impossible or at least hard.

Our audit achieves this by adding a cryptographic seal to each record. To compute the seal, an audit session is created upon container start. This session holds the audit configuration

- Sequence number (starting with 0)
- Digest algorithm (SHA256)
- Signature algorithm (HMAC SHA1)
- Session key (random)

The audit session creates or enhances records by prepending the audit command (see below), prepending and incrementing the sequence number, prepending the current timestamp (Unix time) and appending a secure hash (auditSeal).



The secure hash (auditSeal) is computed by

- create a digest from the following values, separated by “;”
  - the previous computed digest or nothing if first digest
  - auditCommand, UTF-8 encoded

- 4 byte sequence number in little endian encoding
- 8 byte timestamp in little endian encoding
- record data, UTF-8 encoded. The record data is computed by concatenating the string representation of each configured column, separated by “;”
- sign the computed digest using the audit session signature algorithm

### **“I”nitialization command**

First the session creates an initialization record. This record is marked with the audit command “I”, followed by an enumeration of all audit parameters (needed to make an audit validation), separated by “[”

- Protocol version
- Digest algorithm
- Signature algorithm
- Session key (random)
  - This key can be provided in different formats and is itself serialized in parantheses“(.)”
    - “P” | <base 64 encoded key>
    - “E” | <protocol version> | <base 64 encoded encrypted key>

### **“T”imestamp command**

If a timestamp service is configured, the audit session will create, at least after opening and after closing a session, a timestamp record with a “T” followed by the base 64 encoded timestamp.

### **“D”ata command**

For all external record, the audit session creates a data record with “D”

### **“H”eartbeat command**

A heartbeat is sent every minute to defend against deleting events from the end of the log.

### **“C”lose command**

To mark session end, a record with command “C” is written. This record may be followed by a timestamp record.

#### 6.6.3.4 Example

Here you see the example of an audit log, with user defined columns

- col1        The constant “constant value”
- col2        The variable “record.container.id”
- col3        The variable “record.args.FILENAME”

These columns are embedded in the standard audit columns

- auditCommand
- auditSequenceNumber
- auditTimestamp
- auditSeal

```
auditCommand;auditSequenceNumber;auditTimestamp;col1;col2;col3;auditSeal

I|1.0|SHA256|HmacSHA1|(P|atP...XZpU=);0;1408616507145;constant \
value;fsm;<record.args.FILENAME>;aNBsb7CzjM04tXxg0KWz4r8L7wk=

T|MIHxgYJKoZI...RNHZjro=;1;1408616507555;constant \
value;fsm;<record.args.FILENAME>;QZ7tQHubfYGpCLVNbhzuuBckktA=

D;2;1408616528920;constant value;fsm;test.pdf;HZkpkcAN6xa0fv+00HVruikOFBk=

C;3;1408616540570;constant value;fsm;<record.args.FILENAME>;wdJUCU0X8nCg/wA0ZmVxBiF8CpY=

T|MIHxgYJKoZ...mRjpy8o=;4;1408616540703;constant \
value;fsm;<record.args.FILENAME>;/PTMQN1//SgTPB5/Q9yQR1KHZUE=
```

## 6.7 Advanced service definition

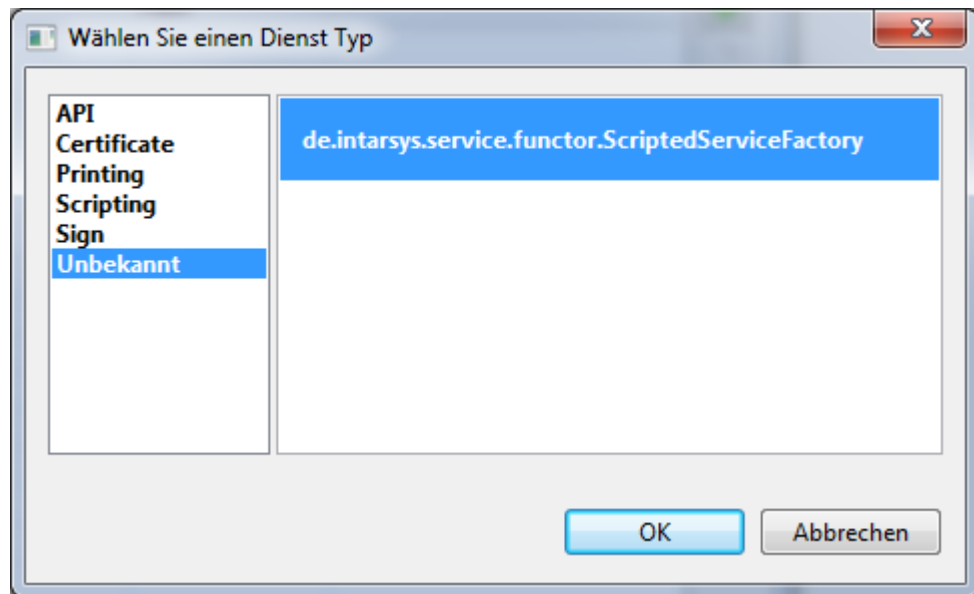
### 6.7.1 Scripted service factory

This scenario is very similar to creating an external script. But while with an external script you provide the required properties with each service definition (“ad hoc” service creation), the “Scripted Service Factory” makes it easy to reuse a prepackaged, external scripted service implementation as a whole. This is a template for others to use and refine. To do this, you create a “factory” for a service that already defines the necessary properties. As this is more of a developer task, you have to do this in a text editor by hand, there’s no GUI for this.

First, you create an instrument with an entry that will finally show up in the service creation dialog for user selection. In the simplest variant you use:

```
<extension point="de.intarsys.claptz.factories">
  <factory
    class="de.intarsys.service.functor.ScriptedServiceFactory"/>
</extension>
```

When opening the service creation dialog, a new type will show up – the category will be “unknown” and the service name defaults to the factory name.



Instances of this service will lookup the script implementations “init”, “invoke”, “error” and “destroy” in the subdirectory “de.intarsys.service.functor.ScriptedServiceFactory/scripts” of the instrument where you defined the **service factory**.

```
+--+ instruments
|
+--+ _DemoHelloWorld
|
+--+ de.intarsys.service.functor.ScriptedServiceFactory
| |
| +--+ scripts
| |
| | + destroy.js
| | + error.js
| | + init.js
| | + invoke.js
|
+--+ INSTRUMENT-INF
|
+--+ instrument.xml
```

This simple scheme is how a lot of internal, scripted services are deployed.

Now we want to make things a little bit more pleasant, functional and flexible. First, for the user interface add a category and some text. Second, we don’t want to have that cryptic directory name for the scripts. We opt simply for “scripts”. And third, we add some scripts. Put the following script in a file named “invoke.js” in the folder “scripts” in your newly created instrument.

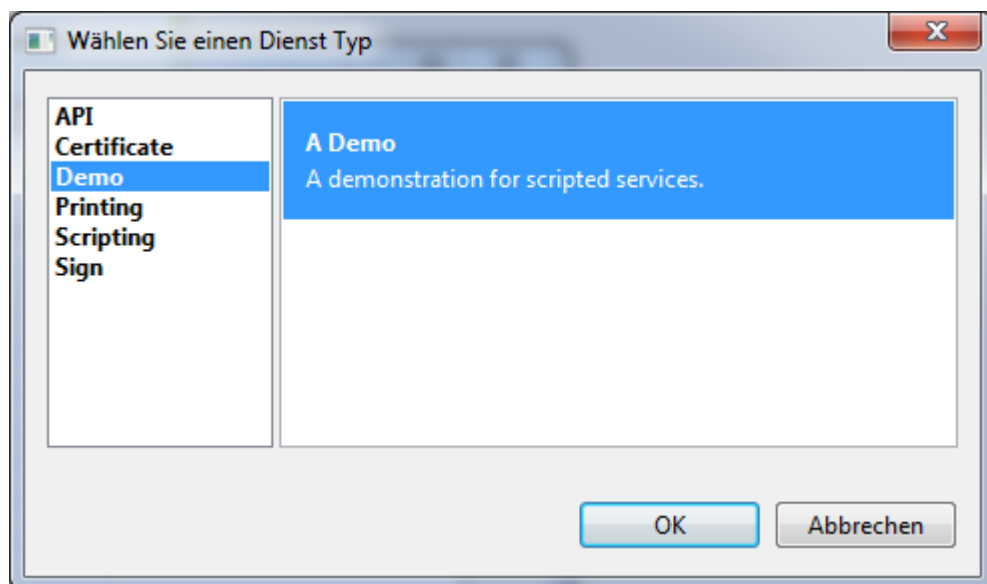
```
app.alert("Hello, World");
```

And then change your instrument declaration according to the following code example.

```
<extension point="com.cabaret.claptz.category.categories">
  <category
    class="de.intarsys.tools.category.GenericCategory"
    id="de.intarsys.category.demo"
    label="Demo"/>
</extension>

<extension point="de.intarsys.claptz.factories">
  <factory
    class="de.intarsys.service.functor.ScriptedServiceFactory"
    category="de.intarsys.category.demo"
    label="A Demo"
    description="A demonstration for scripted services."
    <instanceConfiguration scriptdir="scripts">
    </instanceConfiguration>
  </factory>
</extension>
```

Now you should be able to create service instances for your type simply in the user interface and deploy your service type cleanly packaged as a modular unit.



When executing a service instance, the lifecycle methods ("init", "invoke", "error", "destroy") are looked up first in the instrument defining the service instance, then in the instrument defining the service factory. This way you can override the default implementation in your definition if needed.

Create a new service and deploy it in a “File System Container”. You will see a “Hello, World” for every file dropped in.

### 6.7.2 Java based service factory

Much as the “external script service factory” from the example above to the “external script definition” we will show you here how you not only can bind a Pojo Java object (see above) but define a factory with the required properties for your users upfront.

Again, you create an instrument, in its simplest form:

```
<?xml version="1.0" ?>
<instrument id="de.intarsys.demo.service.helloworld">

  <requires>
    <prerequisite instrument="com.cabaret.service.kernel"/>
  </requires>

  <extension point="com.cabaret.claptz.category.categories">
    <category
      class="de.intarsys.tools.category.GenericCategory"
      id="com.cabaret.category.demo"
      label="Demo"/>
    </extension>

    <extension point="de.intarsys.claptz.factories">
      <factory
        class="de.intarsys.tools.factory.GenericFactory"
        id="de.intarsys.service.demo.pojo"
        label="Demo Java based service"
        description="A demo ..."
        resultClass="de.intarsys.service.api.IService"
        category="com.cabaret.category.demo" >
        <template
          class="de.intarsys.service.functor.ObjectBasedService"
          scope="application">
            <receiver
              class="de.intarsys.service.functor.TestPojo">
            </receiver>
          </template>
        </factory>
      </extension>
    </instrument>
```

After a restart, you can create your new service directly from the factory available in the UI.

### 6.7.3 Accessing context

For some more advanced scenarios you may want to add more flexibility to your service and access contextual information beyond service arguments. Here’s how to do it.

## 6.7.3.1 Service

## 6.7.3.2 Service context

## 6.7.3.3 Service container

## 6.7.3.4 String expansion

The service container “injects” its string variables automatically in the context while executing a service. This is why you can access the string evaluator simply by “TemplateEvaluator.get()” and get the whole context for free.

```
var TemplateEvaluator =  
    Packages.de.intarsys.tools.expression.TemplateEvaluator.get();  
var Args =  
    Packages.de.intarsys.tools.functor.Args;  
var result =  
    TemplateEvaluator.evaluate("${container.id}", Args.create());  
app.alert(result);
```

This code snippet expands a string with a reference to the active service container while running the script.

## 6.8 Session handling

The service framework supports the notion of “session handling”, where multiple requests are executed in the context of a common session. The session allows to remember state between successive calls.

A session has an expiration condition which is set upon session creation to the value defined in the container. Session creation semantics depends on the service container implementation.

A session is created and managed either explicitly or implicitly. Implicit sessions are based on the protocol sessions (like an HTTP session) described below. Explicit sessions are handled by the service framework itself using reserved arguments.

## 6.8.1 File system container

Each listener supports a single protocol session for each start/stop cycle.

## 6.8.2 Jetty

The session is bound to the HTTP protocol session.

From your client you can handle the protocol session as usual using cookies.

If you have a webservice (SOAP) client, you can request session handling also, e.g. via setting the property `BindingProvider.SESSION_MAINTAIN_PROPERTY` to true. You should

consult your webservice programming manual for more information on this.

### 6.8.3 Virtual printer

Each listener supports a single protocol session for each start/stop cycle.

### 6.8.4 Scheduler

There's currently no protocol session implemented.

### 6.8.5 Message queue

Each listener supports a single protocol session for each start/stop cycle.

## 6.8.6 Explicit session handling

### 6.8.6.1 Overview

Using the reserved argument “\_session” the client can control the session himself explicitly.

“\_session” is either a simple id that references a session in the container directly. In this case the session id must be requested by some API means from the container before. A reference to a non-existing session will result in an error.

Example

```
_session=234235-234s-23d2w3
...
```

“\_session” can also be an argument structure itself. The “id” member will hold the id to be used. Again it will be an error if the session does not exist.

Example

```
_session.id=234235-234s-23d2w3
```

### 6.8.6.2 Lazy creation

In some cases it is important that the client can initiate a session without explicitly calling an API. To support this scenario, the client can add either the flag “lazy” to the “\_session” argument structure. For “lazy”, the session will be created on the server automatically if it does not already exist.

You can provide additional creation arguments with this call (see below).



## Example

```
_session.id=234235-234s-23d2w3
_session.lazy=true
```

## 6.8.6.3 Perform session API calls

Using the property “perform” in the “\_session” argument structure allows direct access to the session API. The referenced session feature will be executed and the call is returned immediately. No service level code will be performed!

As always, the target session can be referenced either explicitly via “\_session.id” or implicitly via the protocol session semantics.

## 6.8.6.4 Perform “create”

“create” allows the explicit session creation from client. You can either provide a session id or create a default id on the server. If the session with the referenced id already exists, an error is raised. If another session is already active, an error is raised.

You can provide additional creation arguments with this call (see below).

## Example

```
_session.id=234235-234s-23d2w3
_session.perform=create
...
```

## 6.8.6.5 Perform “dispose”

“dispose” allows the explicit session destruction from client. If the session does not exist, an error is raised.

## Example

```
_session.id=234235-234s-23d2w3
_session.perform=dispose
...
```

## 6.8.6.6 Additional creation arguments

When creating a session lazy in the scenario above, the client can in addition send information on the session expiration condition. The expiration is defined in the argument “expire”. “expire” is an argument structure again, holding the members “type” and “value”.

type	description	value
timeout	Define a timeout value in milliseconds. The timeout is	integer

	reset each time the session is used.	
after	Define a lifetime in milliseconds. After this lifetime the session is expired regardless of use.	integer
at	Define an exact expiration time when the session is no longer valid. The value is measured in milliseconds from midnight, January 1, 1970 UTC	integer
and	Define an expiration predicate that is expired when both operands are expired	expiration operators "op1" and "op2"
or	Define an expiration predicate that is expired when one of its operands is expired	expiration operators "op1" and "op2"

### Example

```
_session.id=234235-234s-23d2w3
_session.lazy=true
_session.expire.type=timeout
_session.expire.value=6000
```

### Example

```
_session.id=234235-234s-23d2w3
_session.lazy=true
_session.expire.type=or
_session.expire.value.op1.type=timeout
_session.expire.value.op1.value=1000
_session.expire.value.op2.type=after
_session.expire.value.op2.value=100000
```

## 6.9 String expansion

Many service implementations support string expansion. In this context, beside the default string expansion namespaces, the following namespaces are available.

NAME
<b>config</b>
DESCRIPTION

## Access service specific information

**VARIABLES**

id	The service id (as specified in the service properties dialog)
path	The path mapping for this service. This can be configured in the advanced settings. The default for the path is “/”.
basedir	The base directory for this service instance. This directory points to the basedir of the instrument that defined the service.
workingdir	The working directory for this service instance. This is the same as the default environment.
profiledir	The profile directory for this service instance. This is the same as the default environment.
tempdir	The temp directory for this service instance. This is the same as the default environment.
param.*	Access any declared initialization parameter.

**AVAILABILITY**

In service configuration and execution context

**EXAMPLE**

```
example ${config.id}
```

```
➔ example test
```

```
example ${config.param.myparam}
```

→ example a parameter value

## NAME

**context**

## DESCRIPTION

Access service specific information

## VARIABLES

id	The service context id. As internally every service is deployed in the root context, this is currently always "".
path	The path for the service context. As internally every service is deployed in the root context, this is currently always "/".
basedir	The base directory for this service context instance. This is the same as the containers environment.
workingdir	The working directory for this service context instance. This is the same as the default environment.
profiledir	The profile directory for this service context instance. This is the same as the default environment.
tempdir	The temp directory for this service context instance. This is the same as the default environment.
param.*	Access any declared initialization parameter. There's currently no user interface for declaring context parameters. As we have a single context per container, you can easily revert to container parameters.

## AVAILABILITY

In service configuration and execution context

## NAME

**container**

---

**DESCRIPTION**

Access service specific information

---

**VARIABLES**

id	The service container id.
basedir	The base directory for this service container instance. This directory points to the basedir of the instrument that defined the container.
workingdir	The working directory for this service container instance. This is the same as the default environment.
profiledir	The profile directory for this service container instance. This is the same as the default environment.
tempdir	The temp directory for this service container instance. This is the same as the default environment.
param.*	Access any declared initialization parameter.

---

**AVAILABILITY**

In service configuration and execution context

---

**EXAMPLE**

```
example ${container.id}
```

```
➔ example test
```

```
example ${container.param.myparam}
```

➔ example a parameter value

## 7. Batch

### 7.1 Overview

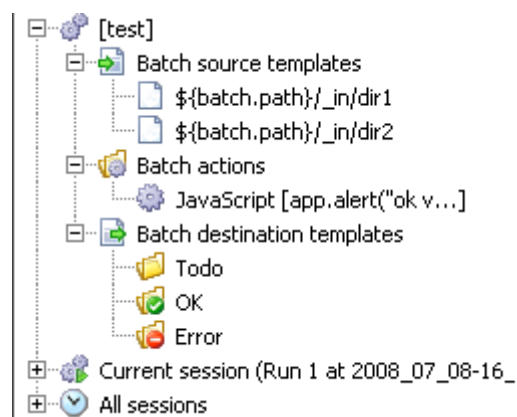
The batch framework allows for interactive or automatic processing of predefined batch scripts. A batch script processes each element from a list of input documents with one or more actions. After processing the processed documents are copied or moved to target folders.

### 7.2 Batch Creation

While you *could* edit a Sign Live! CC batch file in an XML editor, you should not. The Sign Live! CC GUI has a full-fledged batch editor available, which gives you access to and information about the batch's features.

To access the batch GUI you simply need to open a batch file or create a new batch definition from the *Extra* menu.

All features described in this chapter are available via context menus and buttons in the batch GUI.



You can see the following components in the screenshot above.

The *[test]* is the batch definition. The batch is called *test* and has two input folders (batch source templates), one JavaScript action and the three default target folders (batch destination templates).

The *Current Session* denotes the currently selected files in the input folders ready for processing.

*All Sessions* shows the history of runs for this batch.

### 7.3 Define a Batch

### 7.3.1 The Toolbar



The toolbar seen above has the following buttons of interest for this guide which will be discussed below.

Batch Settings: Third button.

Open Settings: Eighth button, available if you click on appropriate items.

The other buttons are self-explanatory in their behavior; just hover for a tooltip.

### 7.3.2 The Lifecycle

The batch is processed in the following order.

- Process all declarations
- Collect and create the BatchTargets.
- Provide the BatchTargets for the BatchDestination *todo*.  
Dependent on the configuration of '*todo*' they are linked, moved or copied.
- Process *begin* of all batch actions in the order of declaration.
- Process *visit* for each BatchTarget of all batch actions in order of declaration.
- Process *end* of all batch actions in the order of declaration.

### 7.3.3 Batch Settings

In the batch settings you are able to define the arguments, attributes and general settings for the batch itself.

Arguments are used to provide external information to the batch run. Attributes are variables that can be used by the batch and are transported over the lifecycle of the batch. The general settings include base folder and logging options.

In all configurations you can use existing attributes. For example:

```
Base Directory: ${batch.locator.parent.fullName}
```

This sets the batch's base directory to the location of the batch file itself.

### 7.3.4 Batch Source Templates

Here you find the selected input folders. This can be one or more directories containing subdirectories.

You can add source templates using the appropriate button and then edit the settings of this template.



Settings are:

- Path: The path denoting the source template.
- Filter: A file extension or file name filter.
- Attachments: Files to attach to each *Target* in this source template.
- Apply to all subdirectories: Parse subdirectories of this source template.

Examples:

```
Path: ${batch.path}/_in/dir1
```

Sets the path to a directory named */\_in/dir* located in the base folder of the batch file itself.

```
Filter: .*\.pdf .*\.txt
```

Selects all PDF and text files.

```
Attachments: pkcs7;p7s
```

Attaches same named PKCS#7 container files to the selected batch targets.

### 7.3.5 Batch Destination Templates

There are three predefined batch destination templates:

- Todo: All batch targets are transported here prior to processing.
- OK: Batch targets on which actions were performed without returning false (meaning no error has occurred) are transported here.
- Error: Batch targets on which actions were performed returning false are transported here.

The settings for each of these three are the same:

- Destination Name: This is fixed (todo, ok, error).
- Destination Label: A label of your liking for the destination label.
- Copy Source: If this is active, the source templates are copied here.
- Path template: The destination path declaration.
- Delete source: If this is active the source templates are deleted after processing.

For the *OK* and *Error* destination templates *source template* means the *Todo* destination template.

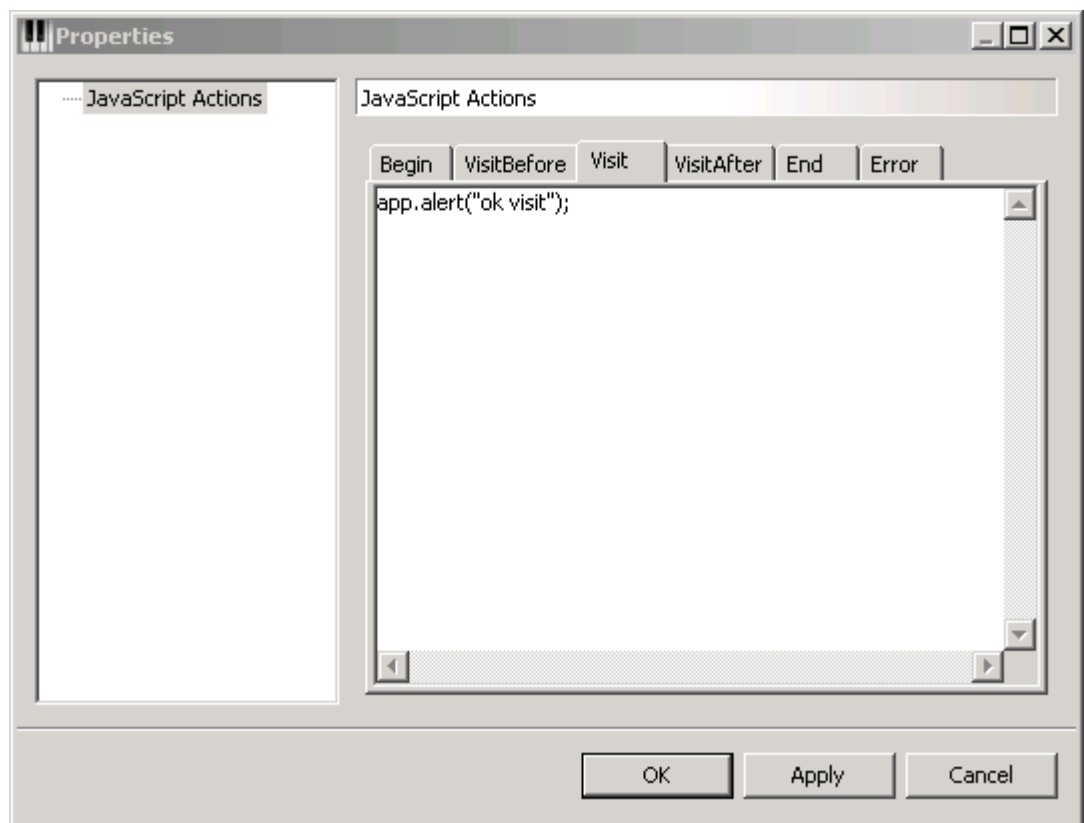
A typical scenario for document movement is: Process files in a source folder, leave them there, but create processed files in *OK* and *Error*. To achieve this you need to set the *Copy Source* and *Delete Source* checkboxes as follows:

- For the *Todo*: Copy Source active, Delete Source inactive.
- For the *OK* and *Error*: Copy Source active, Delete Source active.

### 7.3.6 Batch Actions

Batch actions declare the process to be performed for each batch target. After adding an action to the batch (you can have several), you can modify its content.

The action definition dialog looks like this:



You see seven method labels that can be declared for each action. Only *visit* needs to be declared in order to actually do something for each *BatchTarget*.

- **begin**  
This method will be called before processing all *BatchTargets*. If this method returns false the batch process is stopped. This is the initialization part done before any processing happens.
- **visitBefore**  
If the *BatchTarget* is a folder node this method is called before

descending in the folder. If the method returns false all further processing for upcoming BatchTargets is stopped. All BatchActions and *visitAfter* events for the already visited BatchTargets are still executed.

- **visit**  
If the BatchTarget is a leaf node (an actual object) this method is called. Further processing is stopped if the method returns false. All BatchActions for the current BatchTarget and *visitAfter* events for the already visited BatchTargets are still executed.
- **visitAfter**  
If the BatchTarget is a folder node this method is called *after* descending in and processing the folder. If the method returns false all further processing for upcoming BatchTargets is stopped. All BatchActions for the current BatchTarget and *visitAfter* events for the already visited BatchTargets are still executed.
- **end**  
This method is **always** called after the batch has been processed, even if *begin*, any *visit* or any other errors have stopped the batch.
- **error**  
This method calls upon an error condition. Further processing is stopped if this method returns false. All BatchActions for the current BatchTarget and *visitAfter* events for the already visited BatchTargets are still executed.

## 7.4 String Expansion

### 7.4.1 Overview

You can refer to string variables in a batch script on several occasions. These string variables will be expanded at runtime and yield the desired value. Please refer to the the appendix for more details.

For example

```
hello, ${properties.greeting}!
```

At runtime this string expression will be expanded. Given the command line *-Dgreeting=world* the output would be:

```
hello, world!
```

## 7.5 Advanced Features

### 7.5.1 Attachments

A BatchTarget can receive one or more attachments. This attachment will be treated as a BatchTarget in functions like *transfer* or *delete*. It is never used as event.target for the BatchAction though.

This feature plays an important role in situations where the input for the Batch (the BatchTarget) consists of more than one file. Using attachments, these files can be bundled into one BatchTarget. For example, you can have a batch using *\*.xml* as its BatchTargets and additionally add a *?pdf* for each of the XML files as an attachment to the target.

## 8. Appendix

### 8.1 String Expansion

#### 8.1.1 Basics

##### 8.1.1.1 Why string expansion

Any non trivial application needs variables to be adaptable to the usage context. Examples for this are

- configurable path to store temporary files
- host names to lookup information
- database URL's

String expansion allows for runtime replacement of dynamic content within strings.

Strings are used often in Sign Live! CC's configuration and installation. Using string replacement, you gain access to environment information or runtime information, which gives you more flexibility during deployment and operation.

```
<extension point="com.cabaret.scripting.executableresolver">
  <executableresolver
    class="com.cabaret.scripting.common.SearchPathResolver"

searchpath="c:/programme/my_installation/instruments/MyInstrument/scripts"/>
</extension>
```

This declaration, for example, defines a search path for scripts in the folder *scripts* of a given instrument. What if you rename the instruments folder or if you deliver the instrument, but are unsure about the final name?

```
<extension point="com.cabaret.scripting.executableresolver">
  <executableresolver
    class="com.cabaret.scripting.common.SearchPathResolver"
    searchpath="${instrument.basedir}/scripts"/>
</extension>
```

This declaration uses a variable that contains the installation folder of the current instrument. Using this you don't have to worry about delivery and installation of your instrument.

#### 8.1.1.2 Where we use string expansion

String expansion is used heavily throughout the application. The first and most important usage you will find is parsing the instrument declaration. **Every attribute value is expanded before use.** This way you can easily prepare your declarations for the use in dynamic context.

Other examples you may find when configuring labels or names in the application that have some need for dynamic adaption. Many application parts are able to expand the strings entered by the user in order to get more dynamic information. One example you can find in the printer preferences, where the print job name can be defined using templates.

Depending on where we use the string expansion technique, different variables are available. When starting the system, only the Java VM and execution context may be available. Later on, variables brought into the application by some of the instruments or based on preferences may come up. Special processors like the printer may add further, more dynamic variable content that does not make sense in any other context, like a counter for print jobs already processed.

You see that context matters - when using variables, always ensure which ones are available at the moment.

#### 8.1.1.3 Terminology

When talking about string expansion, we use two different concepts:

- Expression evaluation

"Expression evaluation" means replacing a variable name with its content, much like in any programming language you may know. If the variable `foo` has the value `bar`, then evaluating `foo` means replacing it with its value `bar`. Using this variable for example in a script, you can adapt it more easily to different customer needs by simply changing the variable.

- Template evaluation

While expression evaluation is quite useful by itself, it still can be improved. One problem you will encounter is how to differentiate a plain piece of text from a variable to be replaced, the other problem results from the need for more complex content that cannot be expressed using a single variable. Look for example at a directory name

that should be built using the temporary directory plus the name of the current user.

These problems are addressed by the next level of evaluation - a template based approach. A template is first a plain string. “Hello” is simply “Hello”. But a template is evaluated and scanned for special escapes, marking the embedding of an expression. If it encounters such an expression, it is evaluated as described above and inserted in the original template string.

This is a quite common scenario and we will use it here to build our powerful string expansion. Our escapes will be `${` for marking the beginning and `}` for the end.

So, let’s expand “Hello, `${username}`”. Supposed there is a variable *username* containing the value *Nick* we will get the string “Hello, Nick”. It’s that simple.

#### 8.1.1.4 Syntax

Embedding a variable in a string is preceded by `${` and ends with `}`. Enclosed is the name of the variable to be expanded.

```
hello, ${foo}
```

The variable *foo* is embedded in the string.

Variables are usually prefixed with a name space, which consequently eases search and provides information about the use of the variable.

```
hello, ${stage.basedir}
```

If you need a `${` in the text itself, you simply enclose it in expression escapes itself

```
`${$}` is the start escape sequence
```

will evaluate to

```
`$` is the start escape sequence.
```

#### 8.1.1.5 Constant text in expression

While it seems a little bit strange, constant text within an expression does make sense. The reason are the formatting features mentioned in a later chapter. They reach from number or date formatting to conditional evaluation. This is where constants come into play - you can define constant text that may not be contained in the evaluated template.

```
hello, ${"world"}
```

will evaluate to

```
hello, world.
```

#### 8.1.1.6 Namespaces

String expansion is currently about embedding. But as there is a lot of information around that is interesting in some context and many sources may supply their specific local information, variables are organized in namespaces. This is simply done by segmenting variable names with a dot ".", indicating that the prefix selects a certain namespace. For example our information is organized using the prefixes

- `properties` for selecting among VM properties
- `environment` for selecting execution context information like working directories

and many more.

Namespaces may be hierarchical to any depth.

Example

```
foo.bar.gnu.gnat.var
```

This is a valid name for "var" in the namespace "foo.bar.gnu.gnat"

You will find a complete description of the namespaces available in the next chapters.

### 8.1.2 Namespaces

#### 8.1.2.1 Overview

Here we will learn about the most important namespaces and their respective variables available in Sign Live! CC. All of these namespaces are available in all expansion contexts in Sign Live! CC.

More information on special situations where you will have more information at hand will be found in the next chapter.

#### 8.1.2.2 system

---

##### NAME

system

---

##### DESCRIPTION



Access the Java system information.

---

**VARIABLES**

Name	Description
millis	The current time in milliseconds
time	Synonym for the above
uniquetime	This gives a unique tag based on the current time in milliseconds. If this value is requested multiple times within millisecond range, it will still be returning a unique value by simply adding “1” to the value last returned.
counter	A system wide counter for creating unique ids. The counter starts with one upon the start of the VM and is incremented after each use.
counters.<name>	A named counter for creating unique ids. A counter for <name> is created upon the first use, initialized to 0 and incremented after each use.
uuid	A random unique id
getenv.<name>	The value of the system environment variable “name”
properties.<name>	The value of the JRE property “name”

---

**AVAILABILITY**

This namespace is generally available.

---

**EXAMPLE**

```
example ${system.counter}
```

will evaluate to 0 when used for the first time and be incremented afterwards

➔ example 32

```
example ${system.counters.test}
```

will evaluate to 0 when used for the first time and be incremented afterwards

➔ example 0

### 8.1.2.3 properties

#### NAME

properties

#### DESCRIPTION

Access the VM property values. Remember, for the VM, additional property values can be defined on the commandline using the **-D** option. Some of the more interesting variables are enumerated here.

#### VARIABLES

Name	Description
<name>	Any entry in the Java VM properties map
user.name	The login name of the current user
user.dir	The working dir of the current application
user.home	The home directory for the current user
user.country	The ISO country code for the running system
java.io.tmpdir	The path to a temporary directory
os.name	The operating system name, for example <i>Windows XP</i>

#### AVAILABILITY

This namespace is generally available.

### EXAMPLE

```
example ${properties.user.name}
```

will evaluate to something like

➔ example jim

#### 8.1.2.4 environment

## NAME \_\_\_\_\_

environment

## DESCRIPTION

Access the current file environment.

## VARIABLES

Name	Description
basedir	The directory where the system is installed
profiledir	The directory where user specific information for the system may be stored.
workingdir	The current working directory for the system.
tempdir	A directory where temporary files may be stored.

## AVAILABILITY

This namespace is generally available.

### EXAMPLE

```
example ${environment.basedir}
```

will evaluate to something like

```
➔ example c:\programs\foo
```

8.1.2.5 stage

NAME	
stage	
DESCRIPTION	
Some information about the application itself.	
VARIABLES	
Name	Description
name	The name of the running instance.
version	The version of the running instance.
AVAILABILITY	
This namespace is generally available.	

EXAMPLE

```
example ${stage.name}
```

will evaluate to something like

```
➔ example SignLiveCC
```

8.1.2.6 preferences

NAME
preferences
DESCRIPTION
Gives you access to any preferences value below the “main” application preferences node. See more on the topic of preferences in the “Developer’s Guide”.
VARIABLES

The preferences node and property names are mapped directly to the name to be expanded. Preferences hierarchy is expressed using the dot notation.

---

#### AVAILABILITY

This namespace is generally available.

---

#### EXAMPLE

```
example ${preferences.interactive.shellBounds}
```

will evaluate to something like

```
→ example 777;7;896;1037
```

#### 8.1.2.7 preferencesRoot

---

##### NAME

preferencesRoot

---

##### DESCRIPTION

Gives you access to any preferences value below the root preferences node.

---

##### VARIABLES

The preferences node and property names are mapped directly to the name to be expanded. Preferences hierarchy is expressed using the dot notation.

---

##### AVAILABILITY

This namespace is generally available.

---

##### EXAMPLE

```
example ${preferencesRoot.foo.bar}
```

will evaluate to something like

```
→ example 42
```

depending on what you have stored at the property “bar” in node “foo”.

#### 8.1.2.8 variables

---

##### NAME

## variables

---

**DESCRIPTION**

Access to user defined variables. User defined variables are organized in namespaces themselves. This allows for defining and accessing as many groups of variables as desired. More about user variable declaration you can learn in chapter "Variables" in this document.

The predefined sub-namespaces are:

- **user**  
Access any user defined variables, stored on a per-user base. Each user has an own set of variables (see operators guide).
- **global**  
Access user defined variables, available on a per-installation base.

---

**VARIABLES**

Any variable defined for the respective sub-namespace.

---

**AVAILABILITY**

This namespace is generally available. Additional sub-namespaces can be defined.

---

**EXAMPLE**

```
example ${variables.user.greeting}
```

will evaluate to something like

```
→ example hello, jim
```

### 8.1.3 Special Namespaces

#### 8.1.3.1 Overview

Here we summarize some of the more common special situations where you will have additional information at your hand for evaluation of template strings.

This description can't be complete, as any application component can opt to provide more variables for the expansion process. In these cases, see the respective documentation for the component.

#### 8.1.3.2 The instrument namespace

---

**NAME**

instrument

---

**DESCRIPTION**

Within the instrument declaration you have two additional namespaces to ease customizing, one of it being `instrument`. This namespace contains reflective information about the instrument itself

---

**VARIABLES**

Name	Description
<code>basedir</code>	The directory where the <i>INSTRUMENT-INF</i> directory is contained.
<code>id</code>	The instrument id

---

**AVAILABILITY**

Within the instrument declaration file

---

**EXAMPLE**

With the instrument *foo* declared in the directory *c:\instruments\bar*

```
${instrument.basedir}
```

will evaluate to

```
➔ c:\instruments\bar
```

### 8.1.3.3 The bundle namespace

---

**NAME**

`bundle`

---

**DESCRIPTION**

Within the instrument declaration you have two additional namespaces to ease customizing, one of it being `bundle`. This namespace gives you access to NLS information contained in the instrument bundles.

---

**VARIABLES**

Any property contained in the resource file for the instrument can be accessed using its name. The variable will be replaced with the correct value for the current platform language.

---

**AVAILABILITY**

Within the instrument declaration file, if a *bundle* is declared (see the description of the instrument declaration file for information on the *bundle* attribute).

---

**EXAMPLE**

With “greeting” defined in the resource files for the instrument and appropriate resources provided for a “es” platform

```
${bundle.greeting}
```

will evaluate to

```
➔ Buenos dias!
```

## 8.1.4 Formatting

### 8.1.4.1 Overview

The content provided by the variables may sometimes be not well suited for use in a template. For example take the `${system.millis}` which will expand to something like `162336576223`. If you use this, you most probably really want to see a formatted date, like `23.12.1987`.

In such cases you can post process the variable content using formatting instructions.

The formatting instruction is appended immediately to the variable expression, separated by a `:`.

```
${foo:f}
```

or, in the case of hierarchical names

```
${foo.bar:f}
```

Formatting instructions process the result of the expression they are appended to. As such you can simply chain formatting instructions by simply adding more `:` separated instructions.

```
${foo.bar:*:dts}
```

This will recursively expand *foo.bar* (more to this later) and format the result as a date.



## 8.1.4.2 String formatting

## 8.1.4.2.1 Overview

String formatting is initiated by `s`. This conversion is applied by default.

This instruction allows you to convert the result of the evaluation to a string (if not already) and create suitable substrings.

If the result of the evaluation is not a String and no string conversion instruction is present, the default conversion is used.

## 8.1.4.2.2 Instruction

```
expr ":s" [ "(from, to)" ]
```

The evaluation result is converted to a string. The substring extending from *from* (inclusive) to *to* (inclusive) is used as the result of the formatting operation. If any of the values *from* or *to* is negative, the index is computed from the end of the string where `-1` is the last character in the string. The second parameter *to* may be omitted and is replaced to match the last character in the string.

## 8.1.4.2.3 Examples

With *variables.user.greeting* containing `hello, world`

```
example ${variables.user.greeting:s}
```

evaluates to

```
→ example hello, world
```

```
example ${variables.user.greeting:s(7)}
```

evaluates to

```
→ example world
```

```
example ${variables.user.greeting:s(0,4)}
```

evaluates to

```
➔ example hello
```

#### 8.1.4.3 Integer formatting

Integer number formatting is initiated by `i`.

This instruction allows you to convert number values to integer string representations.

##### 8.1.4.3.1 Instruction

```
expr ":i" [ "b" | "o" | "d" | "x" ]
```

The evaluation result is checked to be a number or convertible to a number. The integer part of the number is then returned as a string, written to the base defined as the second character in the instruction.

- `b` Binary representation
- `o` Octal representation
- `d` Decimal representation
- `x` Hexadecimal representation

##### 8.1.4.3.2 Examples

With *system.counter* containing '17'

```
example ${system.counter:i}
```

evaluates to

```
➔ example 17
```

```
example ${system.counter:ib}
```

evaluates to

```
➔ example 10001
```

```
example ${system.counter:io}
```

evaluates to

→ example 21

```
example ${system.counter:id}
```

evaluates to

→ example 17

```
example ${system.counter:ix}
```

evaluates to

→ example 11

#### 8.1.4.4 Date formatting

Date formatting is initiated by `d`.

This instruction allows you to convert date values to “human readable” strings.

##### 8.1.4.4.1 Instruction

There are two flavors of date formatting, one using instruction characters that quickly reference a predefined format and the other using “pattern” strings that exactly describe the desired output format.

The evaluation result must be a date or a number. This date will be formatted. If the evaluation result is already a string, this string is returned without processing. Any other object will return an empty string.

```
expr ":d" [ "d" | "t" ] [ "s" | "m" | "f" ]
```

This first syntax creates output based on a predefined format. This formatting will always use the platform locale.

The optional first instruction character defines which part of the date will be used for formatting

- no character Date and time portion will be processed
- `d` Only the date portion will be used
- `t` Only the time portion will be used

The optional second instruction character defines the output format

- no character The full formatting is applied
- s Short formatting is applied
- m Medium formatting is applied
- f Full formatting is applied

With no formatting character available at all, a default formatting pattern is used suitable for a technical representation of a timestamp in a file name.

```
expr ":d(" pattern ")"
```

This second syntax creates output based on the format defined in the pattern. The pattern syntax is exactly as described by the standard Java runtime library. This formatting will always use the platform locale.

#### 8.1.4.4.2 Examples

With 'system.millis' containing a timestamp for the 12 of october, 2009 at 23 :33:11 and 1 milliseconds.

```
example ${system.millis:d}
```

evaluates to

```
→ example 2009_10_12-23_33_11_001
```

```
example ${system.millis:ddm}
```

evaluates to

```
→ example 12.10.2009
```

```
example ${system.millis:dts}
```

evaluates to

```
→ example 23:33:11
```

#### 8.1.4.5 Float formatting

Float formatting is initiated by `f`.

This instruction allows you to convert numeric values to strings using a predefined pattern.

##### 8.1.4.5.1 Instruction

The evaluation result must be a number or convertible to a number. This number will be formatted.

```
expr ":f(" pattern ")"
```

This syntax creates output based on the format defined in the pattern. The pattern syntax is exactly as described by the standard Java runtime library. This formatting will always use the platform locale.

##### 8.1.4.5.2 Examples

With *variables.user.price* containing 1234.567 and an US locale

```
example ${variables.user.price.f(0.0)}
```

evaluates to

```
→ example 1,234.6
```

```
example ${variables.user.price.f(000000)}
```

evaluates to

```
→ example 001235
```

#### 8.1.4.6 File path formatting

File path formatting is initiated by `p`.

This instruction allows you to convert an evaluation result to a string that can be accepted by the underlying platform as a valid file name (including path separators). Every suspect character is simply replaced by an underscore `_`.

##### 8.1.4.6.1 Instruction

```
expr ":p"
```

The evaluation result is converted to a string, then every suspect character is replaced by an underscore.

#### 8.1.4.6.2 Examples

With *variables.user.foo* containing *my\*.file*

```
example ${variables.user.foo:p}
```

evaluates to

```
→ example my__.file
```

#### 8.1.4.7 Iteration

The result of evaluating an expression may contain other variables - it needs to be reevaluated. For example in this environment

- *properties.user.name* equals Jim
- *variables.global.greeting* equals Hello, *\${properties.user.name}*
- *variables.global.startmessage* equals  
*\${variables.global.greeting}! Your application is fully functional!*

The last expression should evaluate to “Hello, Jim! Your application is fully functional!”.

Simply applying the declarations as you see above will lead to **Hello, *\${properties.user.name}*! Your application is fully functional!** - The second iteration of replacement is missing. To add iterative re-evaluation you must use this instruction.

##### 8.1.4.7.1 Instruction

- \* Recursively apply the string evaluation process to the result of evaluating this expression.

The nesting depth of recursive evaluations is restricted to 10.

Remember that you can chain formatting instructions, for example to apply a string formatting first, followed by a deep evaluation.

##### 8.1.4.7.2 Example

So, the complete example for the above should be:

```
${variables.global.greeting:*}! Your application is fully functional!
```

evaluates to

```
Hello, Jim! Your application is fully functional!
```

#### 8.1.4.8 Conditional evaluation

Sometimes a certain amount of decision is involved when expanding a template. A good example may be a template for a file to be moved by the system. If the file not already exists at the destination you want it to have the same name as the original file. But, if a file with this name is already present you don't want it to be overwritten. Instead you want the new file to get a new, unique name. You cannot create such a template for the filename with the features you have learned so far.

The solution is a "conditional" template. The result contains a certain part only if a condition associated with it is true. The host system evaluating the template injects the condition before evaluation.

##### 8.1.4.8.1 Instruction

```
expr "?:?" condition
```

The result of evaluating *expr* is inserted into the result value only if *condition* is true.

Currently only simple state variables are supported for the condition. Which state is available depends on the evaluation context and is described in the respective documentation.

If you need to express more complex dynamic expressions, you can always use the *Functor* approach defined in the next chapter. The outcome of the expression evaluation is totally at your command with no restriction on complexity and state.

##### 8.1.4.8.2 Example

In the file system monitor scenario mentioned above, the system will evaluate the template for the output file name twice: The first time with the variable *collision* set to *false*. If the result of evaluation is not unique, the template is reevaluated, this time with *collision* set to *true*.

The above case for example may be written:

```
${path}/{system.millis:collision}{".:?collision"}${filename}
```

If evaluated with *collision = false* the result looks like *c:/temp/mydir/myfile.txt*. With *collision=true* it looks like *c:/temp/mydir/2983749287.myfile.txt*.

##### 8.1.4.9 Functor

Anybody willing (or needing) to define more complicated formatting can use the "functor" approach. Anybody that doesn't feel comfortable with scripting probably shouldn't...

Applying a functor to an evaluation result allows you to program any formatting function you like, using any scripting language supported

(which is for example plain Java or JavaScript). First you have to use the standard way to define an action in Sign Live! CC. Second, you declare this action to be called on the evaluation result using the `#` instruction.

The action is called with the evaluation result as the first argument and the optional arguments to the instruction as the second argument.

#### 8.1.4.9.1 Instruction

```
expr ":@" action [ "(" arg [ "," arg ]* ")" ]
```

#### 8.1.4.9.2 Examples

With *variables.user.bar* containing *wazamba* and the action *upper* declared as returning its input in upper case characters

```
example ${variables.user.bar:#upper}
```

evaluates to

```
example WAZAMBA
```

### 8.1.5 Special Topics

#### 8.1.5.1 Expansion in the instrument.xml

Every attribute content in the extension point definition in the *instrument.xml* is expanded by the system immediately after parsing and before forwarding the values read to the real implementation components. This way you can easily add variables to all part of the system. But, this also means that you must take special care where the system components implement template evaluation themselves.

For example the file system monitoring component allows to define a template for the filenames where computation results are stored. This filename must be evaluated at runtime of the computation, not when the system is started - all files would have the same name in this case. So, this component is aware of templates and evaluates them itself. But this means that a string of the form *\${variable}* must be forwarded to the component without the instrument parser expanding it! The solution to this problem is simply escaping the expression. This is already described in the *Syntax* chapter.

```
${${}variable}
```

The expansion in the parsing phase will evaluate this to *\${variable}* and forward it to the file system monitor for later reevaluation.



This special situation applies to all attributes in extensionpoint definitions. The support of runtime evaluation of templates (needing escape in the instrument definition) is always documented along with the respective component.

#### 8.1.5.2 Debugging

With this action declaration you can inspect the result of using string expansion. Be aware that string expansion is applied on the instrument file upon parse time!

```
<action
  id="com.cabaret.demo.javascript.simplealert"
  label="Simple Alert">
  <effect>
    <perform type="JavaScript" source="
app.alert('${properties.user.name} ' )
    "/>
  </effect>
</action>
```

#### 8.1.5.3 Script integration

While there is no special syntax or global function for accessing string expansion in a script, you can use the plain Java implementation in nearly all script bindings.

This example will show you how to use plain string expansion in JavaScript

```
var evaluator =
Packages.de.intarsys.tools.expression.TemplateEvaluator.get();
var args = Packages.de.intarsys.tools.functor.Args.EMPTY;
var result = evaluator.evaluate("hello, ${properties.user.name}",
args);
```

## 8.2 Installation of PDF Printer

### 8.2.1 PDF Printer

Sign Live! CC can be used as a *PDF Printer*, given the proper configuration. The necessary instruments are already delivered and installed.

The following chapter describes the necessary steps to install and configure a *PDF Printer* on a Windows platform.

### 8.2.2 Installation

#### 8.2.2.1 Sign Live! CC

The installation of Sign Live! CC provides all necessary instruments. No further installation is required.

## 8.2.2.2 GNU Ghostscript

GNU Ghostscript is used to convert the received Postscript print data into PDF. The tested and supported version of Ghostscript is 8.54. You need to install Ghostscript according to its installation guide.

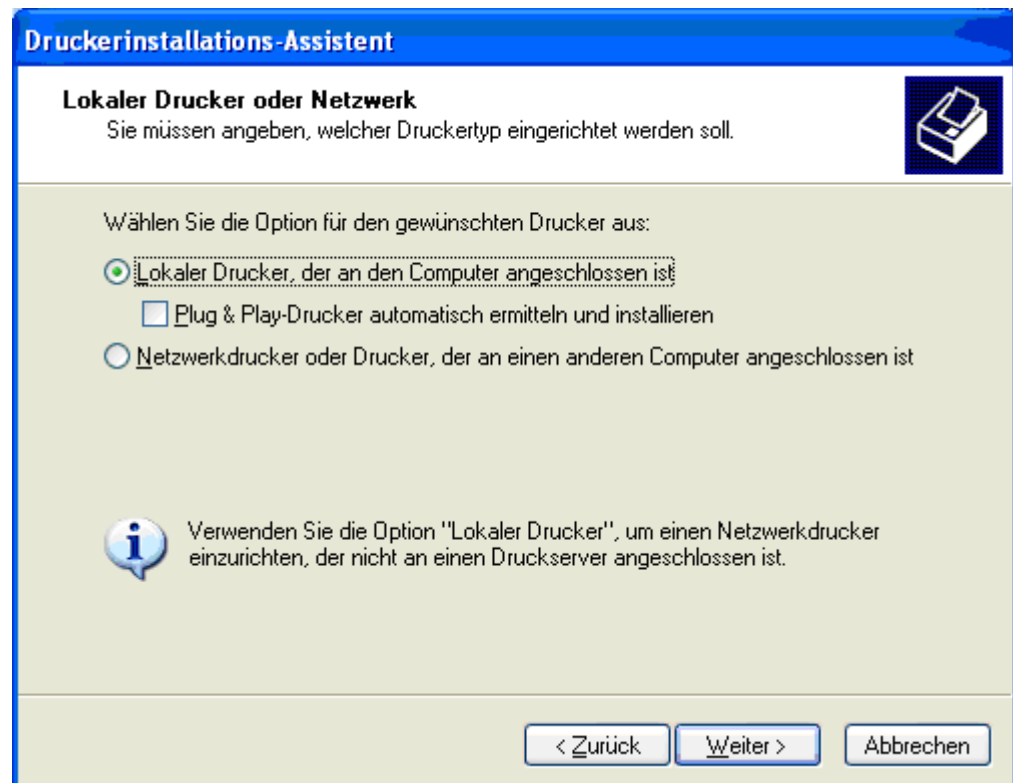
Ghostscript includes a specific Postscript printer driver, which yields the best results regarding PS to PDF. Please install this driver, too.

## 8.2.2.3 Create a Local PDF Printer

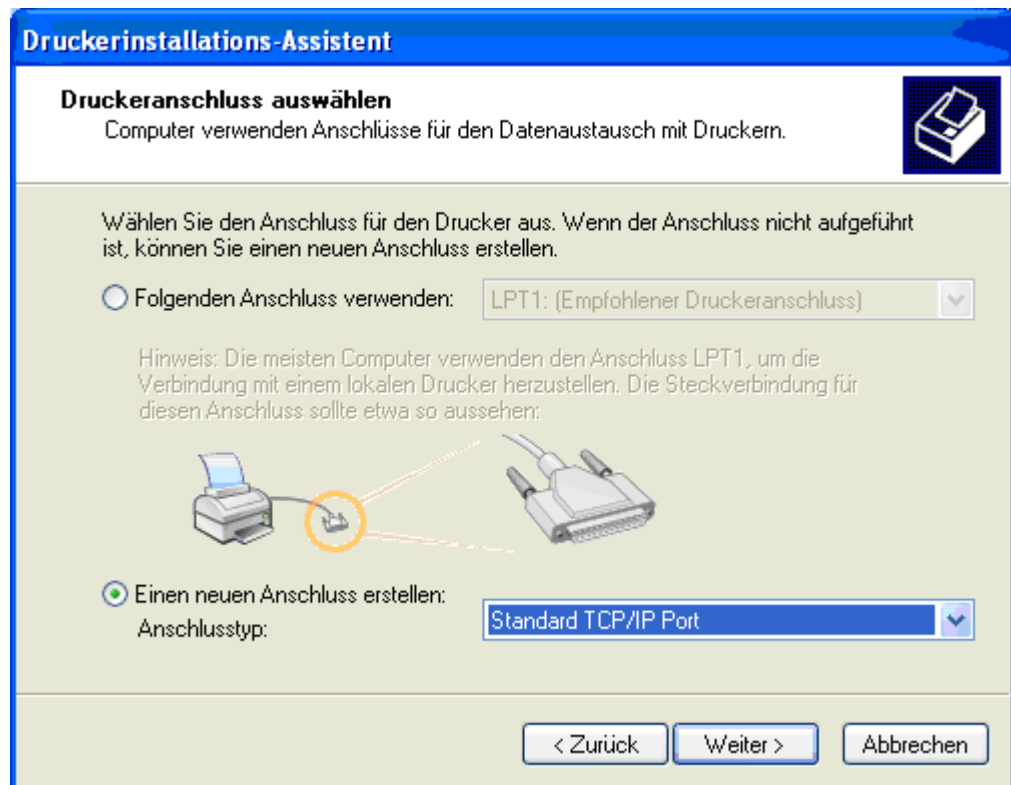
For Sign Live! CC to receive the print data you need to create a Postscript printer on your local machine. Normally this should be the printer driver from Ghostscript, which provides the same settings as a Acrobat Distiller.

You need to configure a local printer on your machine.

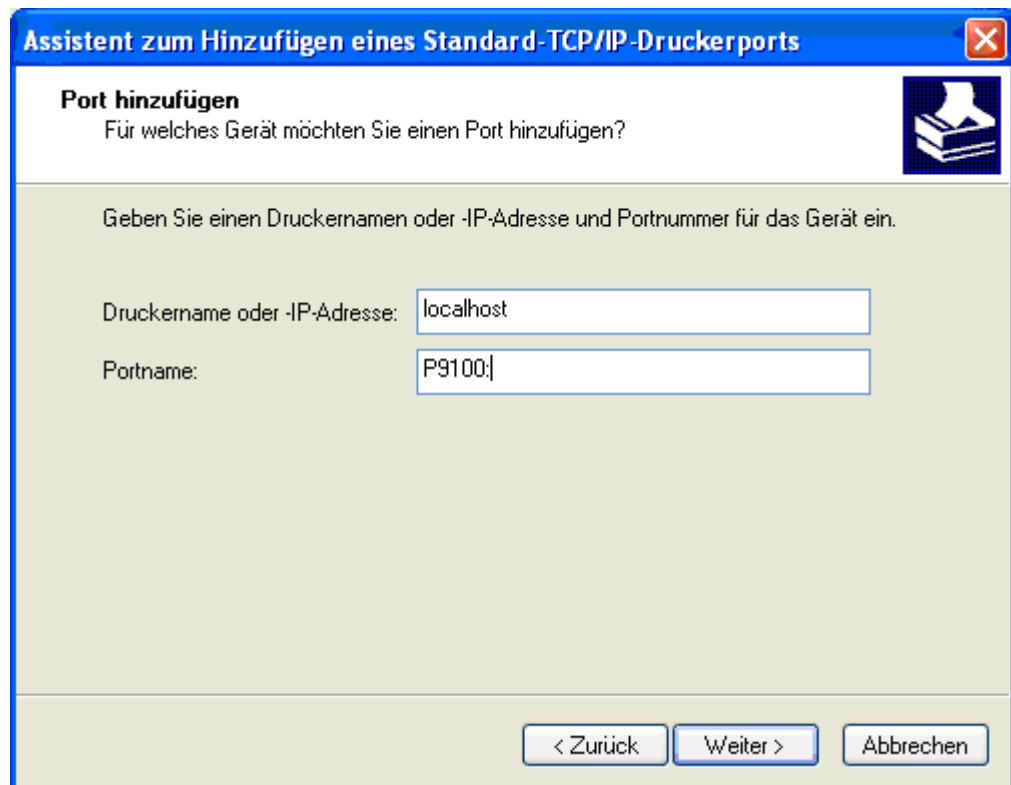
- Systemsteuerung → Drucker und Faxgeräte → “Drucker hinzufügen”.
- Select “Local Printer” without Plug & Play.



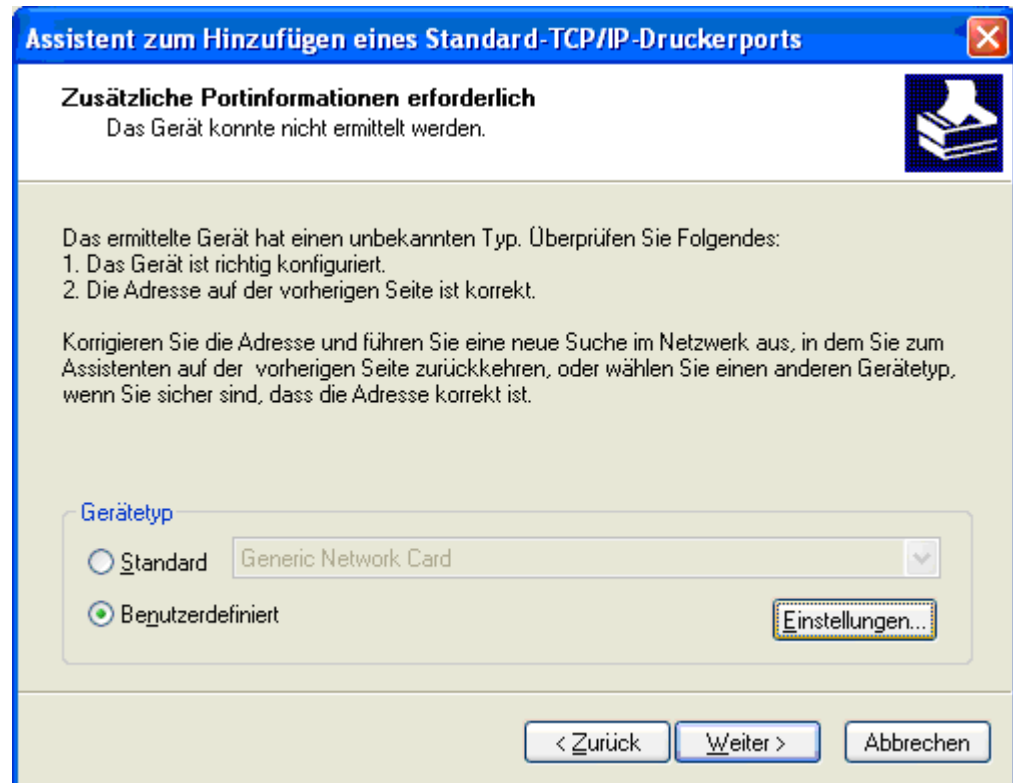
- For “Printer Connection” select “Create a new connection” and then “Standard TCPIP Port”.



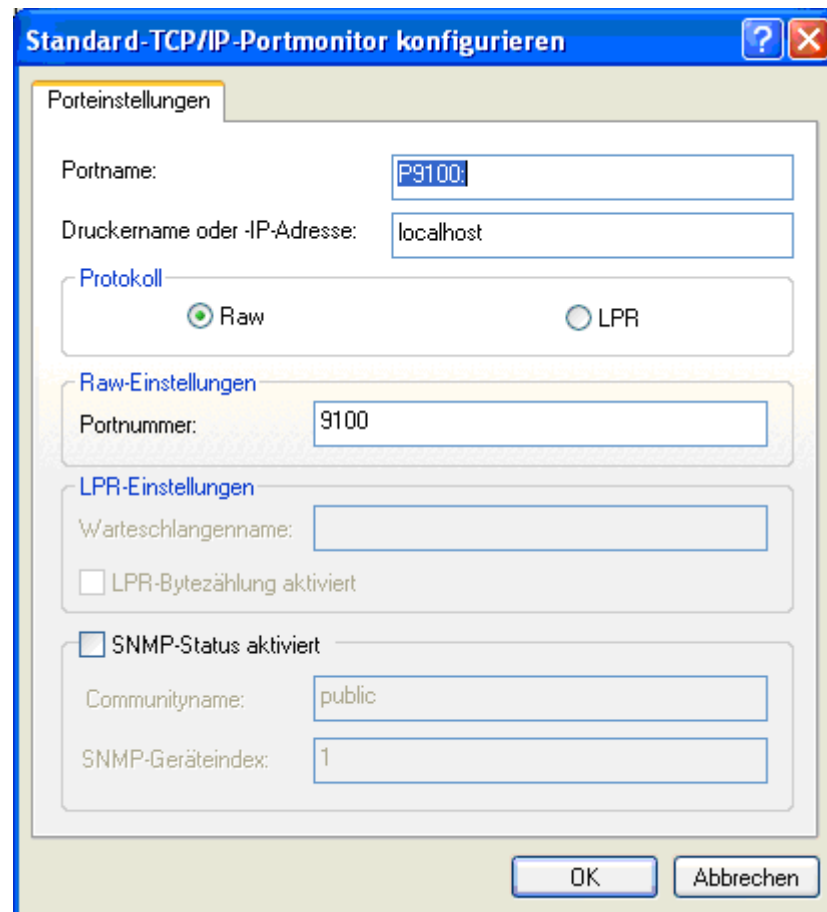
- By “Printer name or IP Address” enter the name of the PC, for example, “localhost”.
- For “Port name” enter a useful name, such as “P9100”. This is the name of the connection (not the name of the printer).



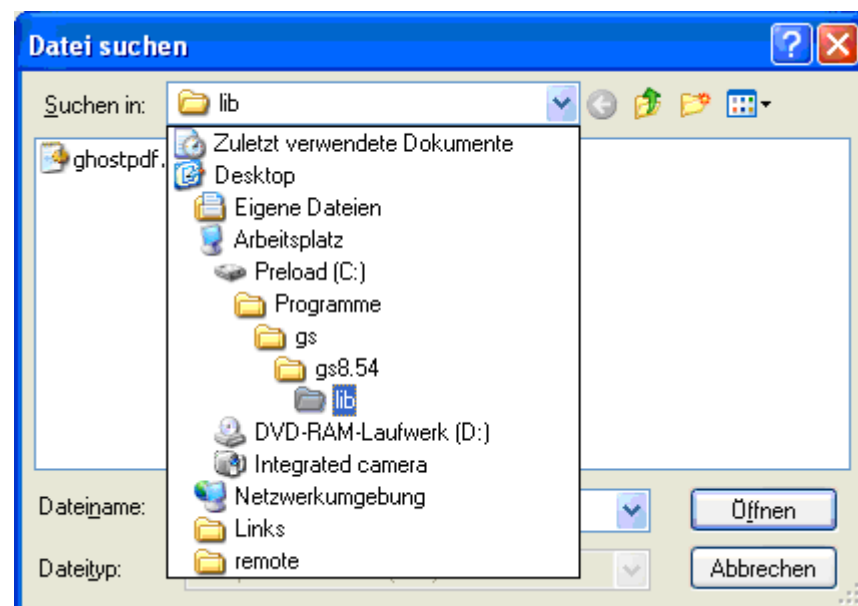
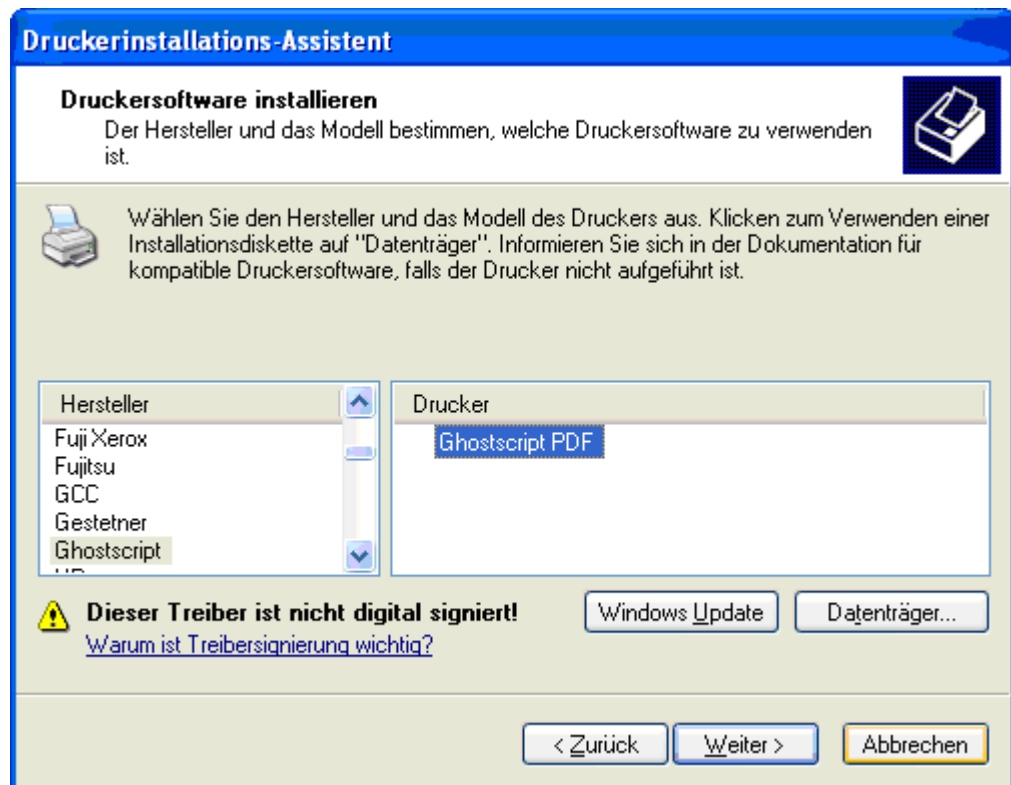
- Then wait a little and you will get the message “the printer is unknown or is not properly configured”.
- Then select for the “Device Type” “User Defined” and click *Settings*.

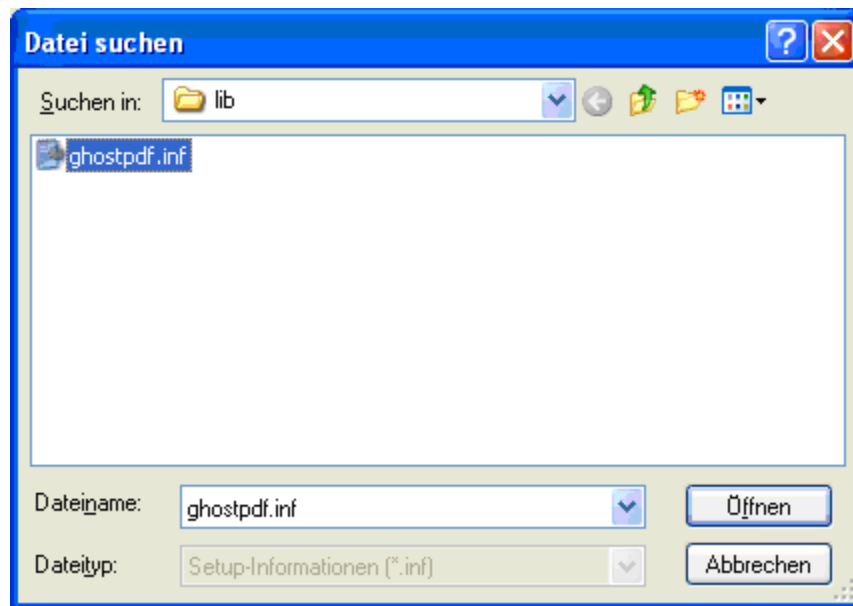


- The protocol should be set to “RAW” and the “Port number” to “9100” and “SNMP” not activated.

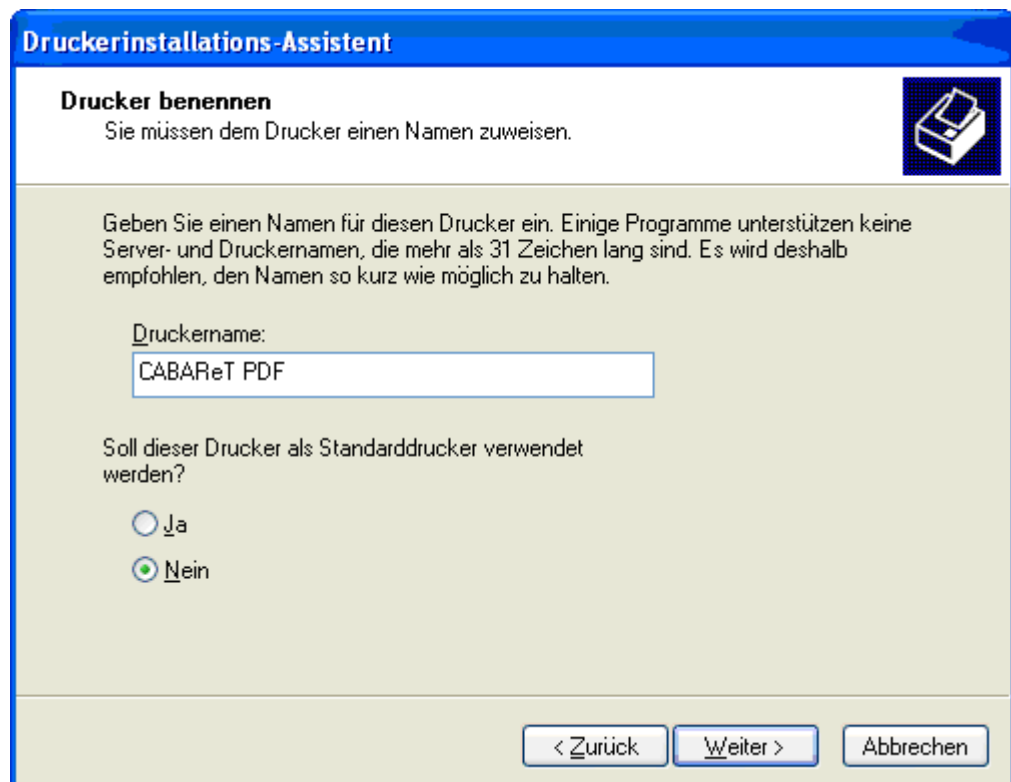


- Then please click OK and create the connection by clicking on "Finish".
- Finally, under "Printer Software" select the manufacturer "Ghostscript" and the printer "Ghostscript PDF". If the manufacturer "Ghostscript" and the printer "Ghostscript PDF" are not displayed, click on the button "Disk", and select the file *<Ghostscript install directory>\gs8.54\lib\ghostpdf.inf*





- If you are asked, if the available driver should be replaced, then it is generally better to select “Yes”. This will replace any older driver currently installed from previous Ghostscript installations with a newer driver.
- Then you will need to give the printer a name, for example, “PDF service”. The printer will be displayed under this name.



- Skip the rest of the dialogs.

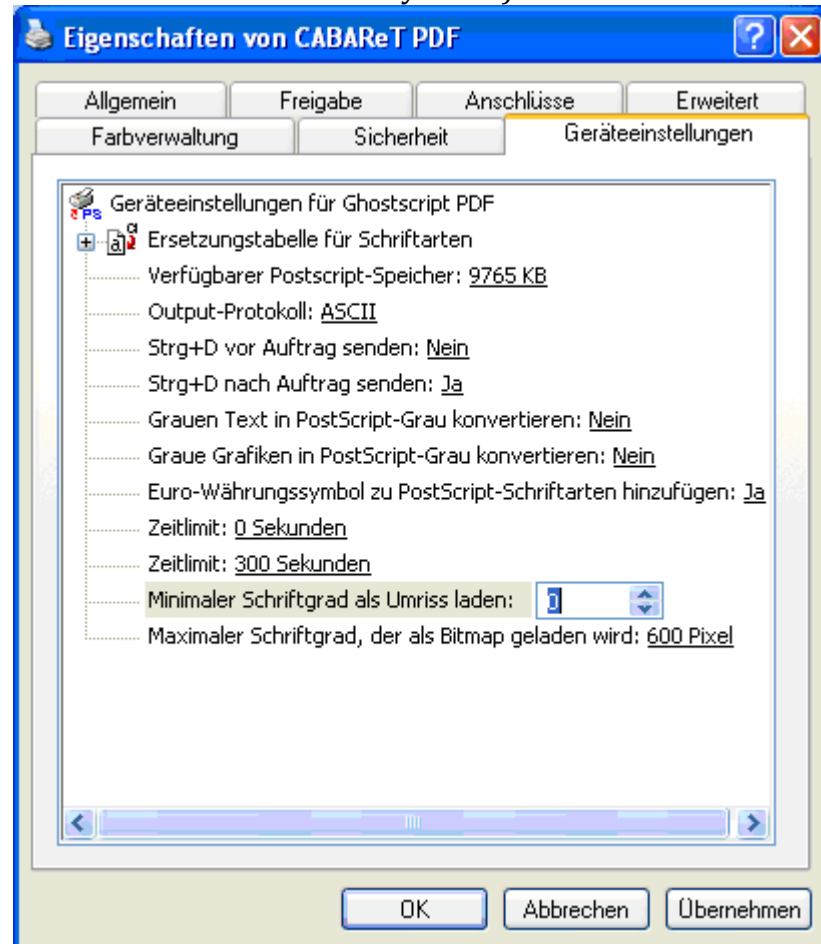
The local printer is then installed and will use Sign Live! CC for the print process over port 9100.

## 8.2.2.4 Further printer settings

Fonts should always be embedded in the PDF. This will require a one-time configuration of the printer.

Open the “Printer Settings” and select “Device Settings”.

- Under “Use minimal font size as model” set the value to “0” (after installation the value is usually “100”).



## 8.2.2.5 Instrument Settings

If Sign Live! CC and Ghostscript have been installed properly, no further settings will be needed. The Ghostscript Installation will be automatically found through the registry entries and used.

If Ghostscript is not found automatically, you can set it manually. This will require adjustments to “ExchangePS\instrument.xml”.



```

<extension point="com.cabaret.exchange.ps.ghostscript">
  <ghostscript>
    <dll path="C:/Programme/gs/gs8.54/bin/gsdll32.dll"/>
    <include path="C:/Programme/gs/gs8.54/lib"/>
    <include path="C:/Programme/gs/gs8.54/Resource"/>
    <include path="C:/Programme/gs/fonts"/>
    ...
  </ghostscript>
</extension>

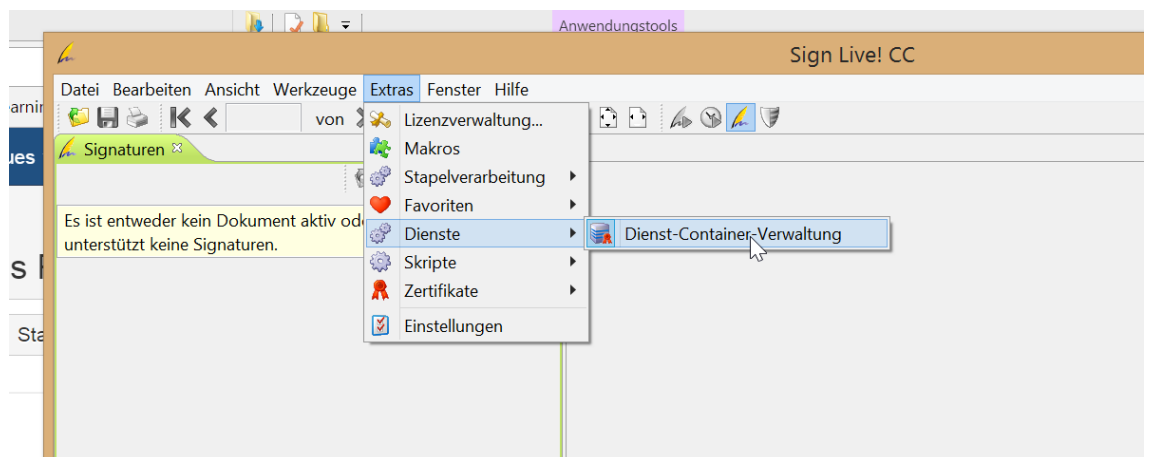
```

Filepaths here are always relative to the Sign Live! CC base directory (*baseDir*).

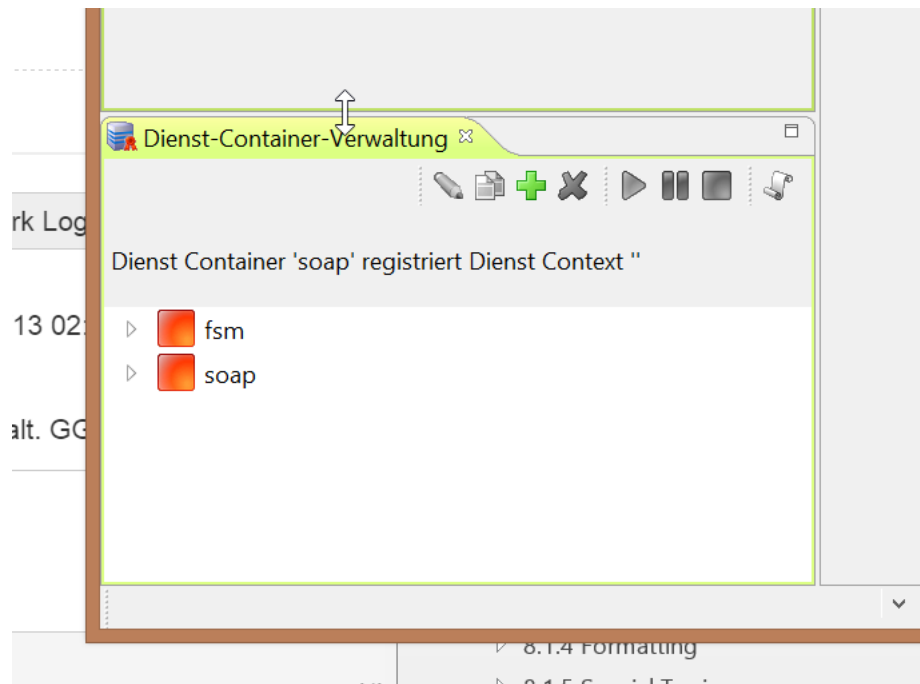
Furthermore, the call parameters for Ghostscript, in accordance with the Ghostscript documentation, can be edited here. The element “arg” is used for this purpose. All declared arguments will be passed to Ghostscript in the order they appear here.

#### 8.2.2.6 P9100 service container

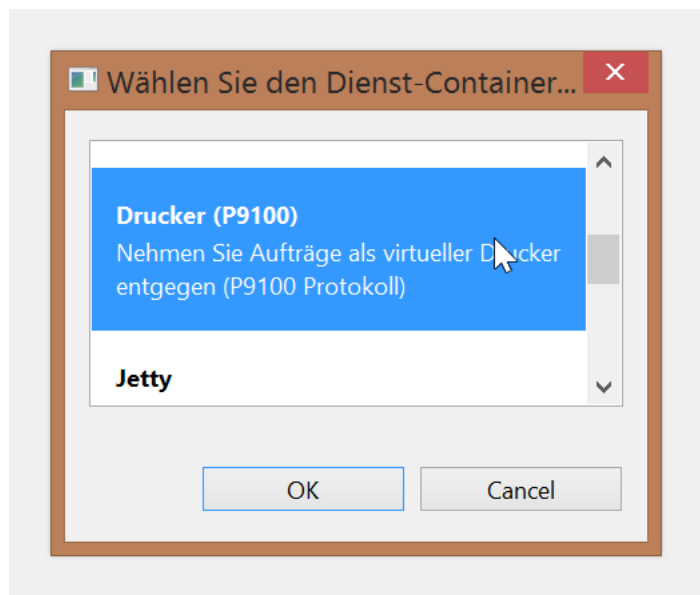
In the application you must define a P9100 service container. If the service container console is not already open, you can find it in the “Extras” Menu.



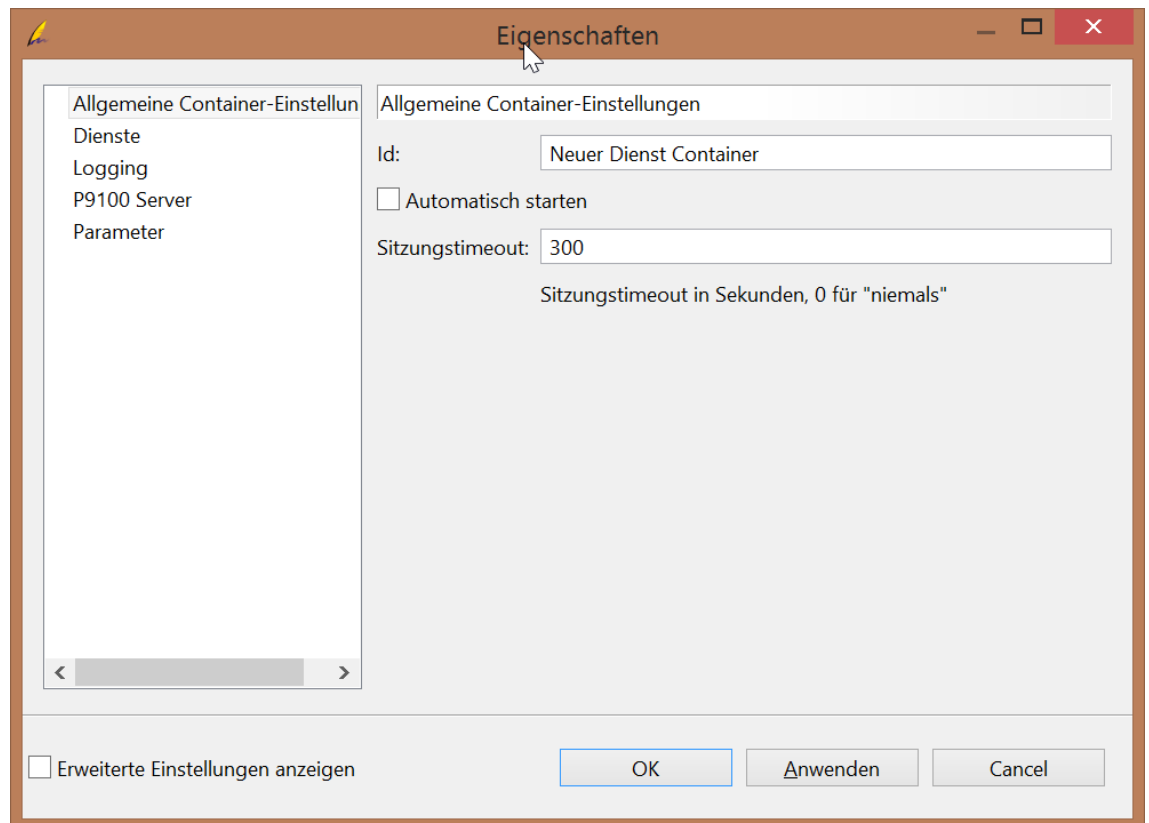
The service container console allows you to define a P9100 printer service.



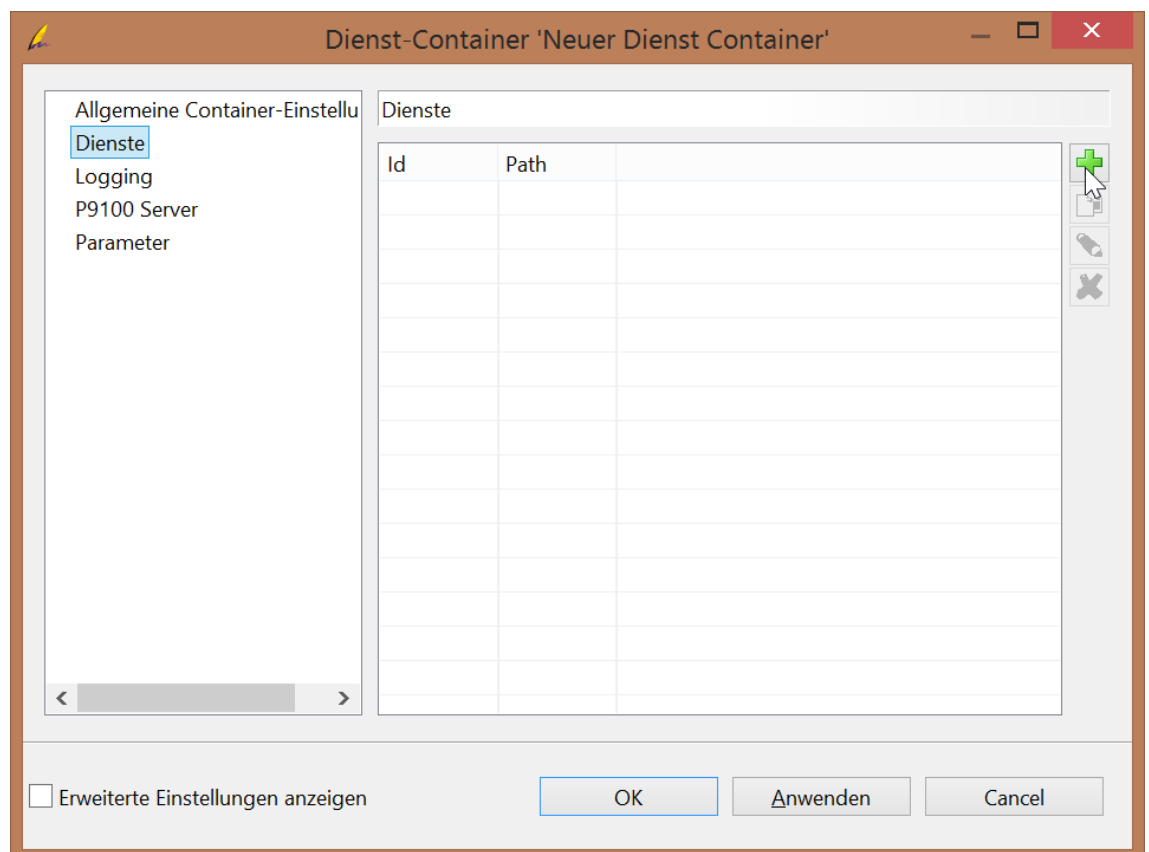
You press the plus-button to add a new service container and select “P9100”.



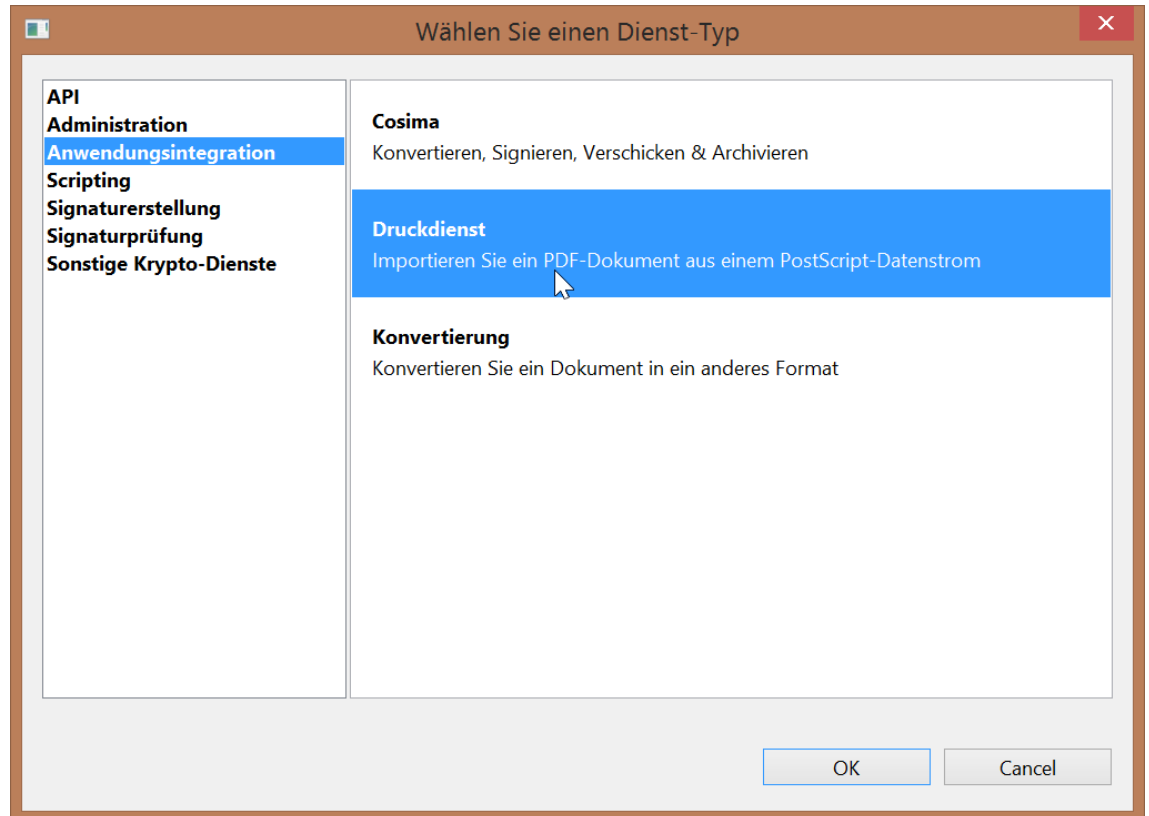
Now you can edit the properties of the container.



The most important setting is “Service” – what should the container do when a print stream is arriving. You add a service by selecting the “Service” tab and pressing the plus button.



You can select from a variety of services, the most simple is “Print Service”.



After selecting the “Print Service” you can change its settings. The most important ones are on the tab “Print service”, where you can select the target file or if the document should be viewed after importing.

That’s all.

