

Sign Live! CC Scripting Tutorial

Februar 2022

intarsys GmbH

Sign Live! CC Scripting Tutorial

Version 7.1

Scripting Sign Live! CC

intarsys GmbH
Sign Live! CC Scripting Tutorial
Version 7.1

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

Preface

- Author and company

This book has been provided by the development staff of intarsys GmbH

- Trademarks

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

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

EForm is a registered trademark of intarsys GmbH.

jPod is a trademark of intarsys GmbH.

Sun, Java and JavaScript are trademarks of Sun Microsystems

Microsoft and Windows are trademarks of Microsoft Corporation.

Adobe and Acrobat are trademarks of Adobe Systems Incorporated

- Who should read this book

This book is intended for Java developers intending to enhance Sign Live! CC or write a new application from scratch. You will gain a deep understanding of the how to's of the Sign Live! CC machinery.

If you simply want to add some buttons to a Sign Live! CC 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 Sign Live! CC or work with simple scripting features. You will find information thereto in the tutorials and the online help.

■ Organization

The basics will be provided in First Step. Here you will discover how a simple script can be created and tested. To top it off the script will be integrated into the Sign Live! CC interface.

The next section Working with Documents provides examples with access to documents within Sign Live! CC. This knowledge can be applied with all document types. After these basics the PDF-specific functions are of particular interest and will be discussed in PDF Documents.

The border to the next topic, Working with Processors, is fuzzy. Processors are the foundational building blocks for creating and processing documents, but also for other tasks unrelated to documents.

Java Integration covers the integration of standard Java APIs. This includes, for example, access to databanks and your own Java creations.

Process Automation and Integration will teach you to call Sign Live! CC functions or self-written scripts using the various platform interfaces.

After covering automation of and with Sign Live! CC “procedurally” we will discuss how to handle templates in JavaScript based Templates.

Several special applications for CodeExit will be introduced in Advanced CodeExit Applications. Here you will learn more about the possibilities for dynamic, script-based implementation of ExtensionPoints as well as several specific configuration possibilities.

■ 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 Sign Live! CC.

Now we come to the fun part, do a little programming - The Scripting Tutorial shows a fast path to customizing Sign Live! CC and some useful tips for scripting it.

A basis for the understanding of PDF is the PDF Reference from Adobe Systems.

The developer reference of the jPod intarsys PDF Library provides a complete overview of the basis APIs, on which Sign Live! CC is also based.

The Developer’s Guide is the book about programming Sign Live! CC. 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 Sign Live! CC application.

■ Reviews and comments

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

EMail

support@intarsys.de

Website

www.intarsys.de

■ Disclaimer

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

The information is provided "as is". The authors shall are in no way liable to any person or entity with respect to any loss or damages arising from the information contained in this book, or from the use of the disks or programs that may accompany it.

Contents

| | |
|-------------------------------------|----|
| Preface | 5 |
| ▪ Author and company | 5 |
| ▪ Trademarks | 5 |
| ▪ Who should read this book | 5 |
| ▪ Organization | 6 |
| ▪ Other documentation | 6 |
| ▪ Reviews and comments | 7 |
| ▪ Disclaimer | 7 |
| Contents | 9 |
| Introduction | 13 |
| 1. First Step | 15 |
| 1.1 Overview | 15 |
| 1.2 The First Script | 15 |
| 1.2.1 Script Manager | 15 |
| 1.2.2 Create Scripts | 18 |
| 1.2.3 The Next Step | 19 |
| 1.3 Interface Integration | 19 |
| 1.3.1 Overview | 19 |
| 1.3.2 The Instrument | 19 |
| 1.4 Icons and Toolbar | 22 |
| 1.5 Binding Scripts | 24 |
| 1.5.1 ScriptFile | 25 |
| 1.5.2 Script | 26 |
| 2. Working with Documents | 27 |
| 2.1 Overview | 27 |
| 2.2 Who am I | 27 |
| 2.3 Save and Close Document | 29 |
| 2.4 Load and Display Document | 29 |
| 2.5 Create and Display New Document | 30 |
| 2.6 Summary | 31 |
| 3. Working with PDF Documents | 33 |

Contents

| | | |
|--------|-------------------------------------|----|
| 3.1 | Overview | 33 |
| 3.2 | Document Properties | 33 |
| 3.3 | AcroForm | 35 |
| 3.4 | Embedding Data | 37 |
| 3.5 | Page Content | 39 |
| 3.5.1 | Symbols | 39 |
| 3.5.2 | Text | 41 |
| 3.5.3 | Tip | 42 |
| 3.6 | Debugging | 43 |
| 4. | Working with Processors | 45 |
| 4.1 | Overview | 45 |
| 4.2 | Basis Document Functions | 45 |
| 4.3 | Handle Documents | 45 |
| 4.4 | HTML to PDF | 46 |
| 4.5 | Attaching Pages | 47 |
| 4.6 | Delete Pages | 47 |
| 4.7 | Stamp Documents | 47 |
| 4.8 | Create form fields | 49 |
| 4.9 | “Flattening” Documents | 49 |
| 4.10 | Signing Documents | 50 |
| 4.10.1 | Overview | 50 |
| 4.10.2 | Simple Signature | 51 |
| 4.10.3 | Visible Signature | 51 |
| 4.10.4 | External (PKCS#7) Signature | 52 |
| 4.11 | Importing Images | 53 |
| 5. | Java Integration | 55 |
| 5.1 | Overview | 55 |
| 5.2 | Accessing the Java Log | 55 |
| 5.3 | Write File | 56 |
| 5.4 | Database Access | 57 |
| 5.5 | Shell Access | 58 |
| 6. | Process Automation and Integration | 59 |
| 6.1 | Overview | 59 |
| 6.2 | Commandline (CLI) | 59 |
| 6.3 | ULS | 61 |
| 6.3.1 | Overview | 61 |
| 6.4 | ULS - HTTP | 61 |
| 6.4.1 | Overview | 61 |
| 6.4.2 | CodeExit call per GET and POST | 62 |
| 6.4.3 | Generic CodeExit Call | 63 |
| 6.4.4 | CodeExit POST Call with Documents | 64 |
| 6.5 | ULS - File System Monitor | 67 |
| 6.5.1 | Overview | 67 |
| 6.5.2 | Configuration | 68 |
| 6.5.3 | CodeExit Call through an Entry File | 69 |
| 6.6 | ULS - P9100 | 71 |

| | | |
|-------|--|----|
| 6.6.1 | Overview | 71 |
| 6.6.2 | Printer Installation | 71 |
| 6.6.3 | Configuration | 77 |
| 6.6.4 | Convert and Open a Data Stream | 78 |
| 6.7 | Review ULS (J2EE Container) | 80 |
| 6.8 | Web Services | 81 |
| 6.8.1 | Overview | 81 |
| 6.9 | ActiveX | 81 |
| 6.9.1 | Overview | 81 |
| 6.9.2 | Declaration | 82 |
| 6.9.3 | C# Demo Container | 82 |
| 6.9.4 | Visual Basic Script Demo Container | 85 |
| 6.9.5 | Parameter Transfer | 85 |
| 6.9.6 | CodeExit with Event | 86 |
| 7. | JavaScript based Templates | 87 |
| 7.1 | Overview | 87 |
| 7.2 | The Script File | 87 |
| 7.3 | Hello, World | 87 |
| 7.4 | Dynamic Content - Calculations | 88 |
| 7.5 | Dynamic Content - Loops | 89 |
| 7.6 | Parameter | 90 |
| 7.7 | Generate HTML | 91 |
| 7.8 | PDF - for Advanced Users | 92 |
| 8. | Advanced CodeExit Applications | 95 |
| 8.1 | Overview | 95 |
| 8.2 | The Dynamic ExtensionPoint | 95 |
| 8.2.1 | Basis | 95 |
| 8.2.2 | Syntax | 95 |
| 8.2.3 | Multiple Interfaces | 96 |
| 8.2.4 | Important | 96 |
| 8.2.5 | Scripting Call Semantic | 97 |
| 8.3 | ExtensionPoint Configuration Information | 97 |

Introduction

The expandability of call Sign Live! CC is based mostly on scripting framework, which allows for simple integration of new functions into the system. This tutorial provides a 'step-for-step introduction' to the concepts and programming of Sign Live! CC.

This document is based on the release versions Sign Live! CC and jPod.

The scripting itself offers unlimited access to the system and its integration abilities - the possibilities are endless.

In the simplest application macro-like processes can be automated and included in your interface, for example, recurring processes such as "sign document, print and archive".

All script functions can also be called from external Sign Live! CC interfaces, for example,

- Commandline
- HTTP (Plain, XML RPC, SOAP)
- File System Monitor
- Mail (POP3)
- Remote Printer (P9100)
- Telnet (Stage/TN3270E)
- ActiveX
- SOAP based web services
- ...more to come...

This allows for simple integration and automation.

The "advanced school" then covers the real programming. This includes access to extensive libraries for working with PDFs or complex predefined building blocks such as document signatures or import/export document functions.

Introduction

If this is still not enough, the complete Java API is available thanks to “Stage/LiveConnect”. Integrate databanks through JDBC, third systems through web services or your own libraries.

1. First Step

1.1 Overview

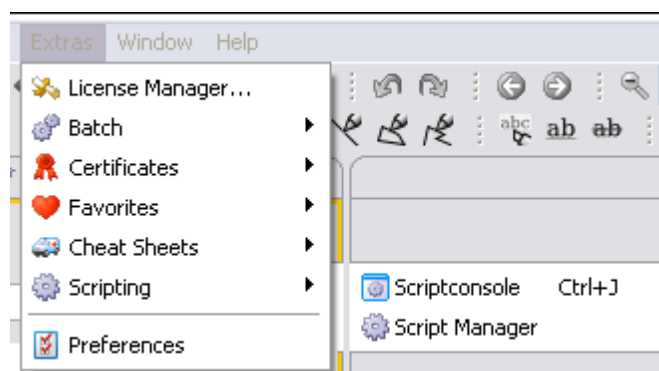
Expanding and customizing Sign Live! CC using scripting is easy to do even for new users. In this section basic concepts and techniques for working with scripting will be provided and introduced using comprehensive examples.

First a simple script will be created “ad hoc” and run within Sign Live! CC. After this has been accomplished we will integrate this function into the interface and optimize it.

1.2 The First Script

1.2.1 Script Manager

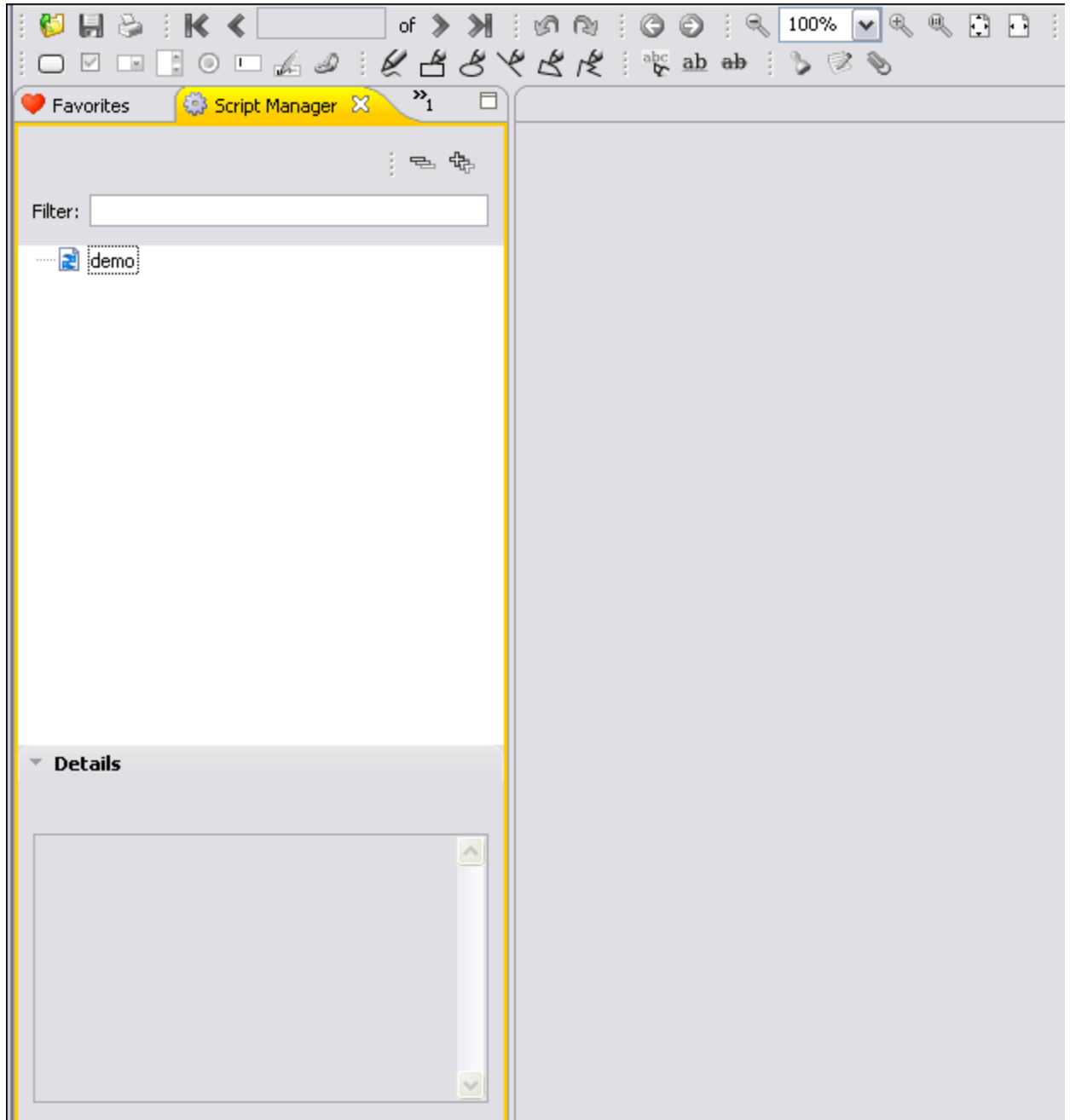
The simplest way to get started with scripts is through the Script Manager.



The script will display an Explorer-like view of the available scripts. You can manage and edit scripts here without needing an external tool. However, the Script Manager is not a full-scale development environment and is primarily intended for experimenting with development and testing integration into Sign Live! CC. For

management purposes you can define your own folder structure, in which links to directories and files in the file system will be created.

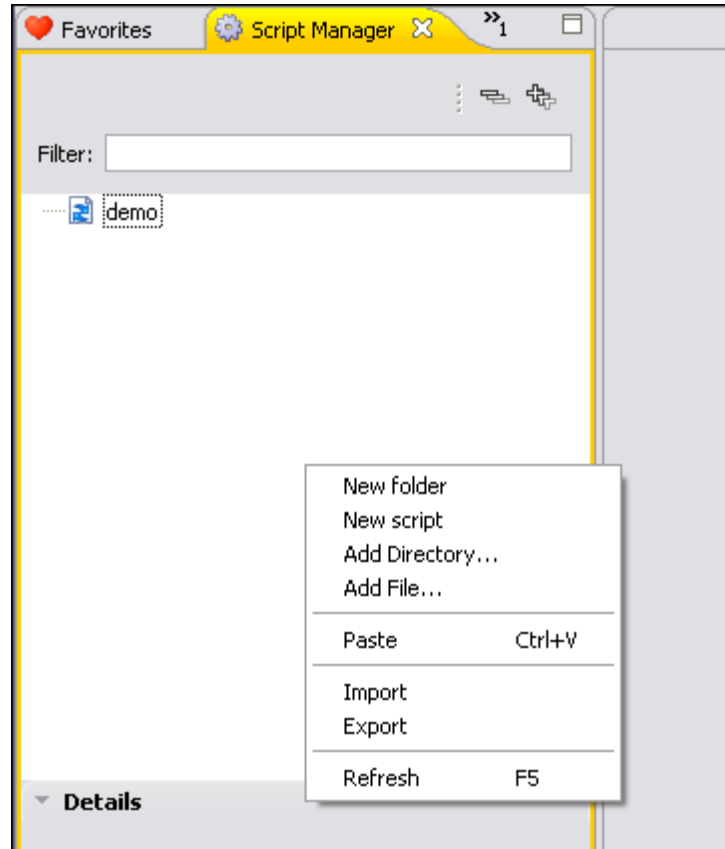
By default the Script Manager will start with a link to the Demo folder of the Sign Live! CC installation. Here you will find further scripting examples.



Create a new folder “TestScripts” using the context menu. This folder will not exist in the file system, but will be used to define your Favorites (Bookmarks) structure. You can define this structure anyway you wish so that your development environment best suites your needs. Changes to these management folders will never affect the file system. Should you, for example, delete the folder “TestScripts” from the Script

Manager at some later point, the files referenced within will not be deleted from the file system.

Of course you can also run operations in real folders (identifiable by the blue arrow in the icon) through the Script Manager. These will also be performed physically (Delete, Copy, Rename).



Create a new script in this folder with the name “helloworld.js” somewhere in your file system using the context menu. Do not forget the extension “.js”, the scripting framework uses this to identify the type of interpreter to use. The logical management folder “TestScripts” now includes a reference to the file “helloworld.js” in your file system.

The new file will be opened as an empty window. Remember that the Script Manager is intended as a development and debugging helper and does not provide a full-scale programming editor.



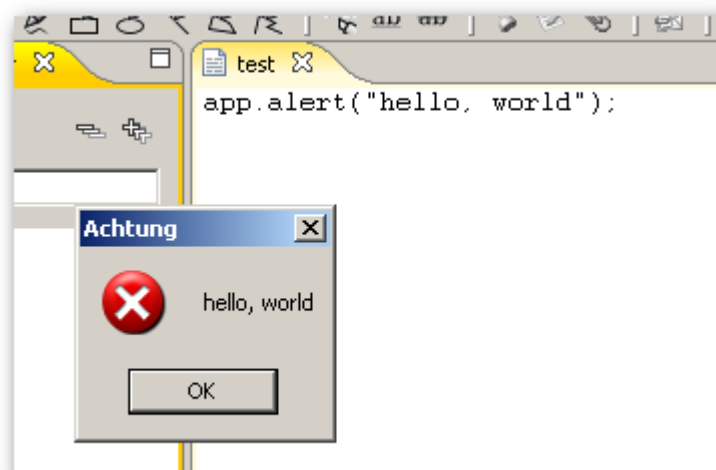
In general you can work with the editor the same way you would work with any other document in call Sign Live! CC. You can control it using the Sign Live! CC menu.

1.2.2 Create Scripts

You can now enter the source code for your script into the editor and save it. The following example is pretty much the grandfather of all demo programs:

```
app.alert("hello, world");
```

Don't forget to save. The script can now be started in the Script Manager per double click (or through the context menu).



1.2.3 The Next Step

After the first successful script there are many other possibilities to be discovered for Sign Live! CC:

- Integrating your own functions into the interface through the Menu or Toolbar
- Controlling functions using one of the many external interfaces such as Commandline, ActiveDoc,... for integration in your processes.
- Use of libraries and function building blocks to edit and control documents
- Unrestricted integration of Java Code.

These subjects will be covered in detail in later sections of this tutorial. For now we will continue to work with our previous example in the Sign Live! CC interface.

1.3 Interface Integration

1.3.1 Overview

Of course it is possible to customize and integrate Sign Live! CC by offering scripts in a logical structure within the Script Manager. But we can do better than that by integrating these changes into the Sign Live! CC interface. This has several advantages:

- Professional Look & Feel
- Context-sensitive control
- Packaging and delivery in the standard Sign Live! CC format as an Instrument

1.3.2 The Instrument

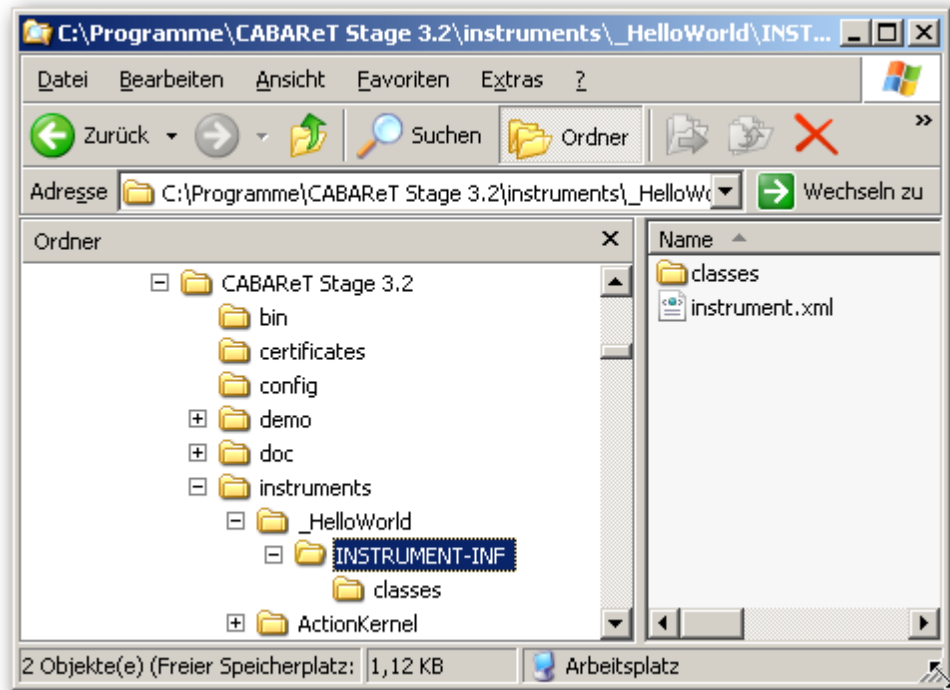
Integration is easiest using an 'Instrument', the component unit for Sign Live! CC.

In this step we will develop an Instrument that creates a menu entry in the interface, with which we can run our example code from the previous step. In the *demo* directory of Sign Live! CC there is a predefined Instrument "_HelloWorld", which already includes the source code for the following example. To quick start, copy it into the *instruments* directory of the installation.

An Instrument is always constructed as follows:

- A directory with an individual name in the *instruments* directory of the Sign Live! CC installation. This is the Instrument's *Home* directory.

- A directory *INSTRUMENT-INF* in the *Home* directory of the Instrument. The meta data for the operation of the instrument in Sign Live! CC is stored here.
- The file *instrument.xml* in the directory *INSTRUMENT-INF*. The declarations for expanding the platform are stored here.



- Create a new directory for your Instrument in the *instruments* directory of the Sign Live! CC installation as described above. In the next steps we will create a new file “instrument.xml” in the *INSTRUMENT-INF* directory.
- This is the complete content of the file that we will describe step-for-step below:

```

<instrument
  id="my.company.HelloWorld">

  <requires>
    <prerequisite
instrument="com.cabaret.scripting.javascript.application.pdf"/>
    </requires>

  <extension point="com.cabaret.claptz.action.actions">
    <action
      id="my.company.HelloWorldAction"
      label="Hello, World">
      <effect>
        <perform type="JavaScript" source='app.alert("Hello,
World")' />
      </effect>
    </action>
  </extension>

  <extension point="com.cabaret.claptz.widget.widgets">

    <widget parent="com.cabaret.widget.menubar.tools/additions">
      <on event="select" action="my.company.HelloWorldAction"/>
    </widget>

  </extension>
</instrument>

```

This Instrument uses predefined extension points (the element *extension*), to declare several extensions for the application. Two extension points were used here for the declaration of an action and the declaration of the interface element that will run this action.

An action defines an operation and its presentation in the interface.

The following XML syntax is used for the definition:

```

<action
  id="my.company.HelloWorldAction"
  label="Hello, World">
  <effect>
    <perform type="Script" source='helloworld' />
  </effect>
</action>

```

The *action* element requires a unique *id* - we recommend always using a prefix - and a *label*. There are further attributes that are described later in this documentation.

The greatest component of the *action* is the sub-element *effect*. This describes the function the action will trigger. The description is set with help from a “CodeExit”, as in many other places within Sign Live! CC. Further information hereto will also follow later in this documentation. To put it shortly the *CodeExit* is Sign Live! CC’s way of engineering a

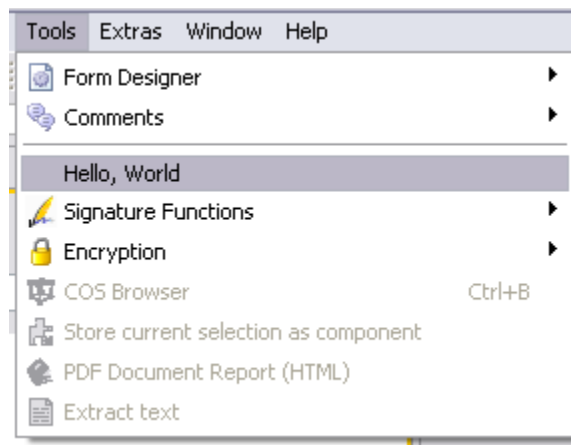
functional extension. It is defined with help from the XML element *perform* and the attributes *type* and *source*. Here *type* takes the role of the *.js* extension from our first script example - the scripting framework must know how to interpret the content of *source*. In this case *type* is JavaScript, i.e. we can enter JavaScript Code directly in “source”. This code is identical to the code we used in our first example for the Script Manager.

After defining the *action* we will still need a link with the interface in order to call the action. This will require that we define an interface element (*widget*).

```
<widget parent="com.cabaret.widget.menubar.tools/additions">
  <on event="select" action="my.company.HelloWorldAction"/>
</widget>
```

The *parent* attribute defines the location in the Sign Live! CC interface where the new interface element will be placed. The available locations are described in detail in further documentation. Here we will include the element in the menu “Tools”. The element *on* describes the action to be taken when the menu is activated. “id” links the defined action.

And this is the result when you save this code as *instrument.xml* in your *INSTRUMENT-INF* directory, exit and then restart Sign Live! CC:



Activating it will result in the message box being displayed, as in the previous example.

1.4 Icons and Toolbar

Of course you can decorate your action with an icon and include it in the toolbar.

Icons are simple: add the attribute “icon” to your *action* definition.

```
<action
  id="my.company.HelloWorldAction"
  label="Hello, World"
  icon="world">
  <effect>
    <perform type="JavaScript" source='app.alert("Hello, World")' />
  </effect>
</action>
```

Sign Live! CC will search for the required resources through the Java Classpath. The simplest and most modular way to store these resources is to create a directory “classes” in your instruments “INSTRUMENT-INF” - Sign Live! CC will look here for resources for your Instrument. Copy an icon file with the name you used into this directory.

Now lets add an interface element in the toolbar for your Instrument. The following code fragment will need to be added to the existing widget.

```
<widget parent="com.cabaret.widget.toolbar">
  <widget>
    <on event="select" action="my.company.HelloWorldAction"/>
  </widget>
</widget>
```

Here is the code including the icon and interface element in the toolbar.

```

<instrument
  id="my.company.HelloWorld">

  <requires>
    <prerequisite
      instrument=
"com.cabaret.scripting.javascript.application.pdf"
    />
  </requires>

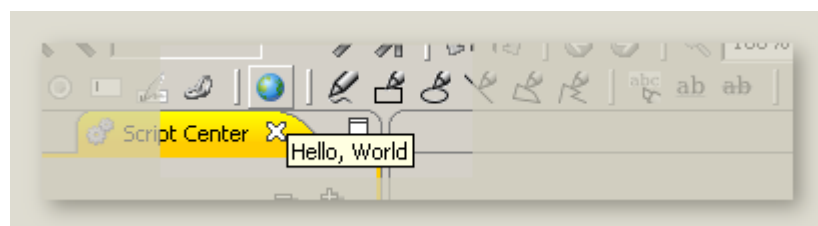
  <extension point="com.cabaret.claptz.action.actions">
    <action
      id="my.company.HelloWorldAction"
      label="Hello, World"
      icon="world">
      <effect>
        <perform type="JavaScript" source='app.alert("Hello,
World")' />
      </effect>
    </action>
  </extension>

  <extension point="com.cabaret.claptz.widget.widgets">
    <widget parent="com.cabaret.widget.menubar.tools/additions">
      <on event="select" action="my.company.HelloWorldAction"/>
    </widget>
    <widget parent="com.cabaret.widget.toolbar">
      <widget>
        <on event="select" action="my.company.HelloWorldAction"/>
      </widget>
    </widget>
  </extension>

</instrument>

```

Since there are no predefined “empty” groups in the toolbar to add an icon to, the definition is nested here - a new group in the toolbar and an icon to be placed in it to activate the action. The result is Sign Live! CC will display an additional icon in the toolbar after restarting.



1.5 Binding Scripts

In this example direct JavaScript Code was used to implement the CodeExit


```
<perform type="JavaScript" source='app.alert("hello, world")' />
```

. This is a powerful way to quickly create expansions or dynamic scripting independent of configuration through external call-up variations such as commandline, HTTP and ActiveDoc.

There are even better customization options using Instruments. Literal JavaScript Code has the following problems here:

- Changes require a new start
- Longer code fragments become unreadable in literal form
- Formatting rules from the XML structure are often missed and lead to errors that can be hard to debug. This includes, for example, the use of quotation marks, special characters (such as "<" and ">") or whitespace (single line comments!!).

Better options are provided by the CodeExit types *Script* and *ScriptFile*. These types introduce an indirection - a file will be searched for or read with the help of a name. The implementation language is determined, as in the first example with the Script Manager, by the file extension.

The main advantages:

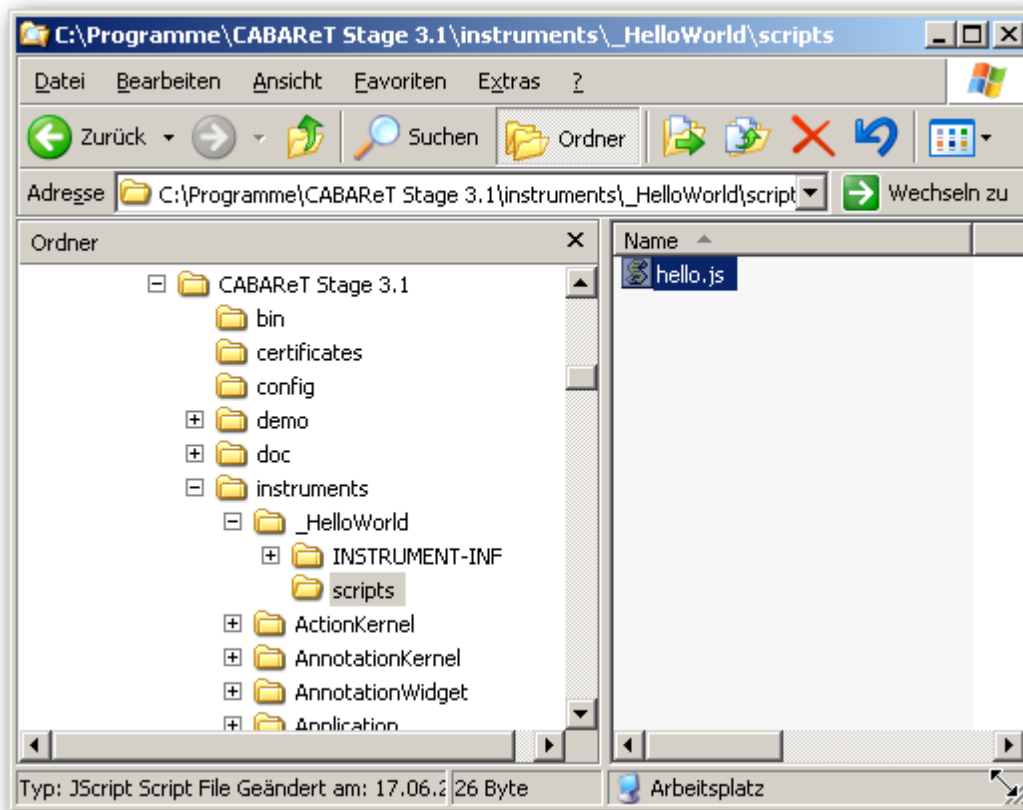
- Hot Code Deployment - the scripting framework will detect changes to the file upon referencing and reload it.
- Clarity and maintainability

1.5.1 ScriptFile

The type "ScriptFile" references a file directly. If you have saved, for example, the first test script in the Script Manager under "c:\temp\helloworld.js", you can call it from your Instrument declaration with

```
<perform type="ScriptFile" source="c:/temp/helloworld.js" />
```

Now to completely round-out your Instrument the script should be included in the Instrument folder - this way it can be easily and completely delivered and installed. Additionally, replacements can be made in the declaration string (complete information about string replacement and the available variables can be found in further documentation).



Create a directory “scripts” in your Instrument directory (not in *INSTRUMENT-INF*, this is no longer meta data) and copy the test script to this directory. Adjust the declarations as follows:

```
<perform type="ScriptFile" source="scripts/helloworld.js"/>
```

The script will be automatically searched for in relation to the Instrument directory and be delivered easily.

1.5.2 Script

A further flexible alternative is the type “Script”. This will cause the scripting framework to search for an implementation with this name in all search directories (a definition of search directory is provided in further documentation) and, of course, in the Instrument directory. In this case there is no extension as the scripting framework will use the first supported implementation that it finds. Other script languages such as Python or Groovy are generally supported and can be implemented.

```
<perform type="Script" source="scripts/helloworld"/>
```

2. Working with Documents

2.1 Overview

In this section we will cover accessing documents within Sign Live! CC scripting. The easiest case is working with documents that have been opened in Sign Live! CC to be further processed.

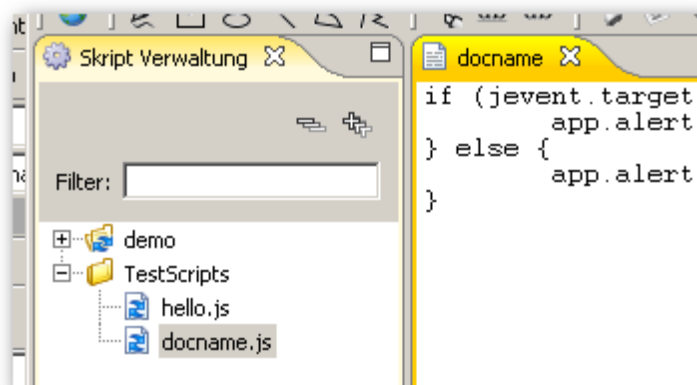
In order to open or create documents Sign Live! CC function building blocks will be needed that won't be formally introduced until the next section: "processors".

2.2 Who am I

Whenever a script is called from Sign Live! CC a parameter "jEvent" will be delivered to the script. Using this parameter the script can access the context of the call and react to it. For example, the application's active document can be determined in order to be processed in the script!

Since it would take too long to integrate every function we wish to discuss here into the interface we will continue to work in the Script Manager.

Create a new script "docname.js" in your directory from the first programming example.



With this script we want to display the name of the currently opened document with help from the “jEvent”. *jEvent* will provide us with the following additional information: the window (more precisely: the processor) that is currently active is entered in the attribute *target*. Furthermore, the window displays the document that is being processed in that window in the attribute *document*. The document then provides information such as:

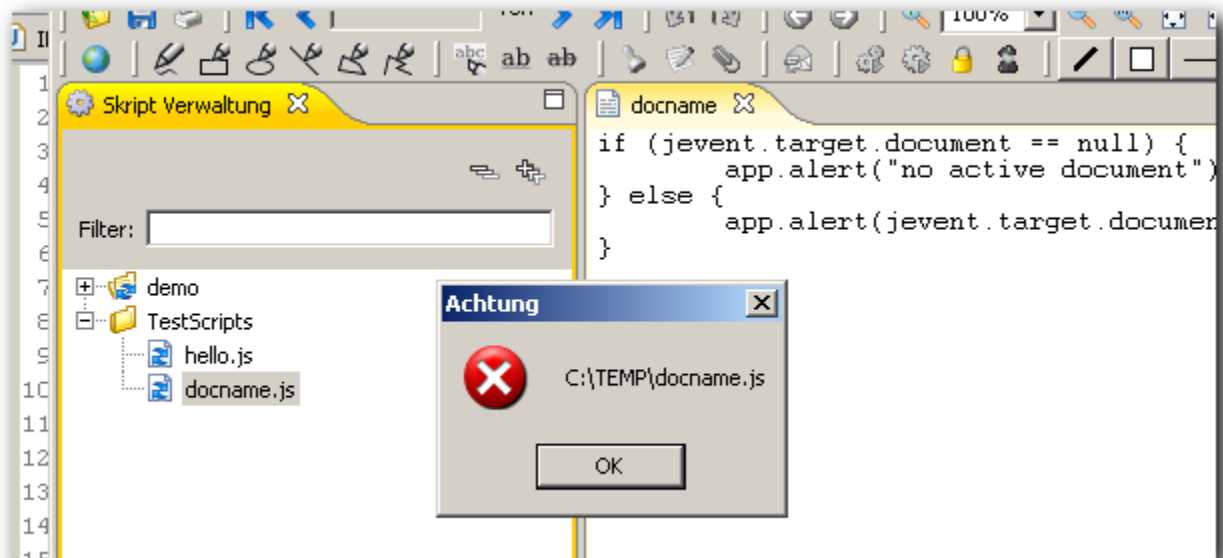
- `longName`
The name of the document in long form
- `shortName`
The name of the document in short form

and much more to those who know what to look for. Complete information hereto can be found in the reference.

The expression `jEvent.target.document.longName` provides the name of the currently active document! With a small security check in case no documents are open this leaves us with, for example, the code:

```
if (jEvent.target == null) {  
    app.alert("no active document");  
} else {  
    app.alert(jEvent.target.document.longName);  
}
```

Enter this code and save it. Now if you run the script (without closing the text editor) in the Script Manager you should see the following:



If you run the code while you have a PDF document open you will, of course, see the name of this PDF document. Play around with this script some!

2.3 Save and Close Document

In this section we want to save the document in front of us and close it. The following code section accomplishes this. We will analyse it shortly:

```
if (jEvent.target == null) {  
    app.alert("no active document");  
} else {  
    var viewer = jEvent.target;  
    var idoc = viewer.document;  
    idoc.save("c:\\temp\\foo.pdf");  
    viewer.stop();  
}
```

In addition to the information described above, such as “longName”, the document also offers methods that you can call up, for example, “save”. This method triggers the saving of the document in question. A parameter can be added to define the save location. If no parameter is given, the document will be saved to the file it was read from. Note the details here. For example, “\” in JavaScript must have the so-called “Escape” symbol “\” added to it, which leaves us with “\\”. A simple “\t” will be interpreted as a “tab” symbol.

Closing the document display is not a document method, but a “Viewer” method instead. This is the case because technically multiple “Viewers” or other “processors” can be working with the same document simultaneously. This is why the “Viewer”, and not the document, must be closed. Accessing this viewer is easy - “jEvent.target” will always include the processing context from Sign Live! CC, which, in this case, is the the viewer itself. A simple *stop* will close it.

Create a new PDF document in Sign Live! CC or open an existing one. Afterwards, open your script *saveclose* by double clicking on it. The document has “disappeared”, but was first saved under “c:\temp\foo.pdf”.

2.4 Load and Display Document

In order to load a document Sign Live! CC needs the type for the new document to be created. The available types can be found in further documentation. For our example we want to work with a PDF document whose type has been registered as *com.cabaret.document.pdf.PDFDocumentType*.

```
var outlet =  
Packages.com.cabaret.claptz.common.document.DocumentOutlet.get();  
var idotype =  
outlet.lookupDocumentType("com.cabaret.document.pdf.PDFDocumentType");  
var idoc = idotype.createFromLocator("c:\\temp\\foo.pdf", null);  
if (idoc != null) {  
    idoc.view();  
    idoc.release();  
}
```

or shorter:

```
var idoc =  
Packages.com.cabaret.claptz.common.document.DocumentTools.load(  
    "c:\\temp\\foo.pdf"  
);  
if (idoc != null) {  
    idoc.view();  
    idoc.release();  
}
```

Loading a document introduces something new: documents are “counted” upon use. This means that a document will eventually need to be re-released after being loaded or newly created. The document will not disappear until no one is still using the document.

2.5 Create and Display New Document

```
var outlet =  
Packages.com.cabaret.claptz.common.document.DocumentOutlet.get();  
var idotype =  
outlet.lookupDocumentType("com.cabaret.document.pdf.PDFDocumentType");  
var args = Packages.de.intarsys.tools.functor.Args.createNamed();  
args.put("pageCount", 7);  
var idoc = idotype.createNew(args);  
if (idoc != null) {  
    idoc.view();  
    idoc.release();  
}
```

Sign Live! CC supports many document types - when writing scripts you yourself are using text documents. Further examples are image formats and HTML documents. This is why when creating a new document the document type must be provided. This is accomplished through the registry of available formats. Here we will find the document format “com.cabaret.document.pdf.PDFDocumentType” and create a new document of this type with *createNew*.

Even after creating a document a “release” will be required after processing the document!

2.6 Summary

At this point we have covered everything useful to know about documents themselves in Sign Live! CC.

Things will start to get really interesting in the following chapters. We can now work with documents. The next sections will build on this, including working with Document Contents and Processors.

3. Working with PDF Documents

3.1 Overview

Finally, we get to the central point of interest for most Sign Live! CC users - working with PDF documents. Sign Live! CC's PDF functions are based on a high-performance PDF library - it is used to implement all internal Sign Live! CC functions.

In this section of the tutorial we will provide a rough overview of this library's possibilities and components and an introduction to more complex functions, such as "Sign Document", which are implemented and available with help from processors.

The PDF API is extremely extensive and the PDF document format very complex. This tutorial will only provide basis information. We highly recommend further literature:

- jPod intarsys PDF Library Reference. This includes more programming examples.
- Adobe PDF Reference

The examples have been kept as simple as possible in order to concentrate on the API being presented. You can use examples from later sections, especially Java Direct Access, to develop powerful little helpers. A practical example would be exporting form contents into your own file structure or company databank. Simply script!

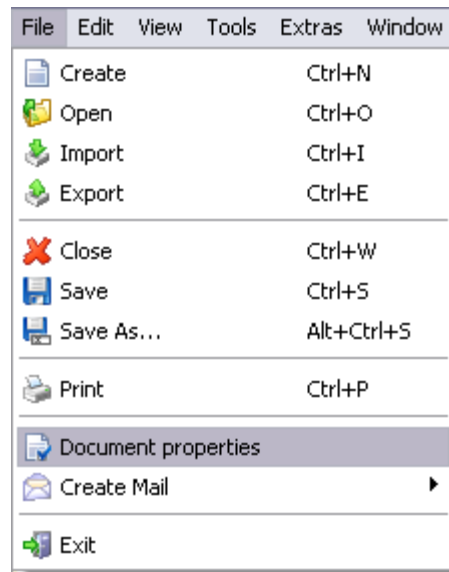
3.2 Document Properties

A PDF document has so-called meta data (data, that describe the document itself) that you can access with help from Sign Live! CC. The following data can be stored in the PDF in the so-called *InfoDict*:

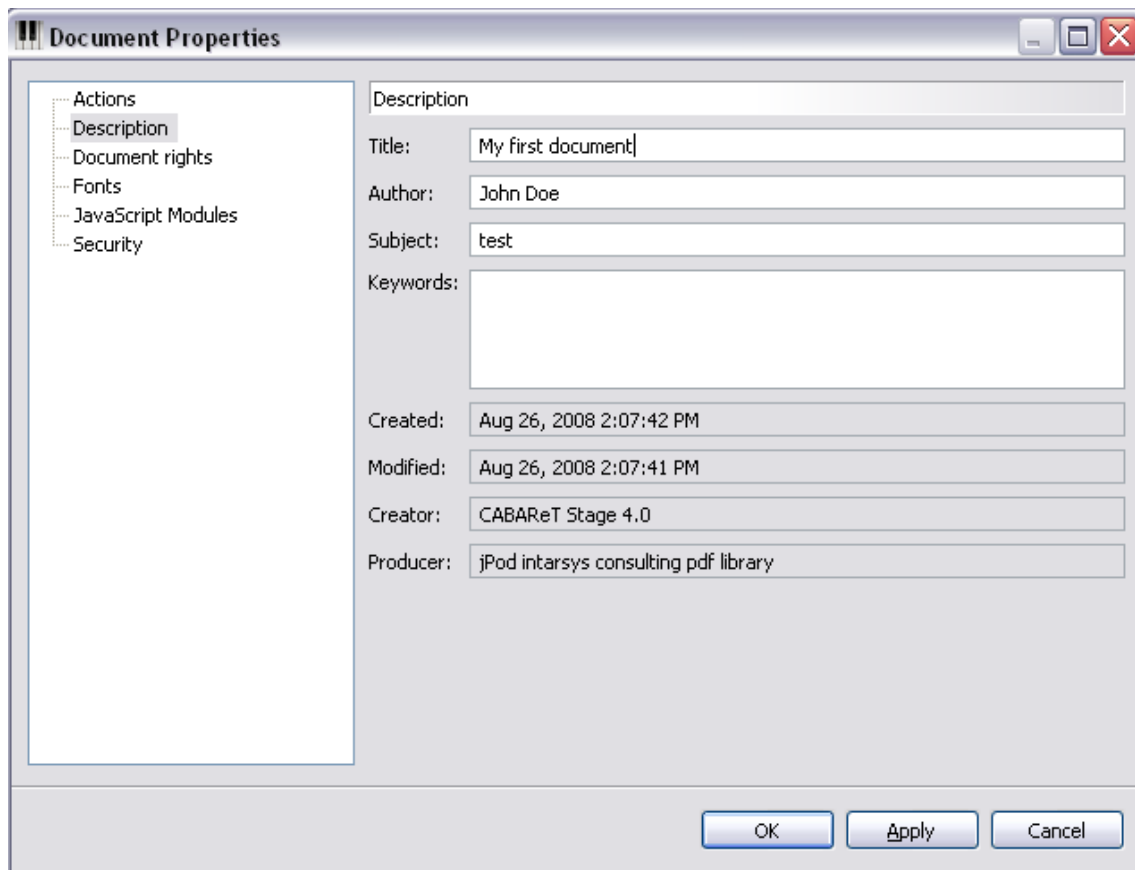
- Title
- Author
- Subject

- Keywords
- Creator
- Producer
- CreationDate
- ModeDate
- Trapped

In the application you can find this meta data in the file menu:



The dialog for the document properties will be displayed



You can use a script to read or write this information.

```
var pddoc = jEvent.target.document.impl;
//
var author = pddoc.author;
var subject = pddoc.subject;
//
app.alert("Von " + author + " mit dem Thema " + subject);
//
pddoc.title = "Hello, Property";
```

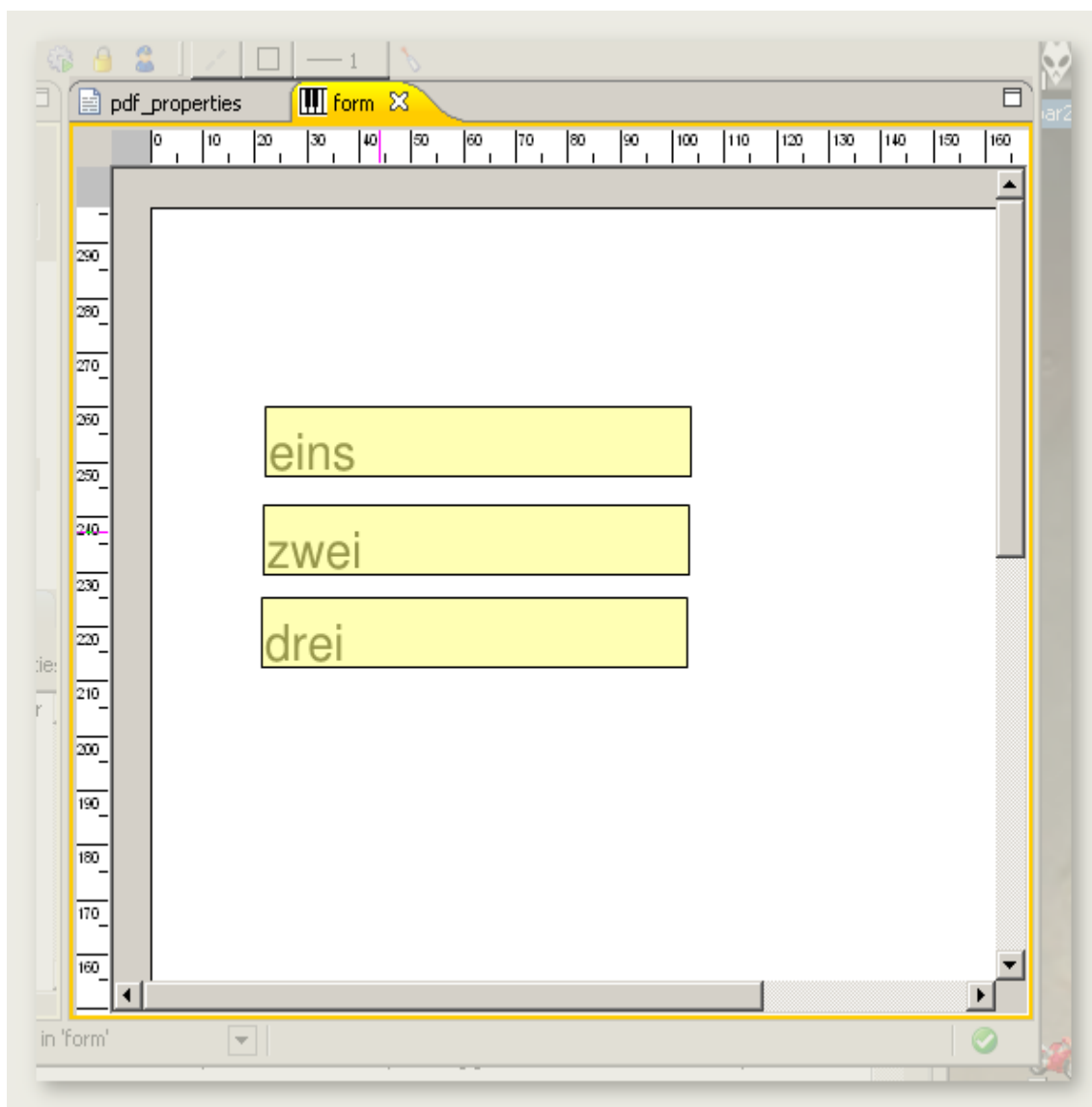
Simply compare your values with those in the document properties dialog.

A PDF document has further structures that allow meta data to be stored. This includes, for example, XMP data structures or *PieceInfo* structures. Generally speaking you can access all of this data. Further information hereto can be found in the PDF API Reference.

3.3 AcroForm

The AcroForm is the data structure within the PDF document that defines a form. You can read and write this data using the PDF library. This allows you to do anything you want - from automatically completing forms to integration of databanks into your operative system.

For this example you will need to create a new form using Sign Live! CC or use an existing form (in which case you will need to adjust the field names). In case of problems, you can find information about creating forms in the online help.



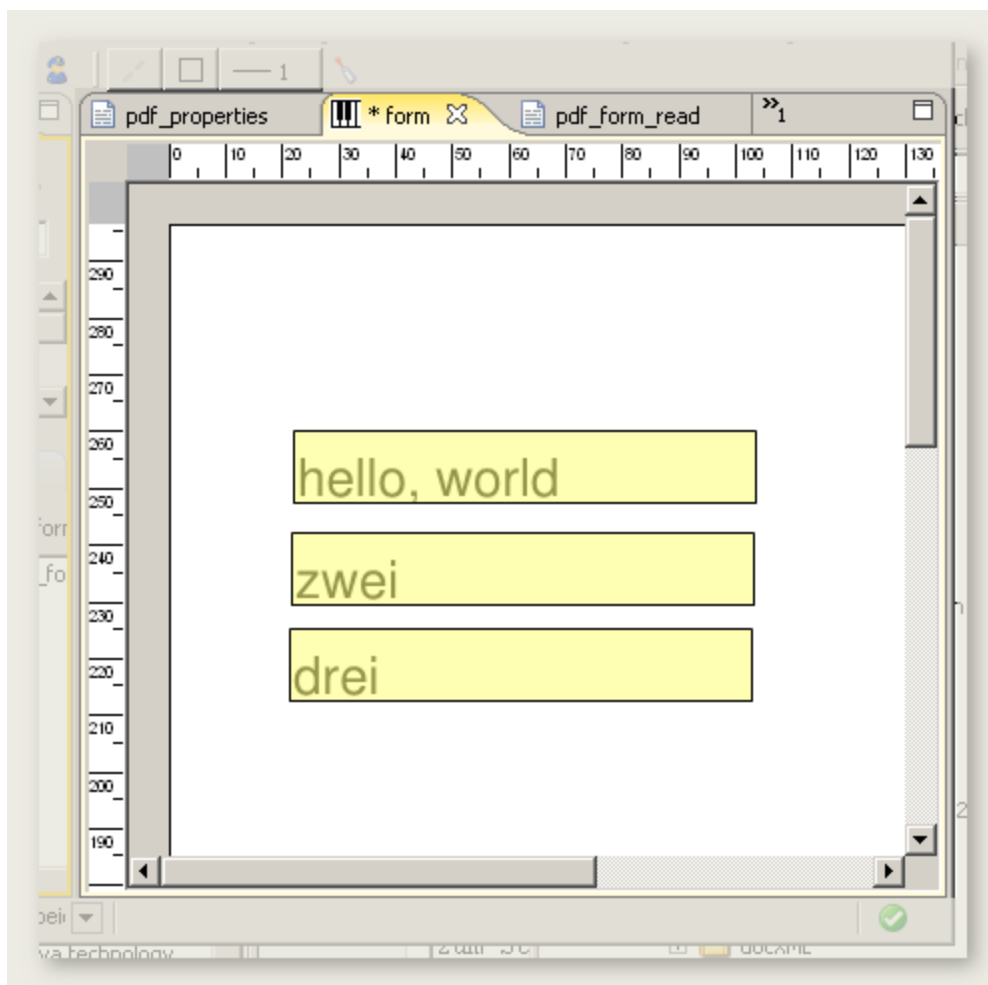
The fields here are named *Textfeld1*, *Textfeld2* and *Textfeld3*. The following code will display the contents of all the fields in order.

```
var pddoc = jEvent.target.document.impl;
var pdform = pddoc.acroForm;
var pdfields = pdform.collectLeafFields().toArray();
//
for (i in pdfields) {
    var pdfield = pdfields[i];
    app.alert(pdfield.localName + " = " + pdfield.valueString);
}
```

Of course you can enter values into this form. The help object “Formhandler” is available to help accomplish this. This object

simplifies the relatively complex steps that are needed to enter values into form fields (the hardheaded are, of course, welcome to control the PDF APIs directly)

```
var pddoc = jEvent.target.document.impl;  
//  
var factory =  
Packages.de.intarsys.pdf.app.acroform.FormHandlerFactory.get();  
var handler = factory.createFormHandler(pddoc, null);  
//  
handler.setFieldValue("Textfeld1", "hello, world");
```



3.4 Embedding Data

Proprietary data can be embedded into a PDF document in several different ways. This data will not be interpreted by PDF applications and can be, for example, billing data, workflow information and much more. This example will show a simple way to work with embedded data.

First, create or open a document, in which you wish to embed the data.

```

var pddoc = jEvent.target.document.impl;

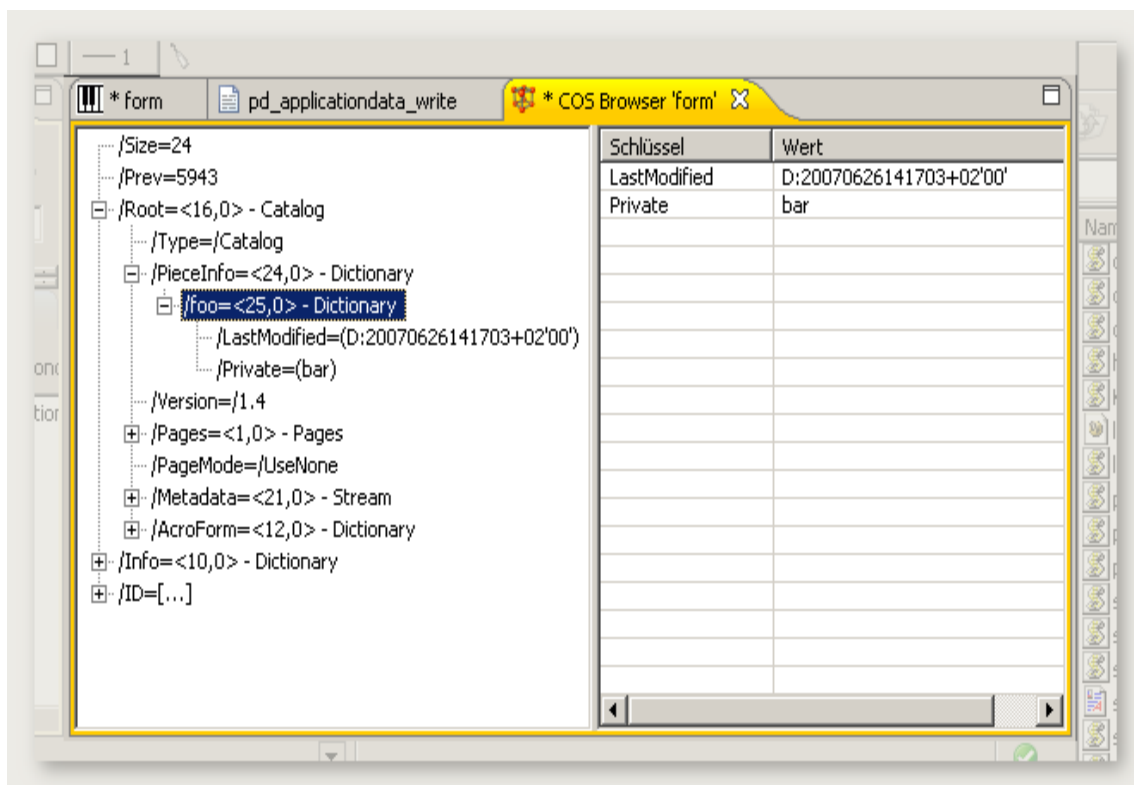
/*
 * first, create a PDApplicationData object containing, for example, a
 * string
 */
var appData =
Packages.de.intarsys.pdf.pd.PDApplicationData.META.createNew();
var data = Packages.de.intarsys.pdf.cos.COSString.create("bar");
appData.cosSetData(data);

/*
 * now store it in the document with the name /foo
 */
pddoc.setApplicationData("foo", appData);

/*
 * now, look it up in the COS browser...
 */

```

This code will create a PDF conform “embedded object” and save it under the name *foo* in the document. The data structure can now be analysed in the COSBrowser included with Sign Live! CC.



The recipient can now read it, under this name, from the document:

```
var pddoc= jEvent.target.document.impl;

/*
 * first, lookup PDApplicationData object named "foo"
 */
var appData = pddoc.getApplicationData("foo");

/*
 * grab data from object and display
 */
var data = appData.cosGetData();
app.alert(data.stringValue());
```



3.5 Page Content

3.5.1 Symbols

The visible contents of a PDF document are located in so-called *ContentStream* objects. These describe the look of pages with help from graphic commands such as 'Line from 12/323 to 34/999'. Of course there is also an API for creating and editing the ContentStream.

The basic process is as follows:

```
var pddoc = jEvent.target.document.impl;

var page = pddoc.pageTree.getPageAt(0);
var left = page.mediaBox.lowerLeftX;
var right = page.mediaBox.upperRightX;
var bottom = page.mediaBox.lowerLeftY;
var top = page.mediaBox.upperRightY;
```

- Select PDF document
- Select pages and retrieve dimensions

```
var cc =  
Packages.de.intarsys.pdf.content.common.CSCreator.createNew(page);
```

- Create new content for the page

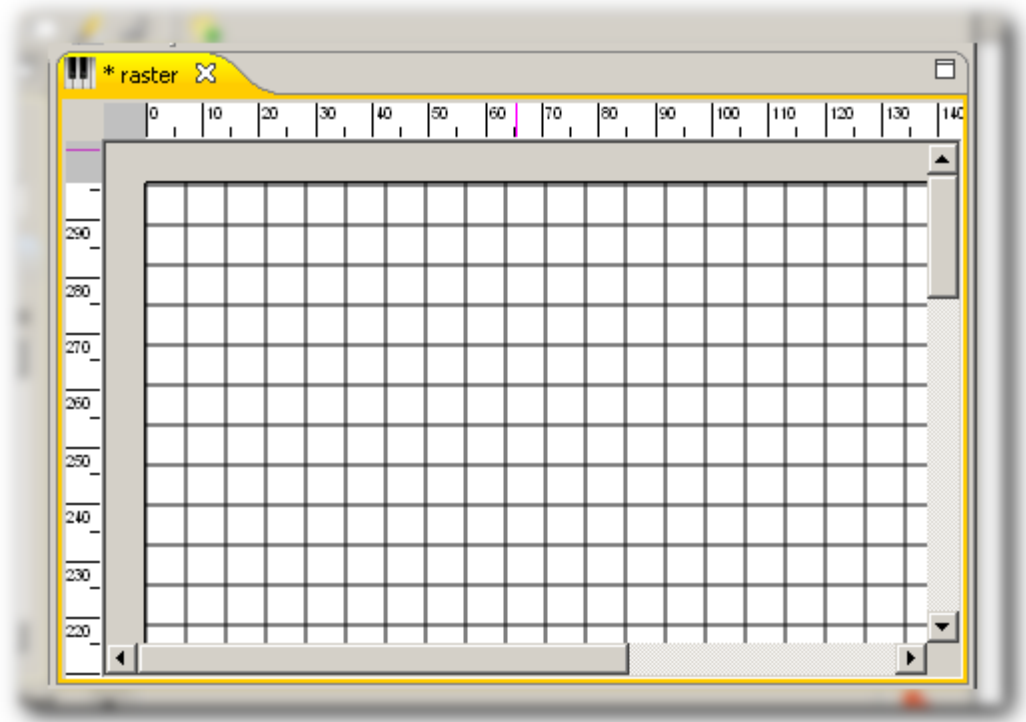
```
cc.saveState();  
cc.setLineWidth(0);  
for (i = bottom; i < top; i = i + 20) {  
    cc.penMoveTo(left, i);  
    cc.penLineTo(right, i);  
    cc.pathStroke();  
}  
for (i = left; i < right; i = i + 20) {  
    cc.penMoveTo(i, bottom);  
    cc.penLineTo(i, top);  
    cc.pathStroke();  
}  
cc.restoreState();  
cc.close();
```

- Define ContentStream contents and close.

And all together:

```
var pddoc = jEvent.target.document.impl;  
  
var page = pddoc.pageTree.getPageAt(0);  
var left = page.mediaBox.lowerLeftX;  
var right = page.mediaBox.upperRightX;  
var bottom = page.mediaBox.lowerLeftY;  
var top = page.mediaBox.upperRightY;  
//  
var cc =  
Packages.de.intarsys.pdf.content.common.CSCreator.createNew(page);  
//  
cc.saveState();  
cc.setLineWidth(0);  
for (i = bottom; i < top; i = i + 20) {  
    cc.penMoveTo(left, i);  
    cc.penLineTo(right, i);  
    cc.pathStroke();  
}  
for (i = left; i < right; i = i + 20) {  
    cc.penMoveTo(i, bottom);  
    cc.penLineTo(i, top);  
    cc.pathStroke();  
}  
cc.restoreState();  
cc.close();
```

Your document should now look something like this:



3.5.2 Text

There are many operations available for the ContentStream. Here we would like to provide a further example for using text.

```
var pddoc = jEvent.target.document.impl;

var page = pddoc.pageTree.getPageAt(0);

var cc =
Packages.de.intarsys.pdf.content.common.CSCreator.createNew(page);
cc.saveState();

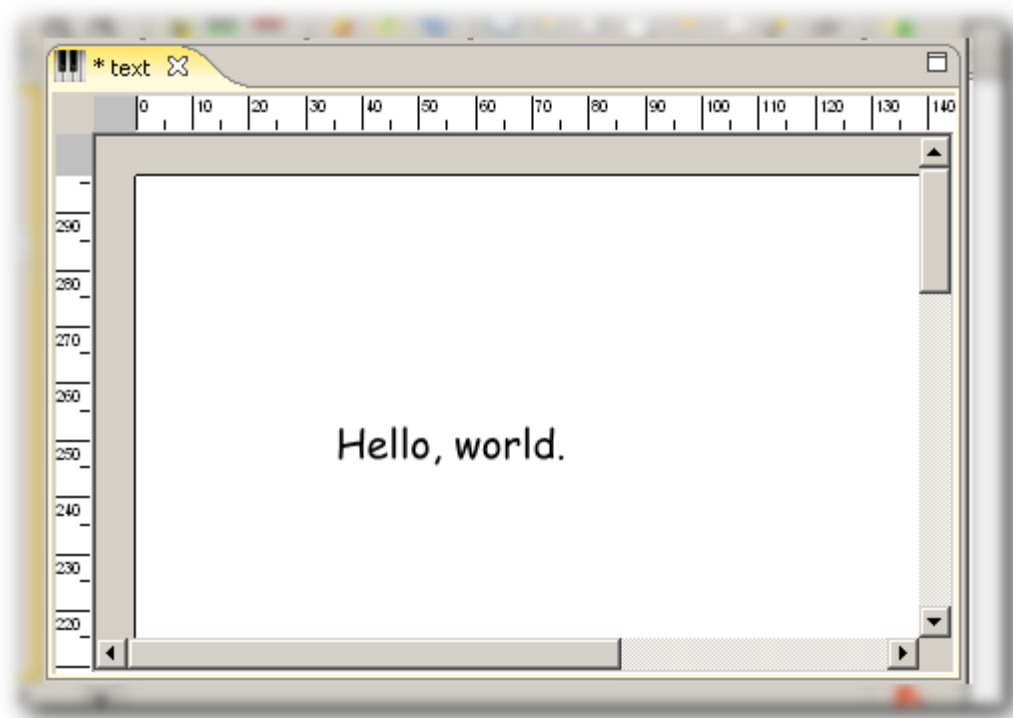
var factory =
Packages.de.intarsys.pdf.font.outlet.FontOutlet.get().lookupFontFactory(pddoc);

var query = new Packages.de.intarsys.pdf.font.outlet.FontQuery("Comic
Sans MS");
var font = factory.getFont(query);

cc.textSetFont(null, font, 20);
cc.textLineMoveTo(100, 700);
cc.textShow("Hello, world.");

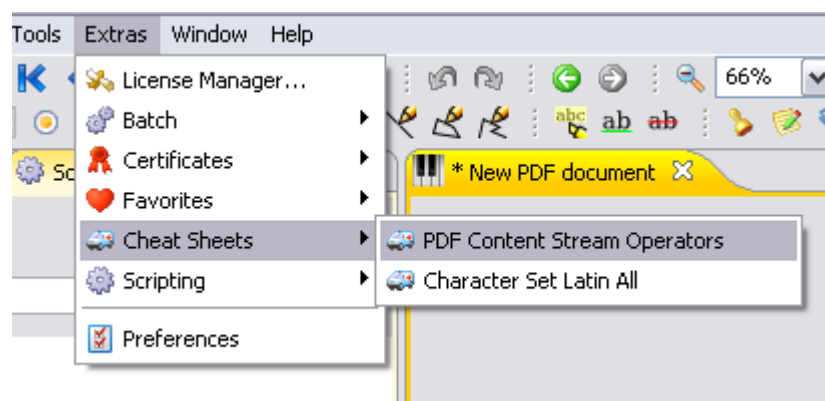
cc.restoreState();
cc.close();
```

The text “Hello, world” should now appear on your document

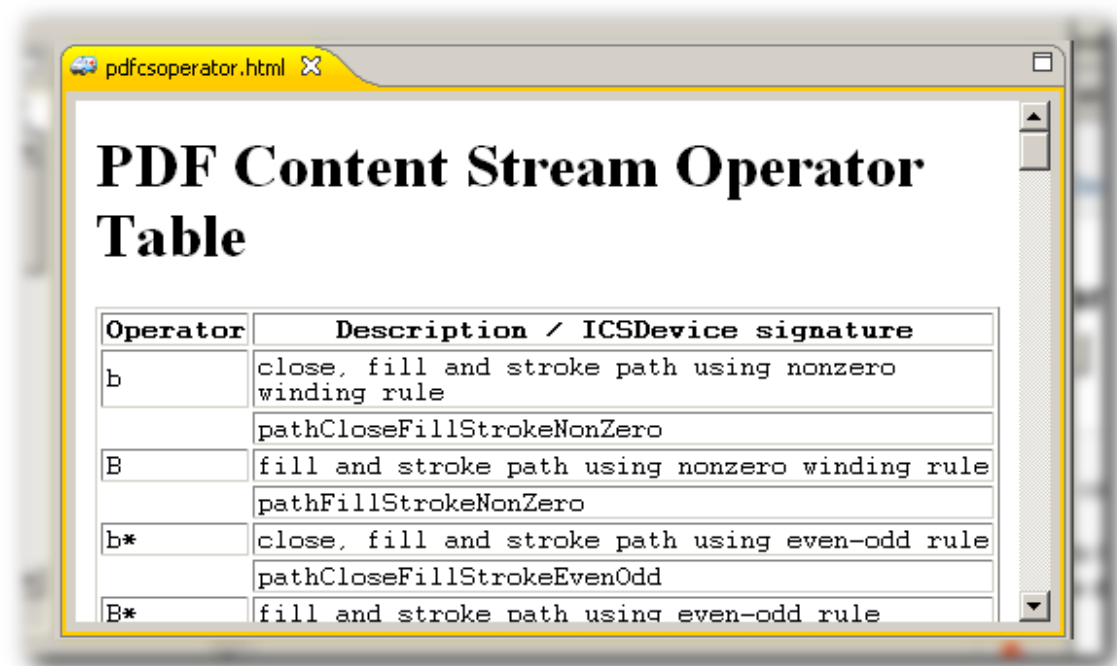


3.5.3 Tip

The complete range of functions for the graphic operators can be found in the reference. You can also access this information quickly in the Sign Live! CC Cheat Sheets:



A list of the defined operators will appear:

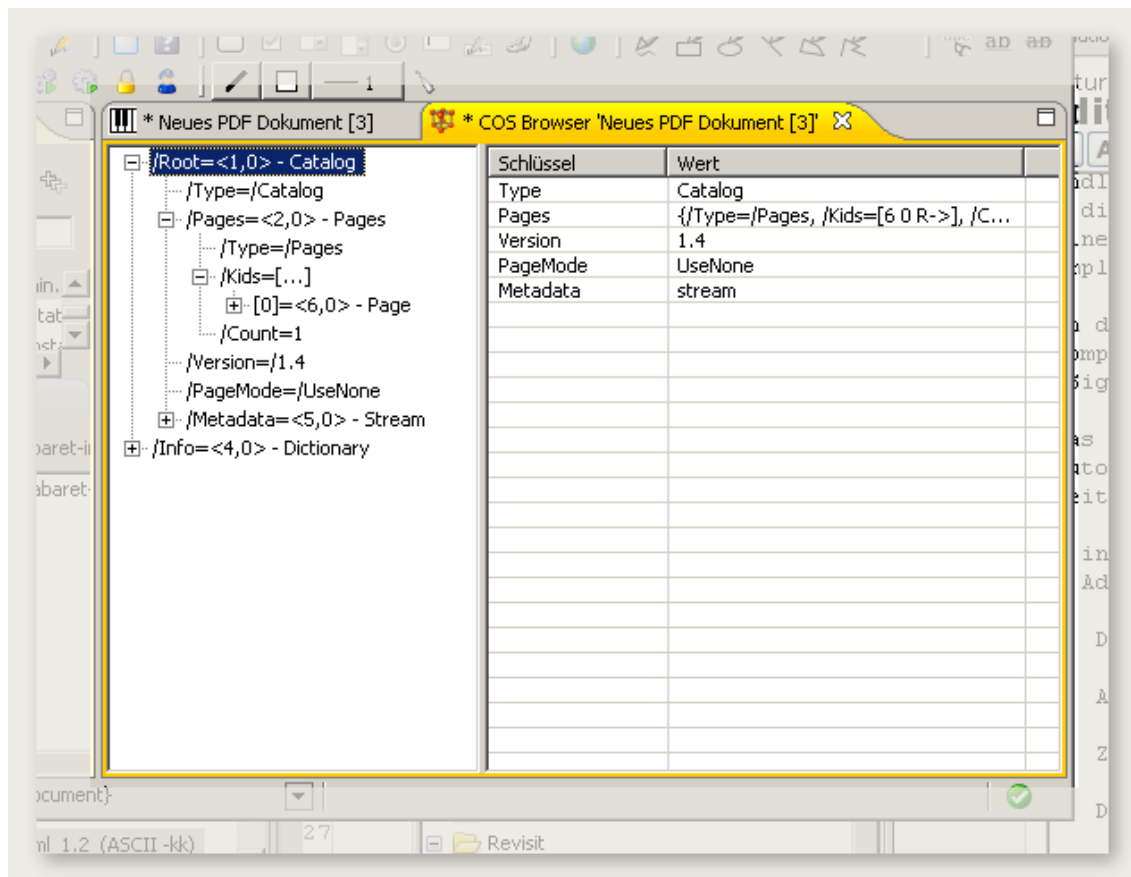


The screenshot shows a web browser window with the title bar 'pdfcsoperator.html'. The main content area displays the title 'PDF Content Stream Operator Table' in a large, bold, black serif font. Below the title is a table with two columns: 'Operator' and 'Description / ICSDevice signature'. The table lists four operators: 'b', 'B', 'b*', and 'B*', each with a description and a corresponding ICSDevice signature.

| Operator | Description / ICSDevice signature |
|----------|--|
| b | close, fill and stroke path using nonzero winding rule pathCloseFillStrokeNonZero |
| B | fill and stroke path using nonzero winding rule pathFillStrokeNonZero |
| b* | close, fill and stroke path using even-odd rule pathCloseFillStrokeEvenOdd |
| B* | fill and stroke path using even-odd rule |

3.6 Debugging

The “COS Browser” available in Sign Live! CC is an extremely useful tool for reviewing internal structures and debugging PDF documents. Review and edit the internal structures of PDF documents.



4. Working with Processors

4.1 Overview

Processors are the most important building blocks in Sign Live! CC. Basically all basis functions in Sign Live! CC are implemented as processors and are available to you. Even the simplest functions such as “Save” and “Print” are built on this scheme. This may seem laborious at first, but it will prove itself as a powerful and universal concept once you are familiar with the material.

You can search for and run processors under precisely defined names for your desired functions. A catalog of the available processors can be found in the corresponding reference.

The following examples show the most basic options - further possibilities are available through the combination and sequencing of the processors. Load a document, generate a test report, convert it to PDF, attach it, sign the result and export it to an archive...

4.2 Basis Document Functions

In the previous example we introduced basic processing of documents. Here we will expand upon these techniques by adding control through processors.

Two differences are important to keep in mind:

- The corresponding processors control more than just basis functions and can free you from tasks that are repeated often.
- The processors are always controlled from a uniform interface and are therefore better suited for automation.

4.3 Handle Documents

Lets use a processor to load a document:

```
var idoc = Processor.callArgs(  
    "DocumentLoaderFactory", {  
        locator: "c:/temp/foo.pdf"  
    });  
  
// ... something to do with idoc  
  
// important: release()  
idoc.release();
```

In addition to loading documents these processors perform other tasks in Sign Live! CC. For example, saving default values for the loading parameter, maintaining the list of most recent files and controlling the internal profiling mechanism.

```
Processor.callArgs(  
    "DocumentSaverFactory",  
    {  
        document: jEvent.target.document,  
        locator: "c:\\temp\\foo.pdf"  
    });
```

This processor saves the document under the provided name.

```
var idoc = Processor.callArgs(  
    "DocumentViewerFactory",  
    {  
        document: jEvent.target.document,  
    });
```

This processor opens the document for display.

4.4 HTML to PDF

In this section we will create an HTML document to be directly imported as a PDF and displayed.

```
var content = Resolver.html2pdf_template();  
var importedIDoc = Processor.callArgs(  
    "com.cabaret.pdf.exchange.html.HTMLImporterFactory", {  
        importLocator: content.locator  
    });  
Processor.callArgs(  
    "DocumentViewerFactory", {  
        document: importedIDoc  
    });  
importedIDoc.release();
```

The example shows, among other things, the use of JavaScript templates, which will be covered more closely in a later section.

4.5 Attaching Pages

This example will load a document (*sourceDoc*) and attach all pages from this document onto the PDF document currently open. This allows for easy implementation of processes that are typical for, for example, release procedures or quality management.

```
if (jEvent.target == null) {
    app.alert("open a document first");
} else {
    var idoc = jEvent.target.document;

    var location = Reflector.locator().getParent().fullName;

    var sourceDoc = Processor.callArgs(
        "DocumentLoaderFactory", {
            locator: location + "/demodoc.pdf"
        });
    sourceDoc.locator.setReadOnly();

    Processor.callArgs(
        "com.cabaret.pdf.processor.pageimporter.PageImporterFactory", {
            document: idoc,
            documentPage: "last",
            documentBeforeAfter: "after",
            sourceDocument: sourceDoc,
            sourceDocumentPageRange: "all"
        });

    sourceDoc.release();
}
```

4.6 Delete Pages

This example will delete the first page of the file selected for this script.

```
if (jEvent.target == null || jEvent.target.document == null) {
    app.alert("open a document first");
} else {
    Processor.callArgs(
        "com.cabaret.pdf.processor.pagedeleter.PageDeleterFactory", {
            document: jEvent.target.document,
            pageRange: '1'
        });
}
```

4.7 Stamp Documents

You can use Sign Live! CC to easily add stamps to your documents automatically.

Depending on the program variation you have installed, this processor may not be available.

```

if (jEvent.target == null || jEvent.target.document == null) {
    app.alert("open a document first");
} else {
    var idoc = jEvent.target.document;
    Processor.callArgs(
        "com.cabaret.pdf.processor.markup.MarkupCreatorFactory",
        {
            document: idoc,
            annotation: {
                subType: 'Stamp',
                stampName: 'Approved',
                position: '100@100',
                size: '200@80',
                pageRange: 'first'
            }
        }
    );
}

```

This code will create a stamp “Approved” from the standard library.

The following stamps are available in this library:

- Approved
- Completed
- Confidential
- Draft
- NotApproved
- Revised
- Void

With Sign Live! CC you can also create your own new libraries. Address the stamps with the following combination <LibraryName> “.” <StampName>.

```

Processor.callArgs(
    "com.cabaret.pdf.processor.markup.MarkupCreatorFactory",
    {
        document: idoc,
        annotation: {
            subType: 'Stamp',
            stampName: 'MyLib.Smiley',
            position: '100@100',
            size: '200@80',
            pageRange: 'first'
        }
    }
);

```

This code addresses the stamp *Smiley* in the library *MyLib*.

4.8 Create form fields

You can use Sign Live! CC to easily add form fields to your documents automatically.

Depending on the program variation you have installed, this processor may not be available.

```
if (jEvent.target == null || jEvent.target.document == null) {
    app.alert("open a document first");
} else {
    var idoc = jEvent.target.document;
    Processor.callArgs(
        "com.cabaret.pdf.processor.acroform.WidgetCreatorFactory",
        {
            document: idoc,
            field: {
                fieldType: "Tx",
                create: true,
                position: "100*300",
                size: "200*80",
                pageRange: "first"
            }
        }
    );
}
```

4.9 “Flattening” Documents

“Flattening” removes dynamic and interactive elements from a document without changing its visual appearance. This process is of particular interest in preparation to signing, archiving or delivering to third parties.

This process will be performed by the following script. This processor may also not be available depending on the program variation.

```
if (jEvent.target == null || jEvent.target.document == null) {
    app.alert("open a document first");
} else {
    var idoc = jEvent.target.document;
    Processor.callArgs(
        "com.cabaret.pdf.processor.flatten.FlattenerFactory",
        {
            document: idoc,
            setReadOnly: true
        }
    );
}
```

Standardly the processor will remove all actions and annotations - so be careful with saving. In order to avoid accidently overwriting important documents, documents are standardly write-protected.

The following arguments are supported:

- `setReadOnly`
The document will be write-protected after “Flattening”
- `actionAllRemove`
All actions will be removed. If this argument is “true”, the following arguments for special actions will be ignored. (Default: true)
 - `actionFormRemove`
All form-specific actions (Import, Export, Reset) will be removed. (Default: true)
 - `actionJavaScriptRemove`
All JavaScript actions will be removed. (Default: true)
 - `actionMovieRemove`
Movie actions will be removed. (Default: true)
 - `actionRenditionRemove`
Rendition actions will be removed. (Default: true)
 - `actionSetOCGStateRemove`
SetOCGStateA actions will be removed. (Default: true)
 - `actionSoundRemove`
Sound actions will be removed. (Default: true)
- `annotationMarkupRemove`
Markup Annotations will be removed
 - `annotationMarkupEmbed`
If Markup Annotations have been removed, the Appearances will first be statically embedded in the page.
- `annotationWidgetRemove`
Widget Annotations (Form fields) will be removed
 - `annotationWidgetEmbed`
If Widget Annotations have been removed, the Appearances will first be statically embedded in the page.
- `annotationOtherRemove`
All other Annotations will be removed
 - `annotationOtherEmbed`
If all other Annotations have been removed, the Appearances will first be statically embedded in the page.

4.10 Signing Documents

4.10.1 Overview

Signing documents is one of the main features of Sign Live! CC. In addition to the possibilities shown here there are many other expansions that allow for signatures in cooperation with many

providers and including qualified signatures in accordance with SigG (German Digital Signature Law).

4.10.2 Simple Signature

The currently open PDF document will be signed using the Sign Live! CC demo certificate. The signature will be embedded into the document according to PDF specifications.

```
if (jEvent.target == null || jEvent.target.document == null) {
    app.alert("open a document first");
} else {
    var idoc = jEvent.target.document;
    Processor.callArgs(
        'com.cabaret.security.document.signing.DocumentSignerFactory', {
            document: idoc,
            digestSigner:

'com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFactory',
            digestSignerArgs: {
                signerIdentifier: 'SerialNumber:8139571262270123122;',
                signerPassword: 'password'
            }
        });
}
```

The resulting document now has a hidden signature.

4.10.3 Visible Signature

Of course you can also create a visible signature field. This will require additional information about the field:

```

if (jEvent.target == null || jEvent.target.document == null) {
    app.alert("open a document first");
} else {
    var idoc = jEvent.target.document;
    Processor.callArgs(
        'com.cabaret.security.document.signing.DocumentSignerFactory', {
            document: idoc,
            digestSigner:

'com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFact
ory',
            digestSignerArgs: {
                signerIdentifier: 'SerialNumber:8139571262270123122;',
                signerPassword: 'password'
            },
            field: {
                create: true,
                fieldName: 'sigField',
                position: '100@100',
                size: '200@80',
                pageRange: 'first'
            }
        }
    ));
}

```

4.10.4 External (PKCS#7) Signature

Sign Live! CC can sign any document type. The following example will create an external signature for the open document - it can also be a text document.

```

if (jEvent.target == null || jEvent.target.document == null) {
    app.alert("open a document first");
} else {
    var idoc = jEvent.target.document;
    Processor.callArgs(

'com.cabaret.security.method.pkcs7.signing.PKCS7DocumentSignerFactory'
, {
        document: idoc,
        digestSigner:

'com.cabaret.security.device.keystore.signing.KeyStoreDigestSignerFact
ory',
        digestSignerArgs: {
            signerIdentifier: 'SerialNumber:8139571262270123122;',
            signerPassword: 'password'
        }
    }
    ));
}

```

A PKCS#7 file is created with the same name as the provided document, but with the extension *.pkcs7*.

4.11 Importing Images

Here we will convert a TIFF document to a PDF and display it.

This processor may not be available depending on your installation variation.

```
var scriptdir = Reflector.locator().parent.fullName;
var filename = scriptdir + "/test.tif";
var idoc = Processor.callArgs(
    "com.cabaret.pdf.exchange.image.ImageImporterFactory", {
        importLocator: filename,
        pageSize: "1000*1000",
        scaleWhen: "never",
        scaleProportional: "true",
        halign: "left",
        valign: "top"
    });
Processor.callArgs(
    "DocumentViewerFactory", {
        document: idoc
    });
idoc.release();
```


5. Java Integration

5.1 Overview

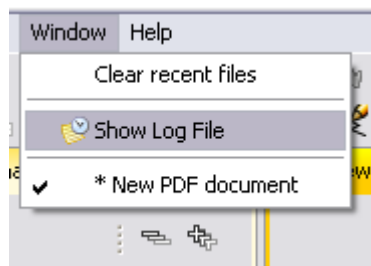
LiveConnect describes the ability of JavaScript implementation (Mozilla Rhino) to seamlessly switch back and forth between the JavaScript environment and Java.

This provides you with access to all the internal Sign Live! CC functions and, of course, a host of Java libraries for all sorts of tasks.

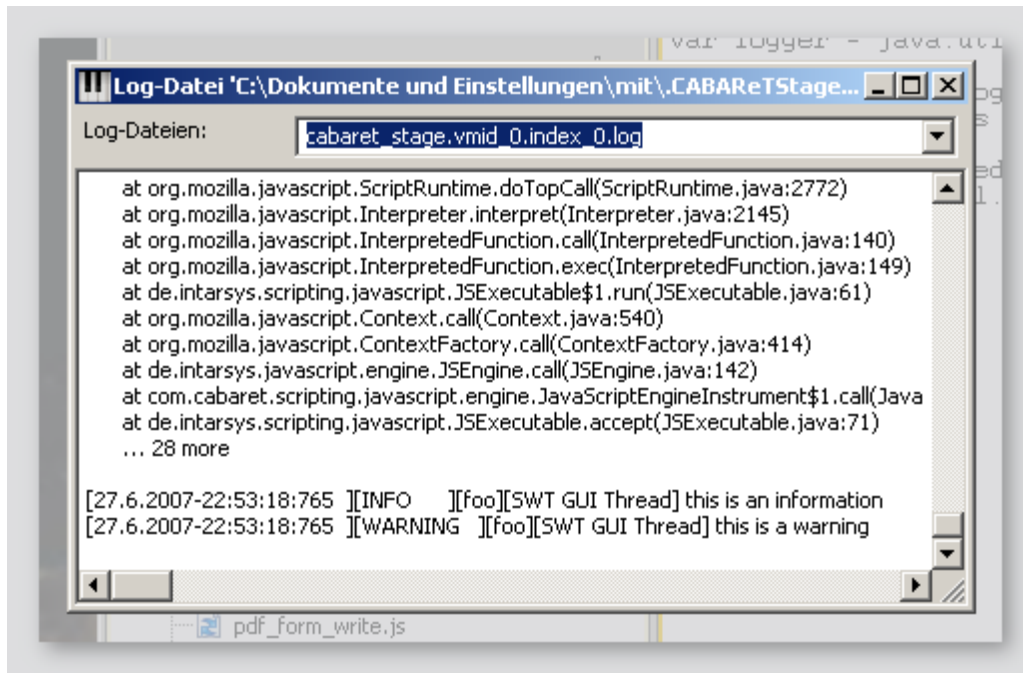
5.2 Accessing the Java Log

```
var logger = java.util.logging.Logger.getLogger("foo");  
  
/* LiveConnect to Logger methods */  
logger.info("this is information");  
  
/* more sophisticated control */  
logger.log(java.util.logging.Level.WARNING, "this is a warning");
```

In order to review the entry, you can inspect the Sign Live! CC Log:



A dialog with the current log file will be displayed. Your entry should be at the end of this log.



5.3 Write File

In this example we want to write data from our form into a file.

```
var file = new java.io.FileWriter("c:\\temp\\out.txt");
//
var pddoc = jEvent.target.document.impl;
var pdform = pddoc = pddoc.acroForm;
var pdfields = pdform.leafFields.toArray();
//
for (i in pdfields) {
    var pdfield = pdfields[i];
    file.write(pdfield.localName);
    file.write("; ");
}
file.write("\r\n");
for (i in pdfields) {
    var pdfield = pdfields[i];
    file.write(pdfield.valueString);
    file.write("; ");
}
//
file.close()
```

The result is something like a simple CSV file.

```
Textfeld2; Textfeld1; Textfeld3;
zwei; eins; drei;
```

As always there are a few details here that we should pay attention to. In particular the automatic conversion between Java and JavaScript data types such as the strings here can keep a programmer on their

toes. If you try the same example with `FileOutputStream` it will get even harder. There `write` is looking for a `byte[]` - something that `LiveConnect` can not create from the string that has in the meantime been converted to JavaScript. This will require manual labor.

5.4 Database Access

Here we will reverse the process. We presume you have a database with employee information available to you. The employees are presented under their Windows login names. We will use this data to at least partially fill out a form we are processing.

Again a suggestion for practical use - set this function to a toolbar icon or even a JavaScript function that will automatically be called when a document is opened. Your forms will be completed faster and more accurately than ever before!

This example will require a basic understanding of JDBC. We will access a simple table `t_demo` in the database `demo` that is installed as an ODBC data source.

```
CREATE TABLE `t_demo` (  
  `f_user` varchar(50),  
  `f_text1` varchar(50),  
  `f_text2` varchar(50),  
  `f_text3` varchar(50)  
);
```

There should be a data string for the logged in Windows user `f_user` - error handling is not included here.

```

java.lang.Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
//
var user = java.lang.System.getProperty("user.name");
var connection = java.sql.DriverManager.getConnection(
    "jdbc:odbc:demo",
    "root",
    ""
);
var statement = connection.createStatement();
statement.execute("select * from t_demo where f_user = '" + user +
"'");
var result = statement.getResultSet();
result.next();
var feld1 = result.getString("f_text1");
var feld2 = result.getString("f_text2");
var feld3 = result.getString("f_text3");
connection.close();
//
var pddoc = jEvent.target.document.impl;
//
var factory =
Packages.de.intarsys.pdf.app.acroform.FormHandlerFactory.get();
var handler = factory.createFormHandler(pddoc, null);
//
handler.setFieldValue("Textfeld1", feld1);
handler.setFieldValue("Textfeld2", feld2);
handler.setFieldValue("Textfeld3", feld3);

```

5.5 Shell Access

Via shell access Sign Live! CC can be integrated into workflows very easily. Sign Live! CC may use the functions provided by the operating system or may call other applications to perform the next step in the workflow.

The following example shows a simple Windows shell access which will rename signature files in the specified working directory. Avoid the use of blanks in parameters. They may cause the failure of the call.

```

var workingDir = new Packages.java.io.File("c:\\temp");
var commandAndParams = ["cmd.exe", "/C", "rename", "*.p7s", "*.s1"];
var runtime = Packages.java.lang.Runtime.getRuntime();
runtime.exec(commandAndParams, null, workingDir);

```

6. Process Automation and Integration

6.1 Overview

One of the applications greatest advantages is its ability to easily integrate functions through open interfaces. The purpose of this ability is the smooth execution of specialized processes, even without using one of the interactive variations of Sign Live! CC (“Headless”, Sign Live! CC Backstage).

This is all accomplished using the combination of scripting and open standard APIs.

6.2 Commandline (CLI)

The simplest way, which is also always available, to integrate a process is with the commandline (CLI). Sign Live! CC is started with a commandline together with control options, processes the commands and closes (optional).

Examples of possible commands

- Load PDF document and save in PDF/A format
- Load PDF document and print it
- Load TIFF file and save it as PDF
- Load document and sign it

This list can go on and on and the examples can be as complex as needed - the commandline for Sign Live! CC allows for flexible processing.

There are some predefined options for common tasks, for example,

- - - file (-f)
Load document

- - - print (-p)
Print document
- - - import (-im)
Import file

These options are covered in detail in the “Operator’s Guide”. In this tutorial we will focus more on scripting-based options. With these examples we will return to the beginning of this tutorial by using a *CodeExit* to realize calling a script from a menu or the toolbar.

Sign Live! CC can also use the commandline to start a *CodeExit* as a event source. The call looks something like this:

```
<SignLiveCC.exe> -perform -pt JavaScript -ps "app.alert('hello, world');"
```

with the following result



You can call your script directly from the commandline! Of course it would be even simpler to take the same route as in our starting example. Save your script as a file and call it using the type *Script* or *ScriptFile*. Simply use the script from the starting example.

```
<SignLiveCC.exe> -perform -pt Script -ps "c:\temp\hello.js"
```

Again, here is the required syntax

- - - perform (-perform), not an argument
introduces a *CodeExit* through the commandline
 - - - performtype (-pt), an argument
defines the *CodeExit* type, for example, *JavaScript*, *Script* or *ScriptFile*.
 - - - performsource (-ps), an argument
the code to be performed by the *CodeExit*.
 - - - performarg (-pa), an argument
an argument for the call. This parameter is a string with the form “<name>=<value>” and can appear as often as needed.

6.3 ULS

6.3.1 Overview

For more demanding environments there is a call up interface for Sign Live! CC available through various channels (Listener) and protocols. For this Sign Live! CC provides a (nearly) full scale J2EE container. Programming, deployment and administration are comparable to a simple web server.

The entire J2EE architecture is known in Sign Live! CC by name *ULS* (Ultra Light Server).

In order to understand the examples and to get the most out of your applications you should have a good understanding of the Java Servlet Specification.

In the standard delivery the ULS J2EE container is not activated. For this reason you will need to copy additional Instruments into your *instruments* directory for the following examples.

6.4 ULS - HTTP

6.4.1 Overview

The HTTP listener converts Sign Live! CC into a simple web server. You can deploy servlets in Sign Live! CC and call them from your client. As we will see later, the HTTP listener is “just one” of the available listener components.

The concepts being used here are as J2EE conform as possible - for this reason details can be found in the corresponding documentation <http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html>.

For the following examples you will need to install an HTTP listener and a web application. Simply copy the *demo\API\ULS\HTTP\Demo_ULS_ContainerHTTP* and *demo\API\ULS\HTTP\Demo_ULS_AppHTTPCodeExit* directories into your installation's instruments' directory. After restarting you will have a web server in Sign Live! CC on Port 8111 with a web application that can process calls through HTTP requests.

The web server is defined in *Demo_ULS_ContainerHTTP*. The indications to the listener are made in the corresponding *instrument.xml*, for example, the port the web server should be started on.

The web applications can be defined in their own Instrument, for example, as in *Demo_ULS_AppHTTPCodeExit*. Here a web application is installed with help from a reference to the web server already defined herein.

6.4.2 CodeExit call per GET and POST

In general you can deploy and call as many servlets in this infrastructure as needed. The delivery already includes one servlet that allows for use of the previously mentioned CodeExit declaration. This allows you to easily make functions available through the HTTP interface. A corresponding web application is defined in the example *Demo_ULS_AppHTTPCodeExit*.

This is how to declare a servlet with CodeExit in *web.xml*:

```
<servlet>
  <servlet-name>hello</servlet-name>
  <servlet-
class>com.cabaret.uls.servlet.codeexit.CodeExitServlet</servlet-class>
  <init-param>
    <param-name>serviceFunctor</param-name>
    <param-value>
<![CDATA[
<perform
  type="JavaScript"
  source="app.alert('hello, world')\"
/>
]]>
    </param-value>
  </init-param>
</servlet>

...

<servlet-mapping>
  <servlet-name>hello</servlet-name>
  <url-pattern>/hello/*</url-pattern>
</servlet-mapping>
```

com.cabaret.uls.servlet.codeexit.CodeExitServlet is the standard servlet that allows for CodeExit declarations.

The CodeExit itself is provided in known syntax (here in a CDATA section, elsewhere embedded in XML) as the servlet parameter *serviceFunctor*. In later examples we will not write any more literal JavaScript CodeExits. Although these are easy to create ad-hoc, they also have several disadvantages:

- No hot code deployment
- Longer scripts can get very complex
- High chance for errors due to the complex syntax

The servlet is mapped as usual on a URL pattern.

The CodeExit can, for example, be controlled with a typical browser through the navigation line.

```
http://localhost:8111/codeexit/hello
```

In case your browser “hangs” - remember, it is waiting for an answer from Sign Live! CC and Sign Live! CC is waiting for you to close the message box in turn!

6.4.3 Generic CodeExit Call

In the predefined example you will also find a servlet configuration that realizes a generic CodeExit call:

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

The implementation of the CodeExits is in “scripts/codeexit.js” in your web application directory

```
CodeExit.callArgs (
  type,
  source,
  {
    event: event,
    jEvent: jEvent
  }
);
```

The following call can also be started in your browser - the call has been split over several lines to make it more readable, please write it directly in one line):

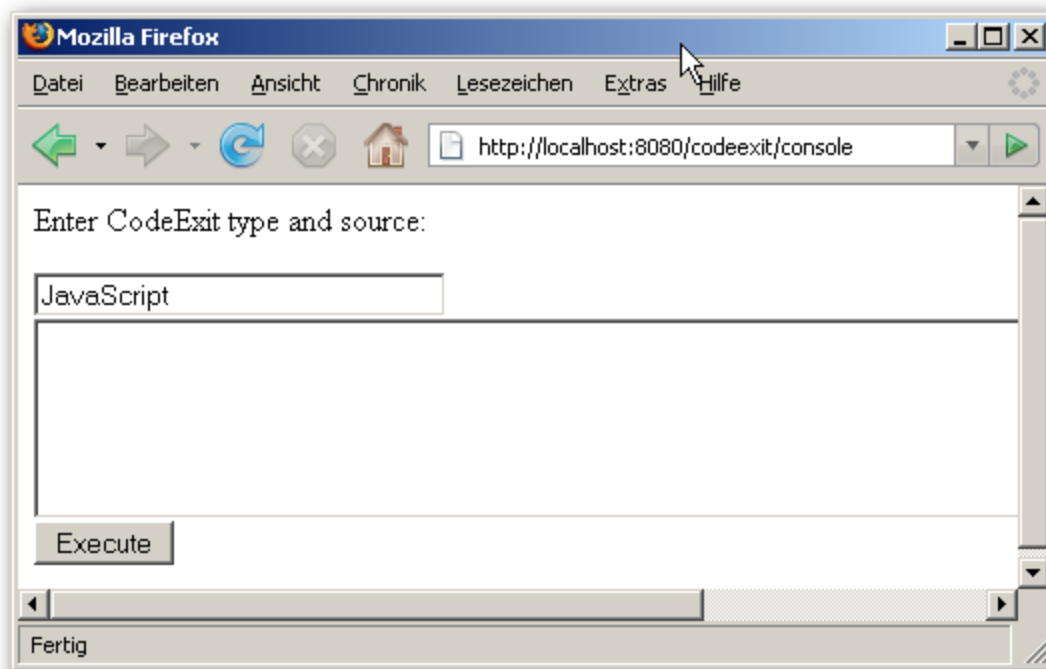
```
http://localhost:8111/codeexit/perform?
  type=JavaScript&
  source=app.alert('hello,world')
```

Of course the other CodeExit types we have discussed - “Script” and “ScriptFile” - will work here as well. By the time you get to the *ScriptFile* type, if not sooner, you will have noticed that making entries through a

browser's navigation line is not exactly a joy - who knows all HTML special character codes by heart...

But this has not gone unnoticed. The following URL will access a form that you can use to enter CodeExit information.

```
http://localhost:8111/codeexit/console
```



For advanced users: Of course this form is also generated in the Sign Live! CC web server based on the template "console.jst", which is located in the Instrument *Demo_ULS_AppHTTPCodeExit* in the *scripts* directory. Material is available here for those who wish to play around.

6.4.4 CodeExit POST Call with Documents

For lots of automation processes a document is provided by a client. Although you can generally realize any of your protocols for your client - due to the standard structure of Sign Live! CC and implementation based on J2EE - there is often a CodeExit-based servlet included in the delivery that you could also use. Here we assume that a document is being delivered through a POST request to the HTTP interface. The request can be delivered "plain" (the document is flat in the HTTP request data stream) or "multipart encoded".

You can use the following declaration in *web.xml*:

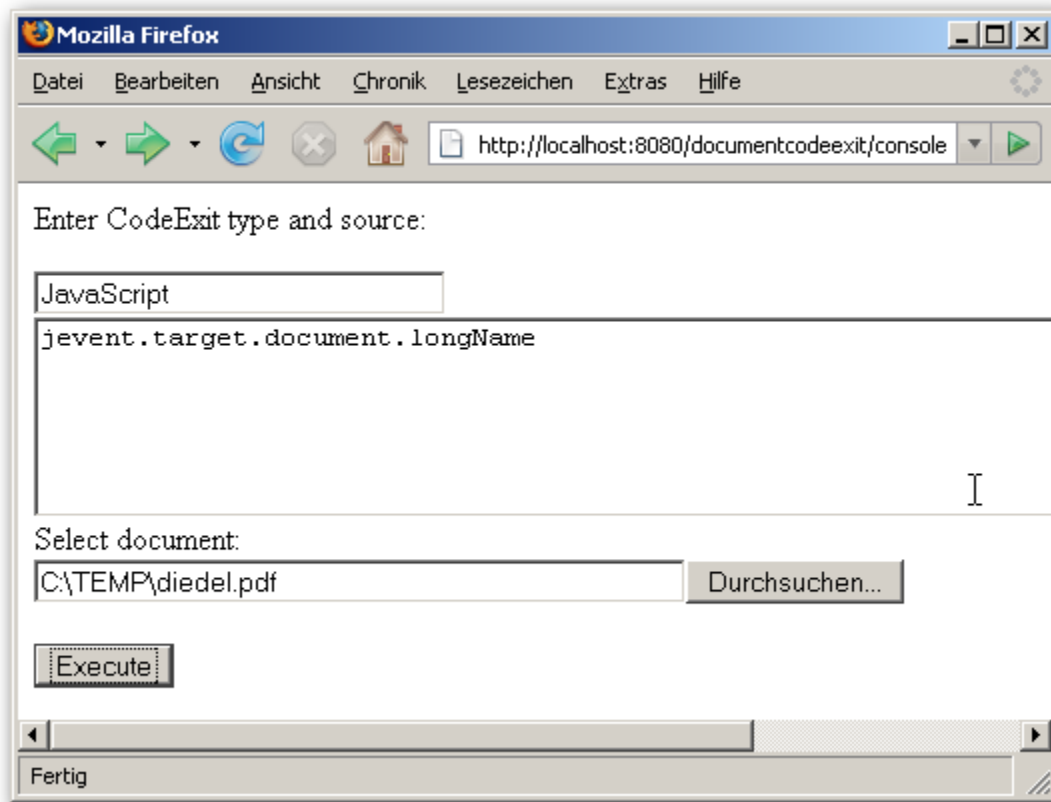

```
<servlet>
  <servlet-name>perform</servlet-name>
  <servlet-
class>com.cabaret.uls.servlet.application.DocumentUploadServlet</servl
et-class>
  <init-param>
    <param-name>serviceFunctor</param-name>
    <param-value>
<![CDATA[
<perform type="Script" source="scripts/codeexit"/>
]]>
    </param-value>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

The servlet is now *com.cabaret.uls.servlet.application.DocumentUploadServlet*. The CodeExit call itself is the same as in the previous example. The implementation of the CodeExit is in “scripts/codeexit.js” in your web application directory.

```
CodeExit.callArgs (
  type,
  source,
  {
    event: event,
    jEvent: jEvent
  }
);
```

This interface can no longer simply be called using the navigation line, but here there is also a simple form that realizes an “upload”.

```
http://localhost:8111/documentcodeexit/console
```



Use it as a test client or as a basis for your own scenario. Of course you can also perform this call programmatically in your system.

Play around with the script code some and combine it with other examples from this tutorial. The following code, for example, will deliver the name of a document back to the server:

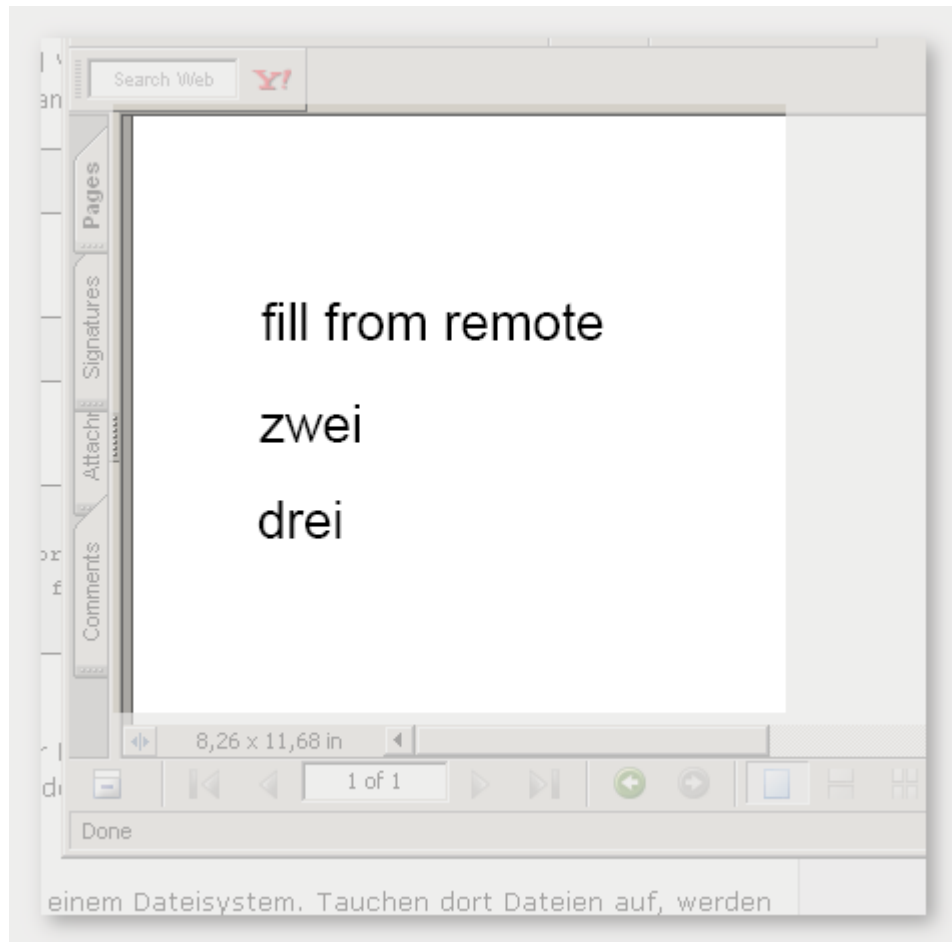
```
jevent.target.document.longName
```

or the document itself

```
jevent.target.document
```

or a completed document on the server

```
var pddoc = jEvent.target.document.impl;  
var factory =  
Packages.de.intarsys.pdf.app.acroform.FormHandlerFactory.get();  
var handler = factory.createFormHandler(pddoc, null);  
handler.setFieldValue('Textfeld1', 'fill from remote');  
// this line necessary to save changes made above  
jEvent.target.document.save();  
// this line is necessary to define document as return value!  
jEvent.target.document;
```



The possibilities are unlimited...

6.5 ULS - File System Monitor

6.5.1 Overview

Another member of the ULS listener family is the File System Monitor. The file system monitor observes directories in a file system. If files appear, they will immediately be passed on for further processing.

This helps to cover a host of processing scenarios using simple configurations:

- “Silent Printing”
When a file appears it will be printed in the background.
- Display “Print Operation”
An application generates print output in the background. This will then be opened at the workstation for display and further processing.
- Convert, sign, archive, ... in the background
Files from a core application are taken over in the background, automatically processed by Sign Live! CC scripts and then passed on.

The file monitor is fully embedded in the ULS framework. The configuration for processing is therefore also conformed to the J2EE standard and based on our predefined or your self-written servlets.

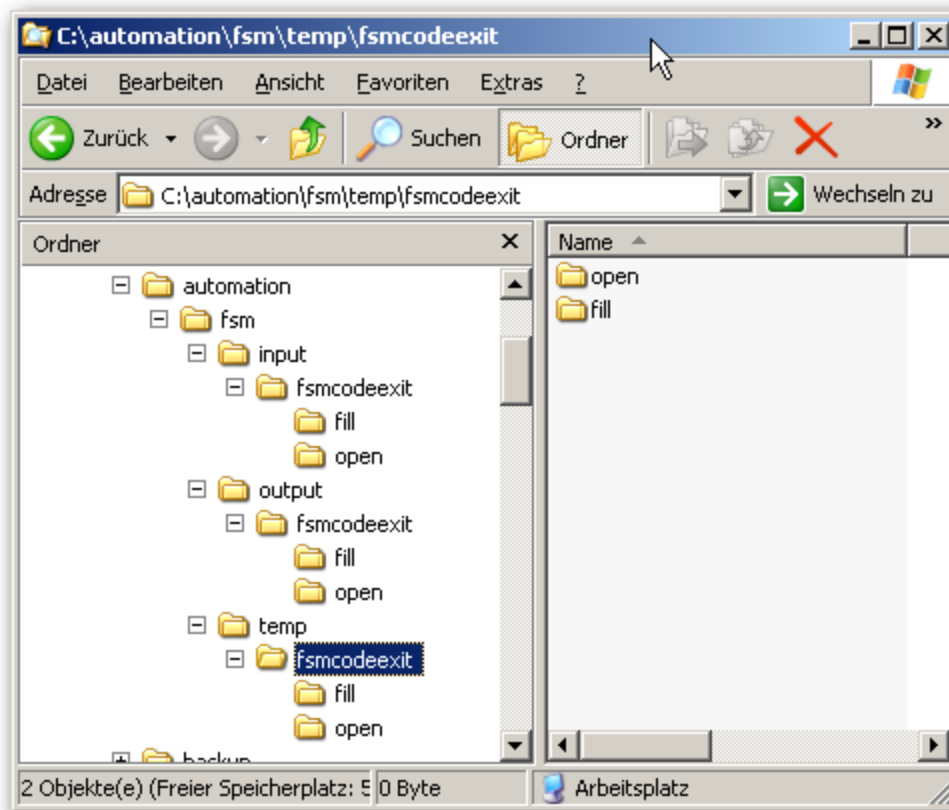
The examples here are based on the Instruments *Demo_ULS_ContainerFSM* and *Demo_ULS_AppFSMCodeExit*, which you should copy from the *demo/ULS/FSM/instruments* directory into your *instruments* directory of the Sign Live! CC installation.

6.5.2 Configuration

Unlike with HTTP, users will not be able to make entries for monitoring directories. This means that all entries for processing are more or less provided by the FSM listener configuration. These can be found in the “instrument.xml” of the FSM listener, which in our example is located in the directory *Demo_ULS_ContainerFSM*. Detailed information about the available attributes can be found in the reference. Here are just the most important:

- MonitorDir
The directory to be monitored. In *Demo_ULS_ContainerFSM* this is set to “c:\temp\fsm\input”)
- TempDir
A directory for temporary data for FSM (*Demo_ULS_ContainerFSM* “c:\temp\fsm\temp”)
- OutputDir
A directory for the processing results. If nothing is entered here the servlet’s return will be ignored. (*Demo_ULS_ContainerFSM* = “c:\temp\fsm\output”)
- ScanSubDir
Flags whether subdirectories of the entry directory should be monitored as well. (*Demo_ULS_ContainerFSM* = “true”)

Entry files must therefore appear in the directory *c:\temp\fsm\input* or its subdirectories, the results will appear in *c:\temp\fsm\output*.



This is what a suitable container declaration looks like:

```
<extension point="com.cabaret.uls.containers">
  <container id="com.cabaret.demo.uls.container.ContainerDemoFSM">
    <listener
      class="de.intarsys.fsm.kernel.FSMLListener"
      MonitorDelay="5"
      ScanSubDir="true"
      MonitorDir="c:\\temp\\fsm\\input"
      TempDir="c:\\temp\\fsm\\temp"
      OutputDir="c:\\temp\\fsm\\output"
      DeleteImmediately="true"
    />
  </container>
</extension>
```

There are many more options for adjusting the FSM for varying delivery processes. Further information hereto can be found in the reference.

6.5.3 CodeExit Call through an Entry File

The Instrument *Demo_ULS_AppFSMCodeExit* defines a web application *codeexit* with the demo servlet *open*. The *open* servlet will open the delivered document in Sign Live! CC.

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

This servlet definition uses the script “scripts/open” in your web application directory.

```
var idoc = jEvent.target.document;
Processor.callArgs(
  'DocumentViewerFactory',
  {
    document: idoc
  });
/* this line is necessary to define the document as the return value!
*/
idoc;
```

The servlet is mapped to the path /open/*.

Controlling various servlets and servlet context is accomplished using the relative file path name. To control the *open* servlet in the servlet context *codeexit* the file must be completely located in

```
c:\temp\fsm\input\codeexit\open
```

This file will be found by the FSM and displayed in Sign Live! CC.

The result of the processing (here the original document) will be written to the “output” branch. The result can be found in

```
c:\temp\fsm\output\codeexit\open
```

Therefore the naming pattern for the entry is always:

```
<OutputDir>\<Servlet Context>\<Servlet Name>\<Rest>
```

The other directory structures are similarly built.

Combine this example with others in this tutorial - a form can be stamped, signed, completed, etc. in the background.

6.6 ULS - P9100

6.6.1 Overview

The P9100 protocol is a simple and standardize way of controlling printing. You can use this component to control Sign Live! CC as a network printer from anywhere! Details about the P9100 protocol can be found on the Internet.

You can further process the delivered print data stream in Sign Live! CC using the appropriate servlets. A few examples:

- Open the document delivered in PDF format as a preview
- Convert documents delivered in a different format
- Filter the document for embedded commands

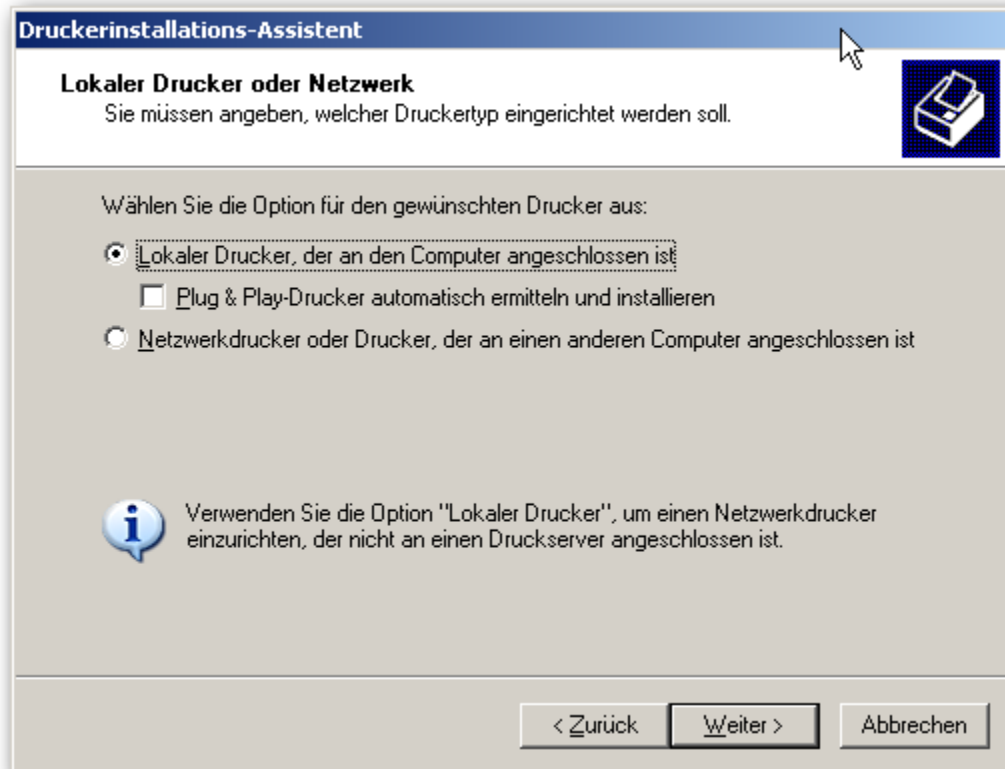
The P9100 monitor is also embedded in the ULS framework. This means that configuration is also J2EE standard conform and is based on our predefined or your self-written servlets.

The examples here are based on the Instruments "Demo_ULS_ContainerP9100" and "Demo_ULS_AppP9100Import", which you should copy from the "demo/API/ULS/instruments" directory into your Instruments directory of the Sign Live! CC installation.

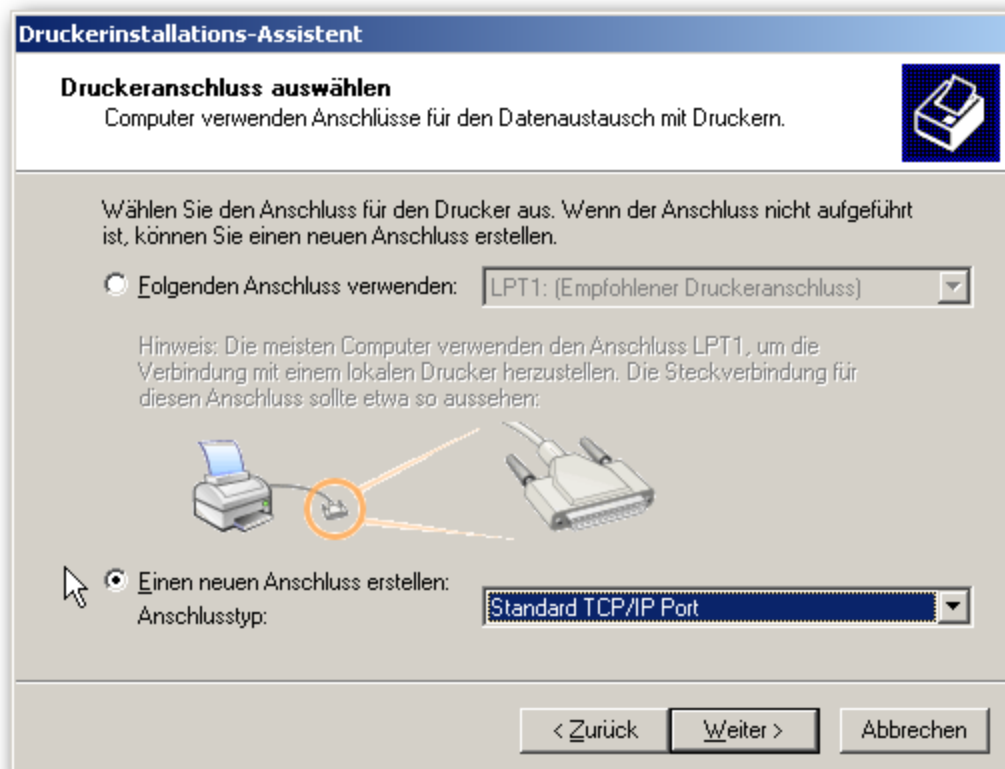
Note that the demo installation for the P9100 protocol has Port 9199 configured, so as not to conflict with other product installations.

6.6.2 Printer Installation

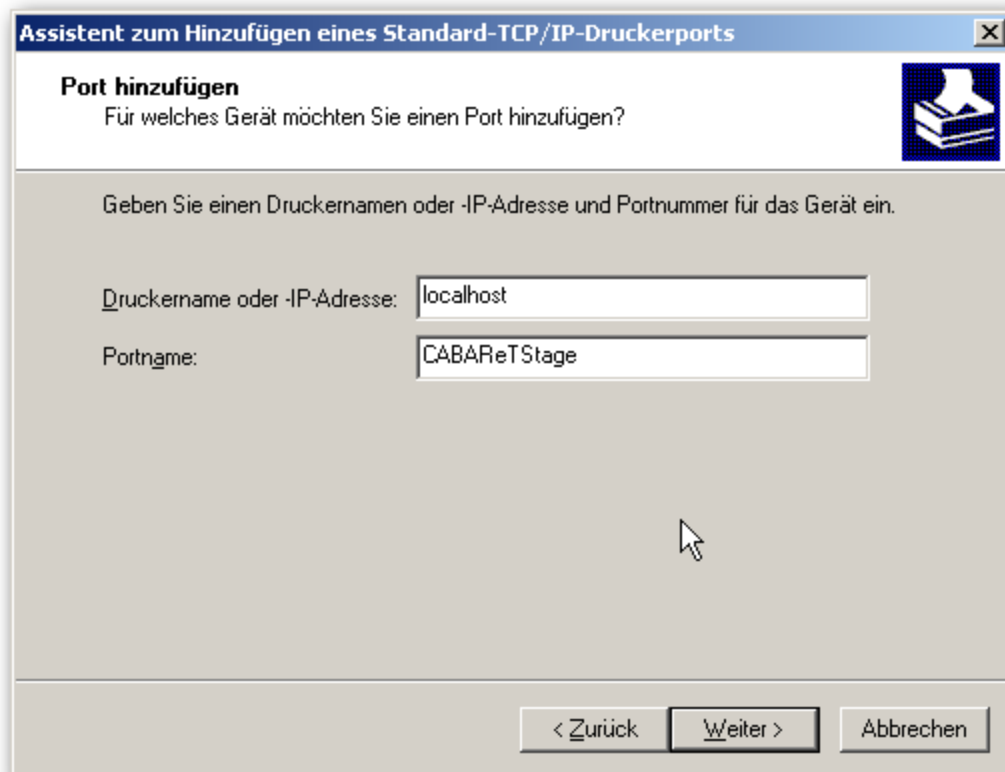
In order to control Sign Live! CC you will require an appropriate printer here, which you can, for example, install in Windows yourself.



A little counterintuitive here: you will definitely need to select a local printer.



Now you will need to create a new connection of the type “Standard TCP/IP Port”



Enter the IP address for the computer Sign Live! CC is running on. In the simplest case this will be “localhost”. The port name is a logical name for the connection, not the port number. When you click on “Next” it can take a little while, since windows will be attempting to get information from the printer (which it won’t find...).



On this page simply select “User Defined”. The settings should be entered as in the previous dialog (don’t forget to use the right port, in our demo 9100).

Standard-TCP/IP-Portmonitor konfigurieren

Porteinstellungen

Portname: CABAReT Stage

Druckername oder -IP-Adresse: localhost

Protokoll

☒ Raw ☐ LPR

Raw-Einstellungen

Portnummer: 9100

LPR-Einstellungen

Warteschlangenname:

☐ LPR-Bytezählung aktiviert

☐ SNMP-Status aktiviert

Communityname: public

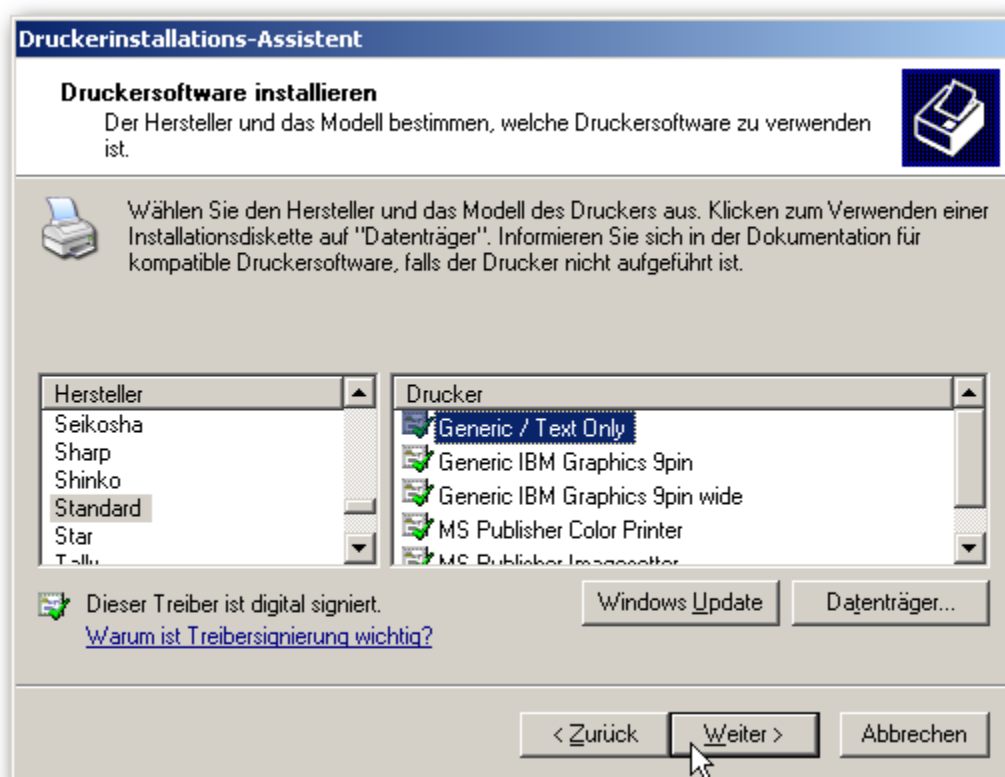
SNMP-Geräteindex: 1

OK Abbrechen

With these settings Sign Live! CC can now be used as a printer port.

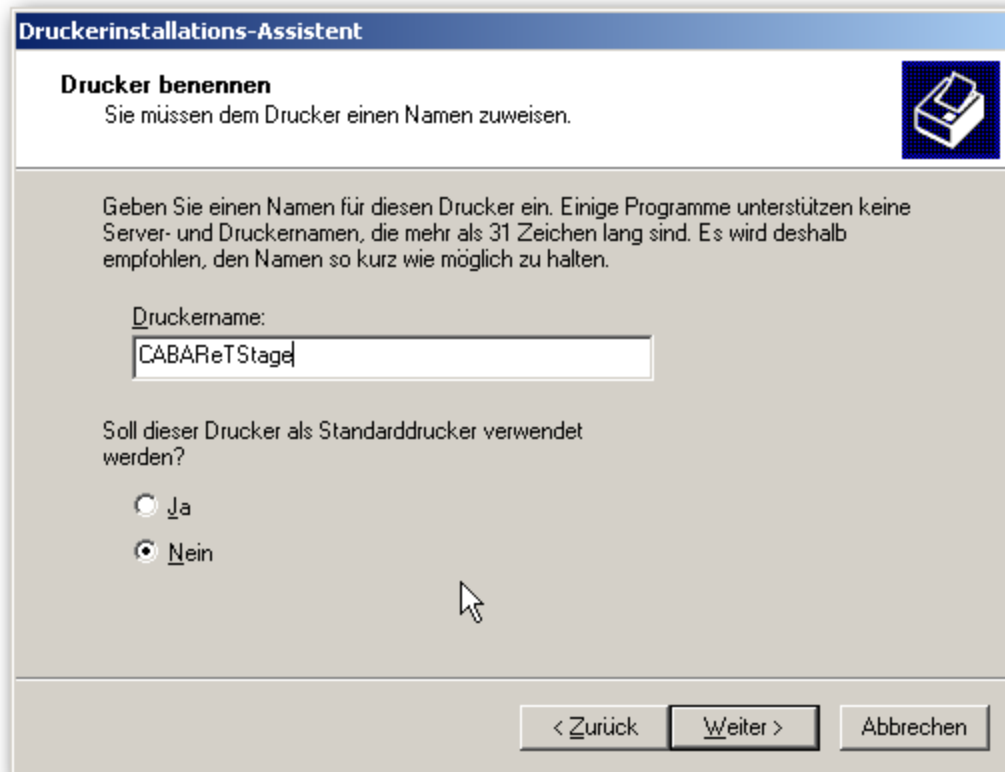


Select "Finish".



Now you will need to select a print driver. The selection will determine which data will later be received by Sign Live! CC. The displayed driver

“Standard/Generic Text only” will only deliver the text data (for example, from Notepad) to the printer and as such is well suited for experimenting. You could, for example, also select a postscript driver, if you are using the option “Postscript to PDF” in Sign Live! CC.



Now provide a name for your printer (the previous name was for the port. The names can be the same.)

6.6.3 Configuration

As with the FSM, here there are also no possibilities for the user to enter parameters. This will again require that all settings are made in the configuration, especially the path for the web application servlet to be called is now fully static. For the FSM a selection could be made for the user directories. That is not the case here.

The configuration is as usual in the “instrument.xml” for the P9100 listener. In our example in the directory “Demo_ULS_ContainerP9100”. Detailed information about the available attributes can be found in the reference. Here is just the most important:

- **URIPrefix**
The selection of the web application and the servlet must be made here. This means that this setting represents the path to your servlet, for example, “/print/import” in our example.

6.6.4 Convert and Open a Data Stream

An interesting use is to automatically convert an entry stream. In the following simple example an HTML text will be converted to a PDF document and displayed. Further scenarios such as automatically open, sign, stamp, etc. can be easily deduced.

Copy the Instrument “Demo_ULS_AppP9100Import”. The configuration of the servlet:

```
<web-app>
  <display-name>Demo ULS P9100 Monitor</display-name>
  <servlet>
    <servlet-name>import</servlet-name>
    <servlet-
class>com.cabaret.uls.servlet.application.DocumentUploadServlet</servl
et-class>
    <init-param>
      <param-name>tempfile</param-name>
      <param-value>import_${request.id}.html</param-value>
    </init-param>
    <init-param>
      <param-name>serviceFunctor</param-name>
      <param-value>
<![CDATA[
<perform type="Script" source="scripts/import"/>
]]>
      </param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>import</servlet-name>
    <url-pattern>/run/*</url-pattern>
  </servlet-mapping>
</web-app>
```

The script for the CodeExit call is expected in the file “scripts/import.js”:

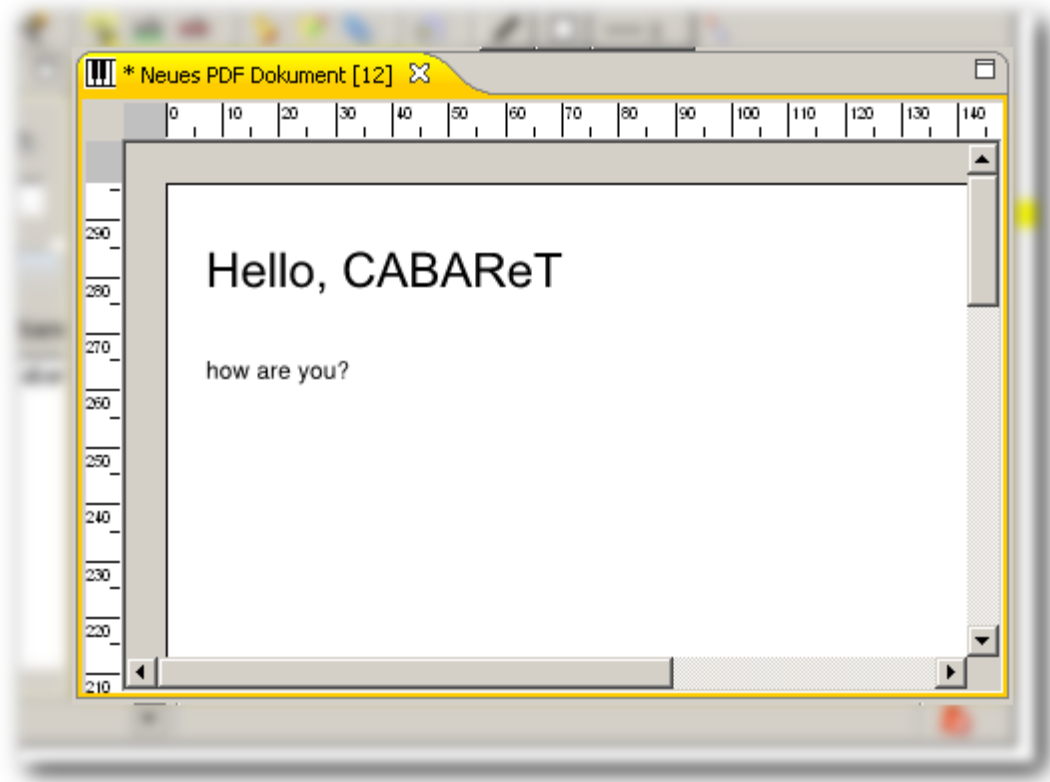
```
var locator = jEvent.target.document.locator;
var importedIDoc = Processor.callArgs(
    'com.cabaret.pdf.exchange.html.HTMLImporterFactory',
    {
        importLocator: locator
    });
if (importedIDoc != null) {
    Processor.callArgs(
        'DocumentViewerFactory',
        {
            document: importedIDoc
        });
    importedIDoc.release();
}
```

When printing, the *import* servlet will be called. The CodeExit there will import the data stream into a PDF document and display it.

Now use a text editor (for example, Notepad) and make sure that no additional headers or footers are generated (otherwise your HTML will be changed). Use the following HTML fragment and print it on your Sign Live! CC printer.

```
<html>
<body>
<p style="font-size:2em; font-family:Arial">Hello, World<p>
<p>how are you?</p>
</body>
</html>
```

Sign Live! CC will create a PDF and display it.



This example can be converted to a real PDF printer with a broad range of filter possibilities using the Sign Live! CC PS importer based on Ghostscript.

6.7 Review ULS (J2EE Container)

You might have already noticed that this is just an appetizer, examples of all the possibilities for Sign Live! CC would be simply too much. Again, here is a quick summary of some of the options available:

- Listener / Connectors
 - HTTP
 - XMLRPC
 - SOAP
 - File System
 - P9100
 - TN3270E
 - POP3
- Servlets
 - CodeExit
 - DocumentCodeExit
 - User defined servlets

- CodeExits / Scripts
 - Freely programmable
- Function Building Blocks
 - PDF Library
 - Predefined processors
 - Your own processors
 - Java integration
- Document types
 - PDF
 - Text
 - HTML
 - Image types (BMP, PNG, TIFF, JPEG)
 - User defined types
- ...

6.8 Web Services

6.8.1 Overview

The Web Services Server allows for simple integration into a SOA based environment. Sign Live! CC (version 4.2.1 or higher) provides an Web Services Server.

Examples for this section can be found in the installation under “demo/API/WS”. For more information see the Developer’s Guide.

6.9 ActiveX

6.9.1 Overview

The ActiveX gateway allows for simple integration into a Windows-based environment. The Internet Explorer integration is, for example, implemented on this basis.

The application (version 3.2 or higher) provides an ActiveX control, which allows for both interactive and “headless” (no window) control of document functions. Future versions will also support integration of functions at the application level.

Examples for this section can be found in the installation under “sdk\ActiveX\demo”.

6.9.2 Declaration

The function to be called in Sign Live! CC is declared in an Instrument to make it available for the ActiveX interface. The following fragment is used in an *instrument.xml* to declare the function *hello*, which will open a message box you might remember.

```
<extension point= "com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.activedoc.CommonActiveDoc"
    name="hello">
    <perform type="JavaScript" source="app.alert( { nIcon: 3 , cMsg:
'hello' } )"/>
  </method>
</extension>
```

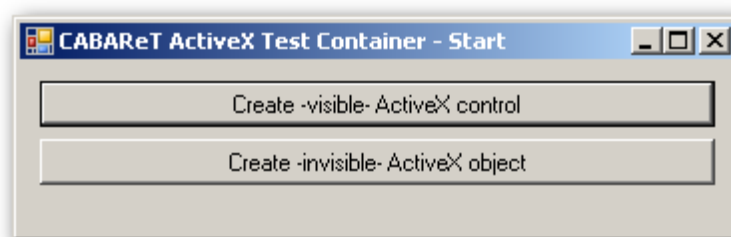
This function can be called through the ActiveX interface using one of the defined methods **AsyncCallXXX** or **SyncCallXXX**:

```
...
activedoc.AsyncCall("hello", handle);
...
```

6.9.3 C# Demo Container

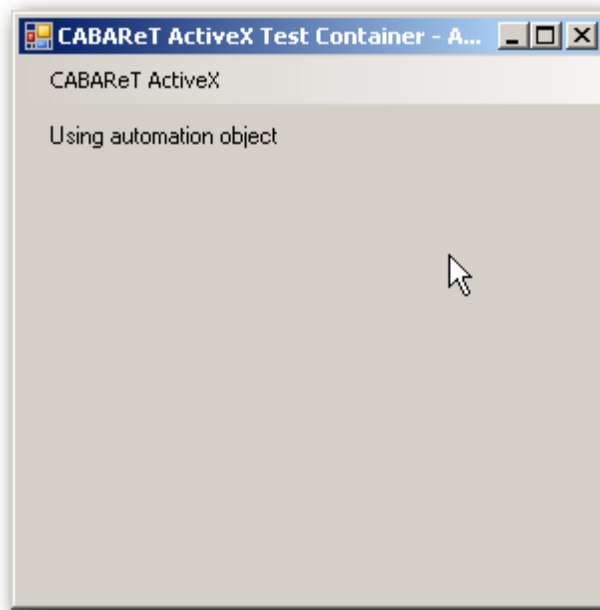
In order to provide you with a source example and a testing environment we have included a test program in the *demo/API/ActiveDocument/DemoContainer* directory of your installation.

When you start this program you can load a document and call the function configured above.

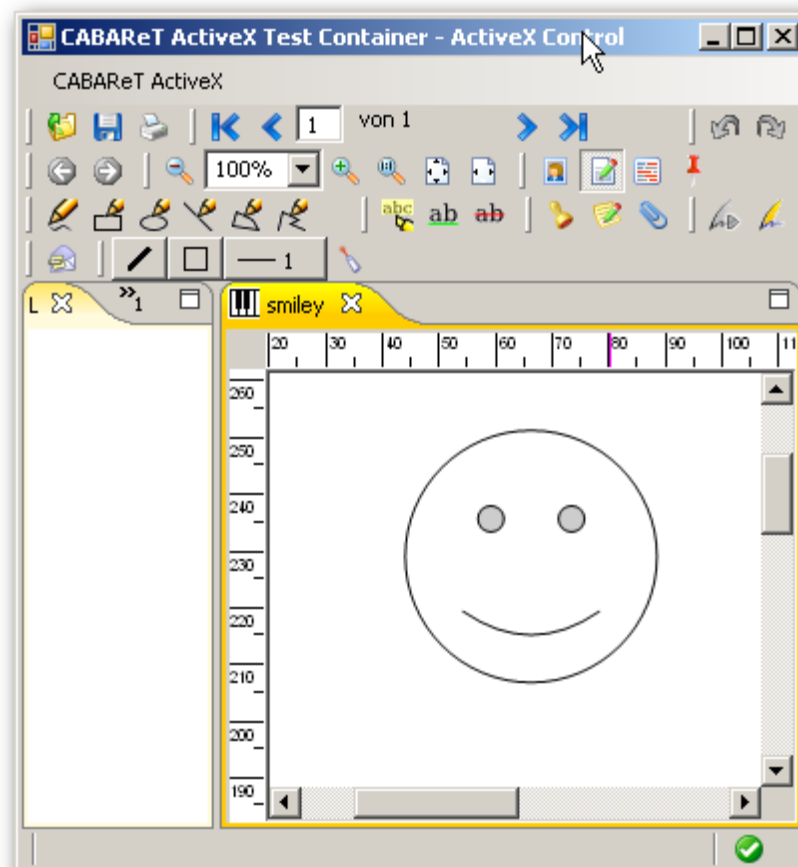


The “main window” allows for the starting of visible and hidden components. After one of the buttons has been selected you will need to select the document to be processed using a normal file dialog.

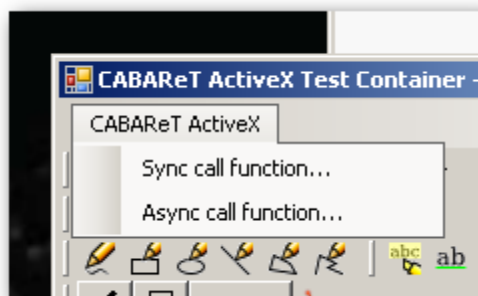
The following window will appear for hidden components:



The visible components will be displayed as usual:



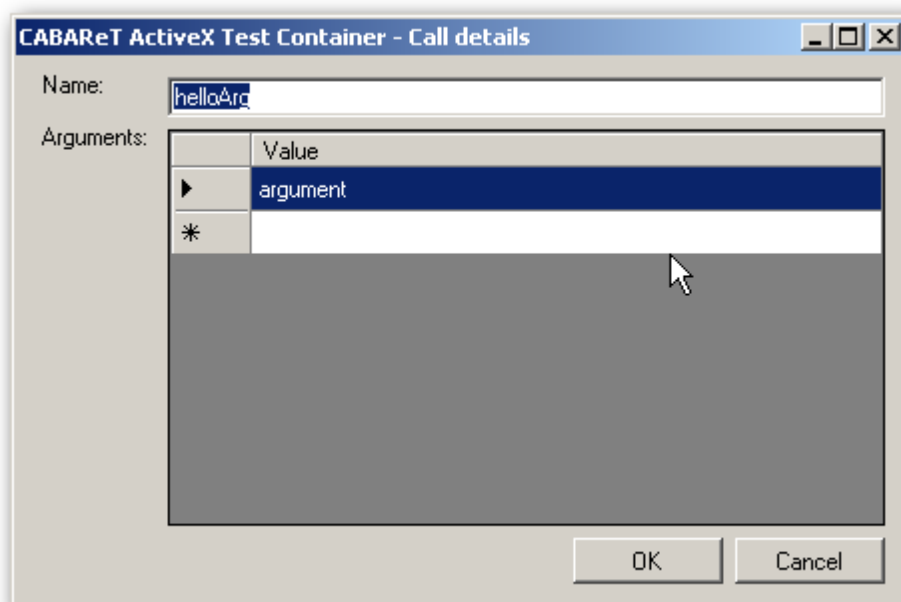
In both cases you can enter commands for the components through the menu.



Sync call function will run a synchronized call, i.e. the result will be waited for and returned to the client.

Async Call function will run an asynchronized call. The call will return immediately without result.

The dialogs behind the menu points are identical:



Enter the name of the function and optional arguments here. By clicking on "OK" the function will be called.



6.9.4 Visual Basic Script Demo Container

The call in VB script is also simple:

An example can be found in “demo/API/ActiveDocument”

```
Dim documentName : documentName = OpenFile(1)
If documentName = "" then WScript.Quit(0)

' the server name used depends on your installed program variation
Set s = CreateObject("CABARETStage.ActiveDocument")
s.loadfile (documentName)
result = s.SyncCall ("hello")

MsgBox "Result: " & result

Set s = NOTHING

'-----
Function OpenFile(filterIndex)
    Dim ofso      : Set ofso =
CreateObject("Scripting.FileSystemObject")
    Dim oDlg      : set oDlg =
Wscript.CreateObject("UserAccounts.CommonDialog")

    oDlg.Filter = "All Files (*.*)|*.*|Signature Files|*.pkcs7"
    oDlg.FilterIndex = filterIndex
    oDlg.ShowOpen

    If oDlg.FileName > "" and ofso.FileExists(oDlg.FileName) then
        OpenFile = oDlg.FileName
    Else
        OpenFile = ""
    End if
End Function
```

6.9.5 Parameter Transfer

All arguments from the client page for the function are made directly available as arguments of the CodeExit. The simplest way to access the arguments is through the declaration in the CodeExit. These can also be accessed through the reflective functions *Reflector.declareArg(name)* or *Reflector.argAt(index)*.

```
<extension point= "com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.activedoc.CommonActiveDoc"
    name="helloArg">
    <perform
      type="JavaScript"
      source="app.alert({ nIcon: 3 , cMsg: Reflector.argAt(0,
'default') })"/>
    </method>
  </extension>
```

6.9.6 CodeExit with Event

This CodeExit is also called with the arguments named “event” and “jEvent” in addition to the indexed arguments.

In this case the *target* is the document’s bundled IProcessor. This way identical scripts can be used for calls via commandline, HTTP or an action call from the user interface..

```
<extension point= "com.cabaret.claptz.objectmodel.members">
  <method
    implementor="com.cabaret.activedoc.CommonActiveDoc"
    name="helloArg">
    <perform
      type="JavaScript"
      source="app.alert(jEvent.target.document.longName)"/>
    </method>
  </extension>
```

7. JavaScript based Templates

7.1 Overview

Templates offer a simple but powerful way to create maintainable text-based documents quickly. A text-based result can be anything from an HTML, XML or CSV file to a business report.

Templates reverse the focus of the programming - instead of lining up character after character with procedural commands, resulting texts are written and then commands and calculations are embedded in them.

Sign Live! CC has a powerful integrated template engine. Its dynamic parts are based on JavaScript, which makes it very easy to learn.

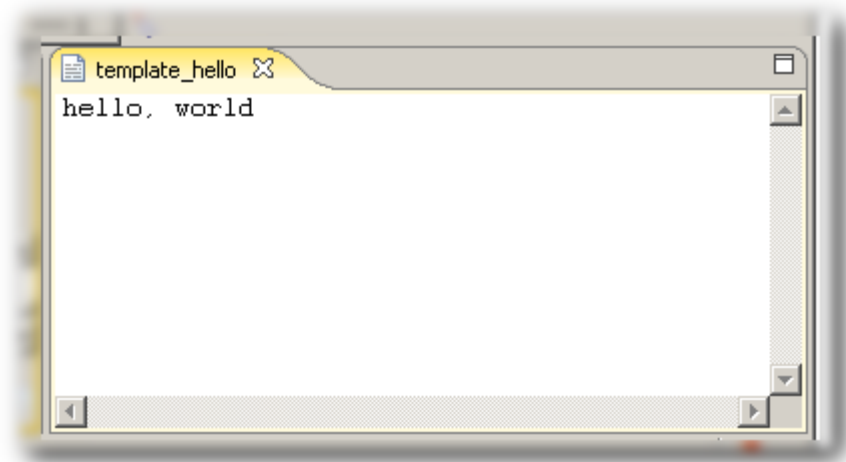
7.2 The Script File

Sign Live! CC recognizes JavaScript-based templates by their extension “.jst”. Scripts with this extension can be called like any other script and they will create a “string” type result.

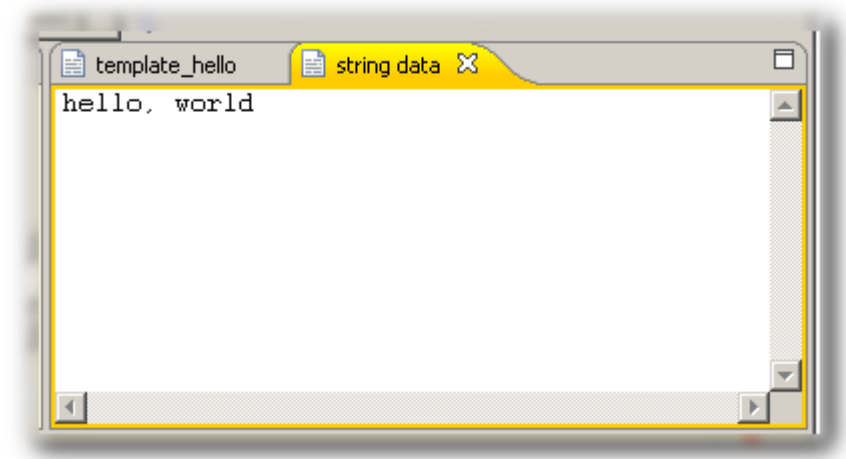
7.3 Hello, World

As always, a simple example to get started. Create a script “template_hello.jst” in the Script Manager with the following contents:

```
Hello, world
```



Save and close the document. After double clicking on the script to start it you should see the following:

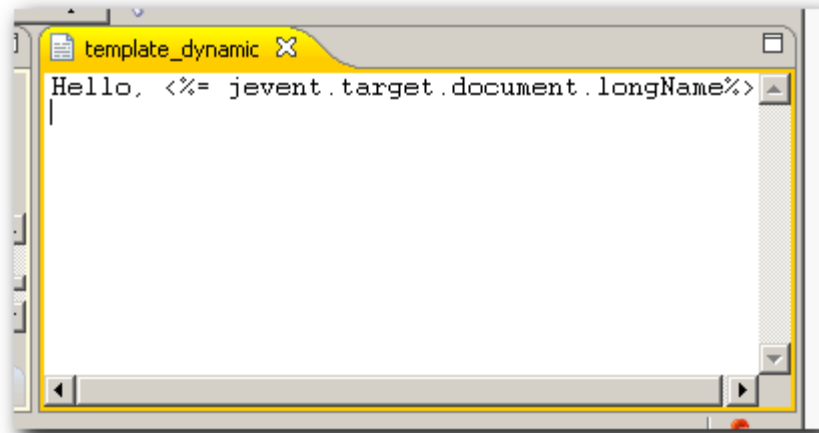


Impressed? No? OK, lets try something else...

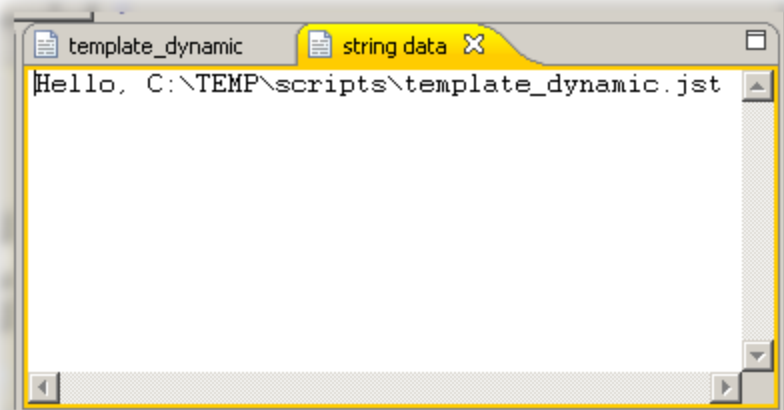
7.4 Dynamic Content - Calculations

What you could not see in the previous example is that the displayed content is not the content of your script, but that of the document that was generated by your script! This can be better seen using an example with dynamic contents. Create a script “template_dynamic.jst” with the following contents:


```
Hello, <%= jEvent.target.document.longName%>
```



After double clicking (a document should, of course, be open, otherwise you will get an error) you will see:



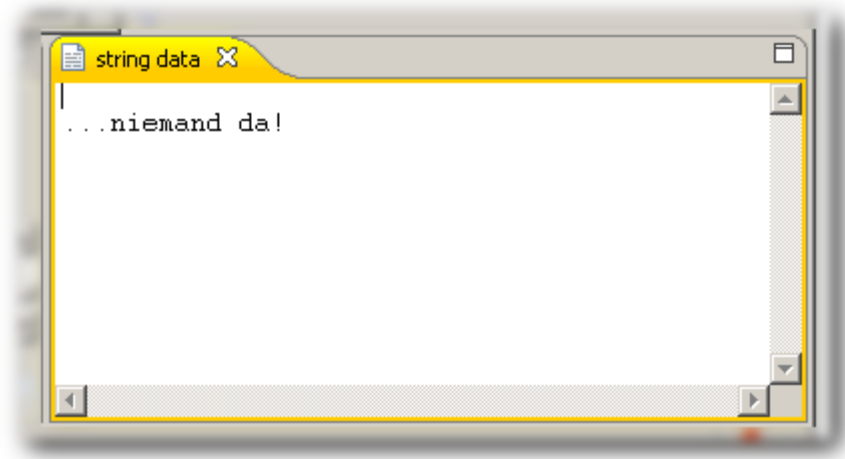
The calculated value for *jEvent.target.document.longName* is automatically inserted into the document!

7.5 Dynamic Content - Loops

This would all be very unspectacular if queries and loops could not be used. The syntax here is a little different:

```
<% if (jEvent.target != null) {%>  
Hello, <%= jEvent.target.document.longName%>  
<% } else { %>  
...niemand da!  
<% } %>
```

If you close all documents before running the script now, you will see the following upon running:



If a result should be calculated and replaced a “=” will need to follow “<%”. If a different code expression is embedded then the “=” will not be needed. The code in “<% .. %>” brackets is valid JavaScript code. The code can be interrupted any location that a valid calculation expression can be placed. A valid JavaScript must result when all the code pieces are put together. In the example above all the brackets need to be properly opened/closed.

Thats all you need to know for template programming!

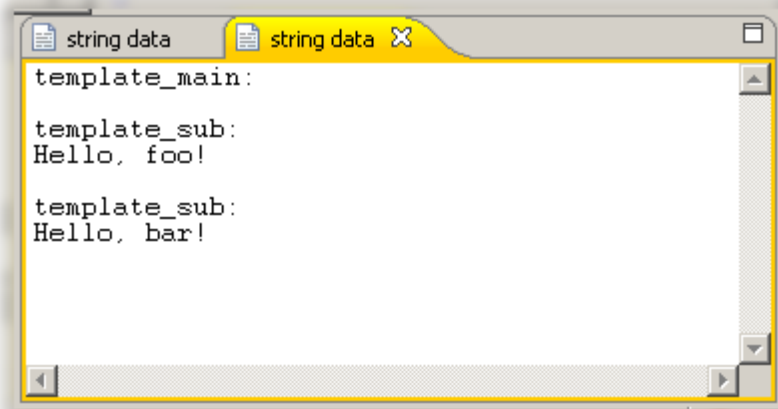
7.6 Parameter

Templates are normal scripts with a slightly different syntax and therefore support parameters. You can use this to define powerful text building blocks that you can parameterize with various contents.

To test this write the two following scripts, “template_main.jst” and “template_sub.jst”:

```
template_main:
<%= Script.callArgs("template_sub", "foo") %>
<%= Script.callArgs("template_sub", "bar") %>
<%= buddy %>
template_sub:
Hello, <%= buddy%>!
```

Start “template_main”. The result should look like this:



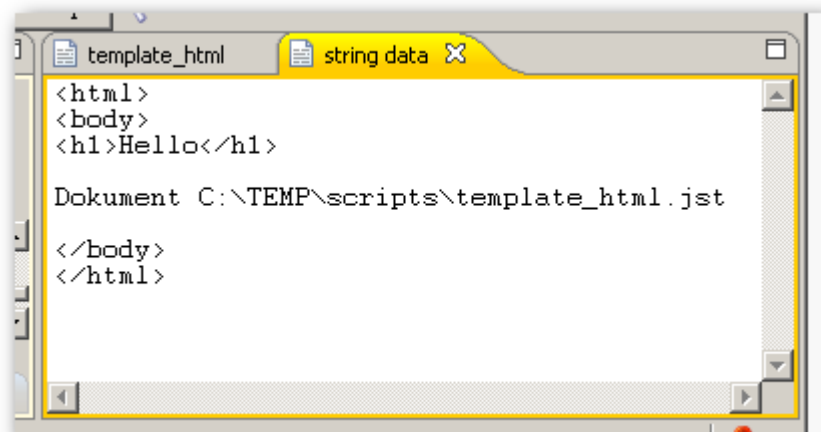
7.7 Generate HTML

With the presented syntax and a combination of scripts through calls you can generate complex documents. For this reason we would like to present another common application.

HTML-formatted texts are an obvious choice for template generation. The text structure is simple to create and the output looks good! Many of the internal documents and reports in Sign Live! CC are created using this technique. So, let's write an HTML document:

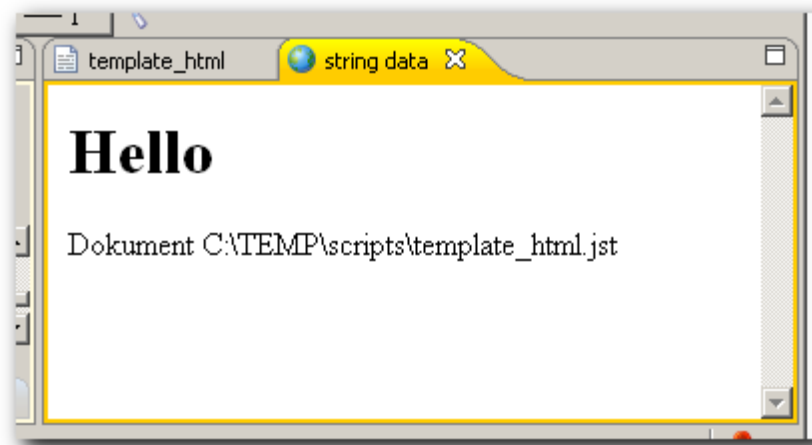
```
<html>  
<body>  
<h1>Hello</h1>  
<% if (jEvent.target != null) {%>  
Dokument <%= jEvent.target.document.longName%>  
<% } else { %>  
...and goodbye  
<% } %>  
</body>  
</html>
```

After starting you will see...



Ooops - that's not exactly the result we were looking for. It is HTML, but the result is not being displayed in the appropriate viewer. But we can fix that! We will add another variable to the template - a result type declaration.

```
<%@ contentType:text/html %>
<html>
<body>
<h1>Hello</h1>
<% if (jEvent.target != null) {%>
Dokument <%= jEvent.target.document.longName%>
<% } else { %>
...and goodbye
<% } %>
</body>
</html>
```



contentType allows for a mime type declaration for the data stream that the template generates. The caller - in this case the Script Manager itself - can review this information and use it accordingly.

In case you call a similar script in your integration yourself, you will need to handle the appropriate steps for creating the document and processors. But that should not be a problem after reading this tutorial.

7.8 PDF - for Advanced Users

Now that we can generate and typify text streams, whats stopping us from generating a PDF? Well, we can (for very exotic application scenarios...)

```
<%@ contentType:application/pdf %>\
\
%PDF-1.4
1 0 obj
<<
/Type /Catalog
/Pages 3 0 R
>>
endobj
2 0 obj
<<
/Producer (handcrafted)
/Creator (handcrafted)
>>
endobj
3 0 obj
<<
/Type /Pages
/Kids [4 0 R]
/Count 1
>>
endobj
4 0 obj
<<
/Type /Page
/MediaBox [0 0 595 841]
/Parent 3 0 R
/Contents 5 0 R
>>
endobj
5 0 obj
<<
/Length 41
>>

stream
q
0.5 g
0.5 G
100 600 100 100 re
f
Q
endstream
endobj
trailer
<<
/Size 5
/Root 1 0 R
/Info 2 0 R
>>
%%EOF
```


8. Advanced CodeExit Applications

8.1 Overview

CodeExits are the lynchpin for the dynamic expansion of Sign Live! CC. Anywhere a description is required for a configurable behavior, it is accomplished with a CodeExit. Examples include actions or Sign Live! CC calls through automation interfaces.

Here we will provide some additional information about how you can even more out of your system using this concept.

8.2 The Dynamic ExtensionPoint

8.2.1 Basis

Many of the system's ExtensionPoints des Systems require a Java class declaration with a specific interface. An instance of this class is then built and its method is called in accordance with the ExtensionPoint's semantic.

The ExtensionPoint *com.cabaret.application.startstops*, for example, requires an instance of the *de.intarsys.tools.component.IStartStop* type. When the application is started the method "start" is called; when it is closed the method "stop" is called.

For an ad-hoc expansion of the system you will, unfortunately, require a Java class - or? Of course the answer is "no". You can "script" an interface directly with help from the CodeExit declaration.

8.2.2 Syntax

For an interface's dynamic scripting the interface is entered directly into the element's "class" attribute. The sub-element "implementation" will then be required. Here an element "method" will be added to each

method to be implemented. The implementation of the methods will be described by a CodeExit.

Example:

```
<extension point="com.cabaret.application.startstops">
  <startstop
    class="de.intarsys.tools.component.IStartStop">
    <implementation>
      <method name="start">
        <perform type="JavaScript" source="
java.util.logging.Logger.getLogger('startstop').warning('i am
started');
        "/>
      </method>
      <method name="stopRequested">
        <perform type="JavaScript" source="
java.util.logging.Logger.getLogger('startstop').warning('i will
stop');
true;
        "/>
      </method>
      <method name="stop">
        <perform type="JavaScript" source="
java.util.logging.Logger.getLogger('startstop').warning('i am
stopped');
        "/>
      </method>
    </implementation>
  </startstop>
</extension>
```

A method element will be created for each of the interface's required methods. The arguments for the methods are passed on as arguments for the CodeExit. It is important to think about the method's return value in order to avoid ClassCastException or NullPointerException!

8.2.3 Multiple Interfaces

It is also possible to implement multiple interfaces using this technique. This will require that you enter the names of the interfaces separated by “;” in the “class” attribute.

You can create a “method” element for each method in each interface.

8.2.4 Important

In order to relieve the internal consistency mechanisms “always” implement the dynamic ExtensionPoint proxies “de.intarsys.tools.attribute.IAttributeSupport”. This will, for example, secure the object identity when switching to a different script language.

The interface's methods can “not” be re-defined by a “method” element.

8.2.5 Scripting Call Semantic

The ExtensionPoint proxy is a definite object that implements all declared interfaces. This proxy object is the receiver in the CodeExit call. You will receive an identical object on every call.

This allows you to store local variables in this object. An example thereof can be found in the next section.

8.3 ExtensionPoint Configuration Information

If an object is created for an ExtensionPoint, it can carry out further configurations itself, based on the elements. This requires that the object implements the interface

com.cabaret.platform.extension.IExtensionConfigurable. In the case of our dynamic ExtensionPoint declaration in the last section this means the multiple implementation of interfaces.

Example:

```

<extension point="com.cabaret.application.startstops">
  <startstop
    class="
      de.intarsys.tools.component.IStartStop;
      com.cabaret.platform.extension.IExtensionConfigurable"
    foo="bar">
    <implementation>
      <method name="configure">
        <perform type="JavaScript" source="
java.util.logging.Logger.getLogger('startstop').warning('i get
configured');
this.foo = Reflector.argAt(1).attributeValue('foo');
java.util.logging.Logger.getLogger('startstop').warning('foo is ' +
foo);
        "/>
      </method>
      <method name="start">
        <perform type="JavaScript" source="
java.util.logging.Logger.getLogger('startstop').warning(foo + ' is
started');
        "/>
      </method>
      <method name="stopRequested">
        <perform type="JavaScript" source="
java.util.logging.Logger.getLogger('startstop').warning(foo + ' will
stop');
true;
        "/>
      </method>
      <method name="stop">
        <perform type="JavaScript" source="
java.util.logging.Logger.getLogger('startstop').warning(foo + ' is
stopped');
        "/>
      </method>
    </implementation>
  </startstop>
</extension>

```

You should find the following information in the log upon start and finish:

```

[25.11.2007-15:26:11:302 ][WARNING ][startstop][main] i get
configured
[25.11.2007-15:26:11:333 ][WARNING ][startstop][main] foo is bar
[25.11.2007-15:26:11:942 ][WARNING ][startstop][main] bar is started
...
[25.11.2007-15:26:35:276 ][WARNING ][startstop][SWT GUI Thread] bar
will stop
[25.11.2007-15:26:35:276 ][WARNING ][startstop][SWT GUI Thread] bar
is stopped

```

