

Sign Live! cloud suite gears
incubator

Oktober 2025

intarsys GmbH

Sign Live! cloud suite gears incubator

Version 8.14

cloud suite gears

intarsys GmbH
Sign Live! cloud suite gears incubator
Version 8.14

All rights reserved
© 2019 intarsys GmbH
www.intarsys.de

Preface

- Author and company

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

- Trademarks

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

jPod is a trademark of intarsys consulting.

Sun, Java and JavaScript are trademarks of Oracle

Microsoft and Windows are trademarks of Microsoft Corporation.

- Who should read this book

This book provides both an overview of the product design and architecture and a reference for using the components and services.

So, this is the document for architects, developers and operators.

- 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

Contents

Preface	5
▪ Author and company	5
▪ Trademarks	5
▪ Who should read this book	5
▪ Reviews and comments	5
Contents	6
1. Overview	10
2. Keystore device	11
2.1 Overview	11
2.2 Custom device configuration	11
2.3 Built-in keystore device	11
2.3.1 Principal <i>Sign Live cloud suite gears Built-In CA</i>	12
2.4 DB-backed keystore device	12
2.4.1 Principal creation	12
2.4.2 Principal deletion	16
2.4.3 Private key security	16
2.5 Keystore pooling	16
2.6 Example	17
3. Modal UI	19
3.1 Overview	19
3.2 Configuration	19
3.3 Example	19
4. Shortcut definition	21
4.1 Overview	21
4.2 Configuration	21
4.3 Features	21
4.3.1 Syntax	21
4.3.2 Modifier keys	22
4.3.3 Special keys	22

4.3.4	Function keys	22
4.3.5	Other keys	23
5.	Widget visibility	24
5.1	Overview	24
5.2	Configuration	24
5.3	Example	26
6.	User defined icons	27
6.1	Overview	27
6.2	Fontawesome icons	27
6.3	Plugin	28
6.4	Example	28
7.	Embedded use	31
7.1	Overview	31
7.2	Setup	31
7.3	Protocol	32
7.3.1	noop protocol	32
7.3.2	windows protocol	33
7.3.3	Protocol messages	33
7.4	Protocol lifecycle	36
7.4.1	ready	36
7.5	Example	36
8.	Control overlay	39
8.1	Overview	39
8.2	Mechanics	39
8.3	Configuration	40
8.3.1	Properties	40
8.4	Example	41
9.	Signature shapes	43
9.1	Overview	43
9.2	Graphics model	43
9.2.1	Container	43
9.2.2	Shape	43
9.2.3	Position	44
9.2.4	Anchor	45
9.2.5	Size	46
9.2.6	Align	47
9.2.7	Relative coordinates	50
9.3	Shapes	50
9.3.1	Shape	51
9.3.2	Text	52
9.3.3	Icon	53
9.3.4	Rectangle	53
9.3.5	Ellipse	54
9.3.6	Path	54

9.4	The decorator	55
9.5	Examples	56
9.5.1	Simple rectangle	57
9.5.2	Multiple fonts	57
9.5.3	Multiple icons	59
10.	Additional crypto and security components	61
10.1	TLS client authentication with signer device	61
10.1.1	IKeyManagerProvider	61
10.2	OAuth 2.0 login to gears control panel	62
10.2.1	Backend configuration	62
10.2.2	Frontend configuration	64
11.	Blackening Editor	66
11.1	Overview	66
11.2	The Editor	66
11.3	Configuration	67
12.	Standalone operation	68
12.1	Overview	68
12.2	Installation	68
12.3	Operation	68
12.3.1	Shell application	68
12.3.2	Service	69
12.4	Configuration	70
12.5	Docker containerization	70
12.5.1	Ubuntu image with Tomcat	70

1. Overview

This book presents features of the product that are not yet fully implemented, designed or tested.

You may use the features and provide feedback, both to ensure that the features are improved and to make us aware that they are used and some customers care to make them fully fledged product features.

The features presented in this document are volatile and subject to change without notice. Be sure to not use this in production without prior interaction with our product team.

You can find sources and snippets for some of the incubation features in the "incubator" folder of the deployment.

2. Keystore device

2.1 Overview

In some cases, it may be desirable to work with softkeys, regardless if created from an in-house PKI or from a trust service provider.

To support such scenarios, we provide a special "KeystoreDevice".

2.2 Custom device configuration

You create the device as usual in the spring xml

spring XML fragment

```
<bean class="de.intarsys.security.device.keystore.device.KeystoreDeviceFactoryBean">
  <property name="id" value="soft" />
  <property name="keyStoreDirectoryName" value="${cloudsuite.config.shared}/keys" />
  <!--
  keyStorePassword=file password
  keyPassword=key password
  -->
  <property name="keyStorePassword" value="plain#ZmlsZSBwYXNzd29yZA==" />
</bean>
```

You must provide the unique **id** which is used for device selection later.

keyStoreDirectoryName defines a directory that is scanned on startup for all supported key stores.

The **keyStorePassword** must be the same for all stores in the directory.

2.3 Built-in keystore device

Gears comes with an optional built-in keystore device with ID "builtin". In order to activate it, please start gears with the Spring profile "builtinkeystore":

```
spring.profiles.active=builtinkeystore
```

This will activate the device "builtin@keystore", which currently contains the principals described in the following sections.

Be aware that this keystore is by no means considered secure, as all information to activate the contained keys is provided within the application itself.

2.3.1 Principal *Sign Live cloud suite gears Built-In CA*

This principal is backed by a self-signed CA certificate and can be used for issuing certificates within the application.

To refer to this principal within the device using a signer identifier, use the String “usage:ca”.

2.4 DB-backed keystore device

Gears comes with an optional database-backed keystore device implementation with ID “database”. In order to activate it, please start gears with the Spring profile “dbkeystore”.

```
spring.profiles.active=dbkeystore
```

This will activate the device “database@keystore”, which initially doesn’t contain any principals.

2.4.1 Principal creation

The DB keystore device supports dynamic creation of principals. In the course of this process, you will

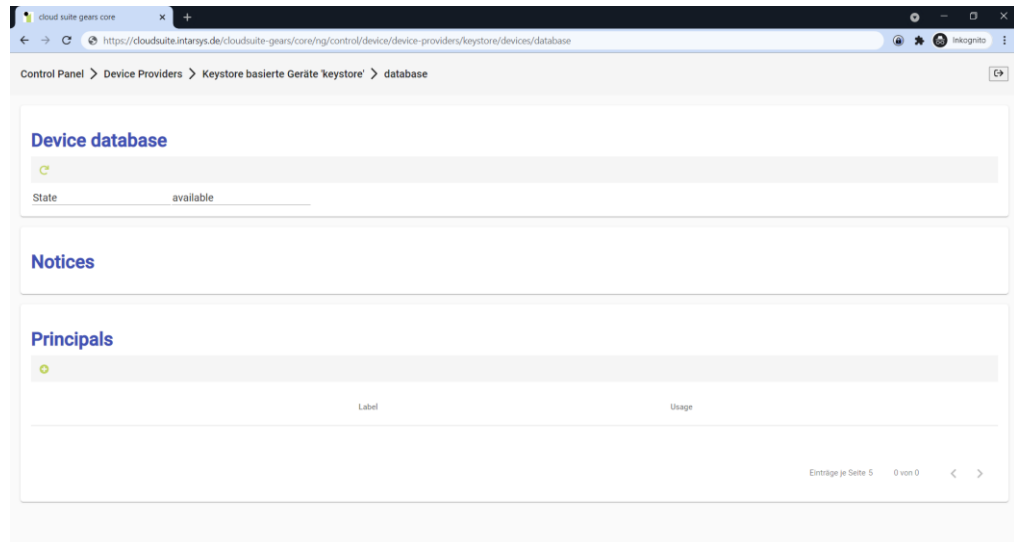
1. Generate a key pair,
2. get a corresponding certificate issued and
3. securely store these components in the database.

By default, key pairs are of size 2048 bits. You can interactively select the key purpose, subject data and validity period to be encoded into the certificate. The certificate itself will be signed by the built-in CA certificate, so make sure that you also activate the corresponding profile:

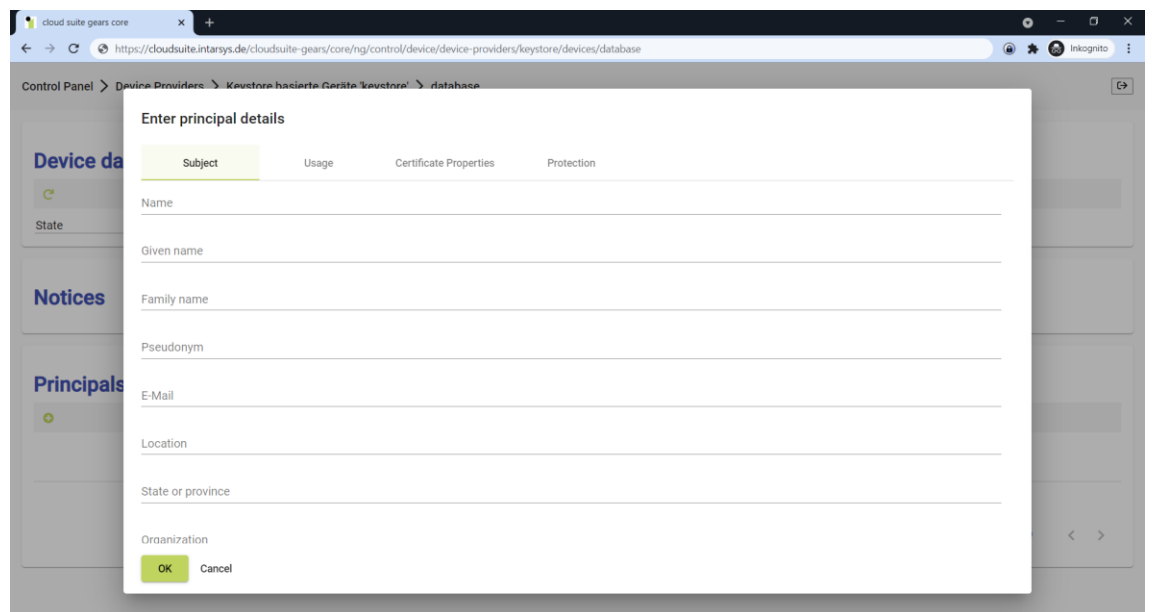
```
spring.profiles.active=builtinkeystore,dbkeystore
```

You’ll get access to principal generation by means of the gears control panel:

Keystore device

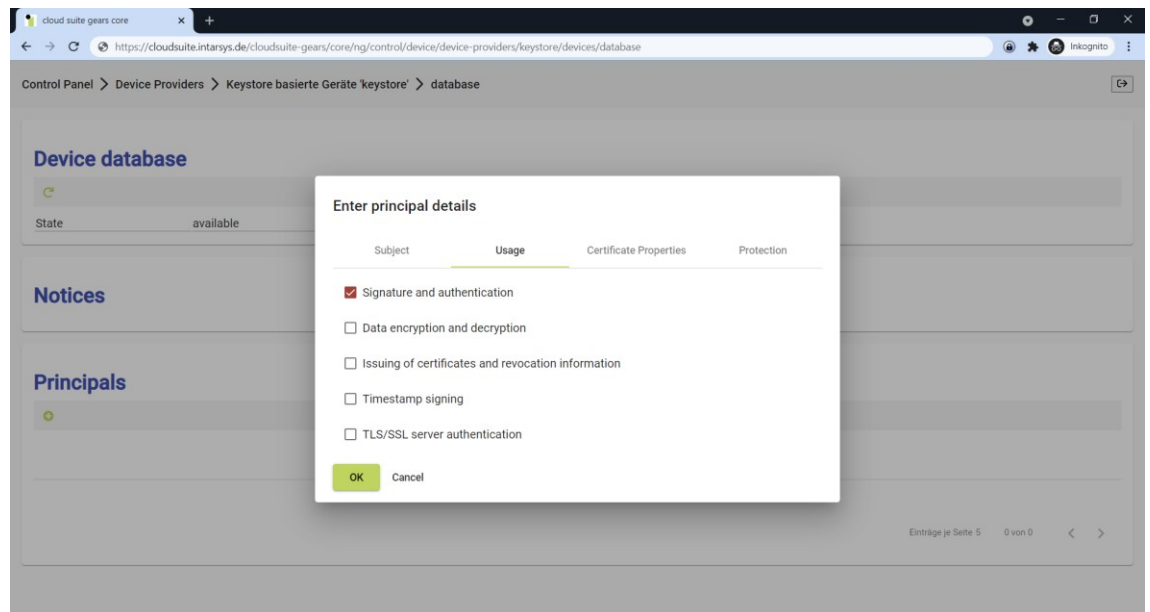


After clicking on the plus sign in the Principals section's toolbar, a dialog will open, initially displaying the "Subject" tab:



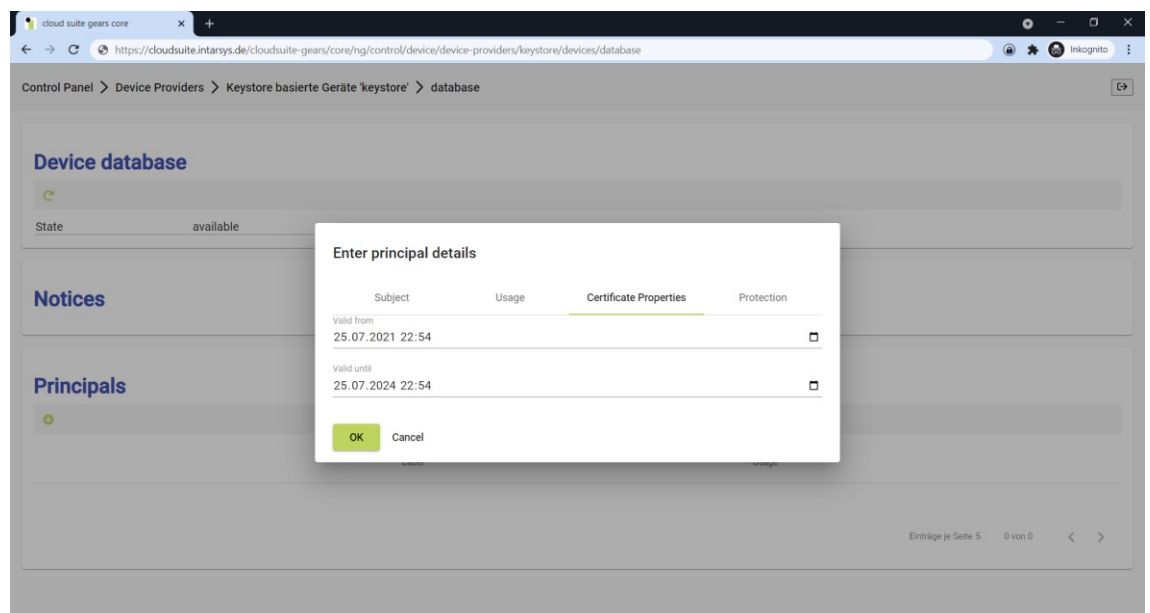
Here you can enter all the details going into the principal certificate's subject property.

On the next tab "Usage"...



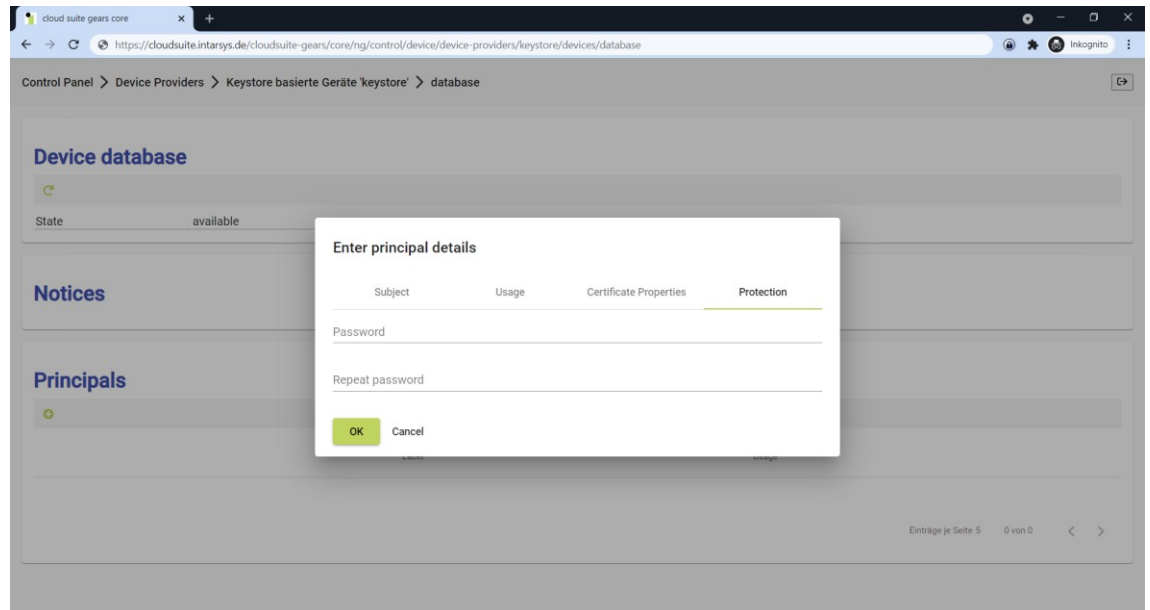
...you get the option to control the certificate's key usage and extended key usage extensions by selecting one or multiple predefined purposes.

On the "Certificate Properties" tab...



...you can control the validity span of the certificate being generated.

Security constraints regarding key protection are to be defined in the "Protection" tab:

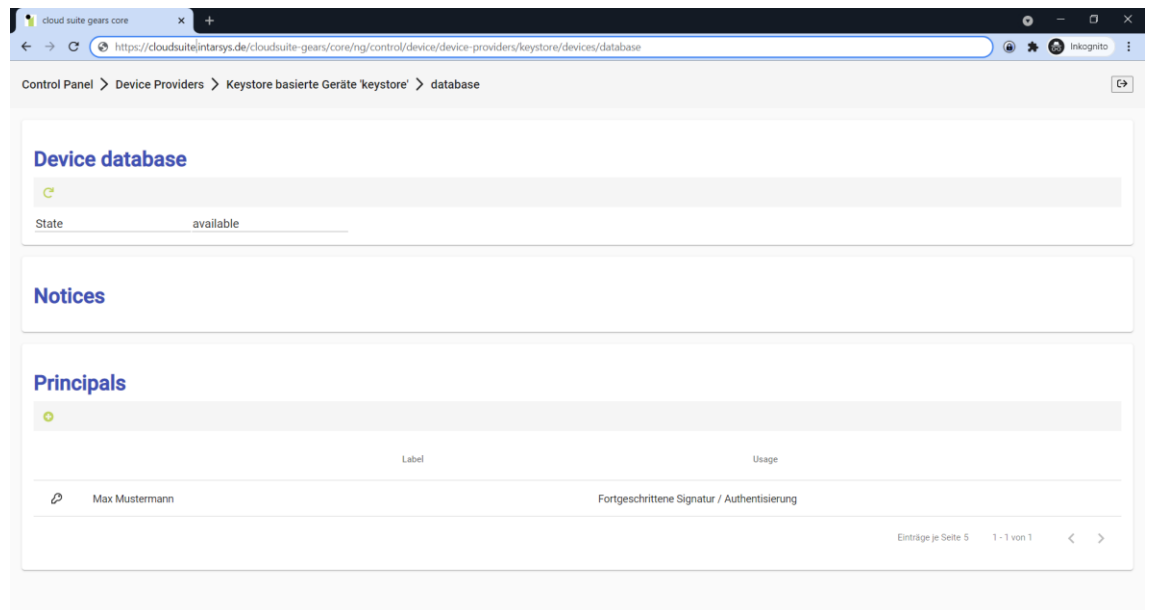


The entered password will be used to encrypt the generated private key before persisting it to the database. You'll need this password later in order to recover and use the key within the application.

Note on key protection:

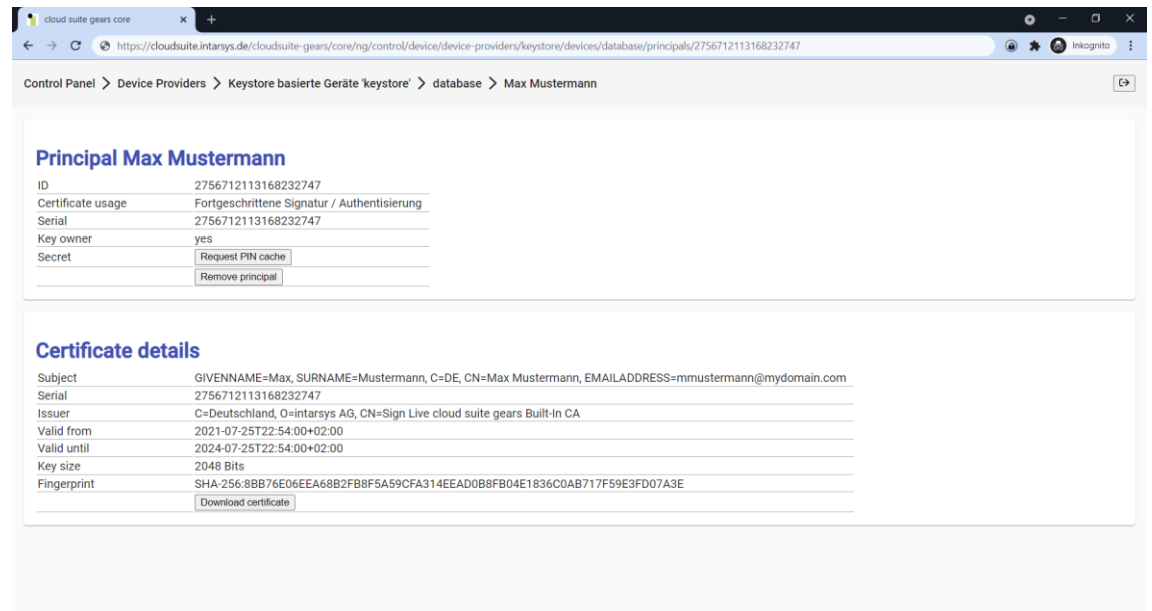
The encryption method used derives its secure key from the application's master key and the password entered during principal generation. Thus you have to make sure to **first** define a master password for the installation and **then** generate the required principals. If the master password is changed afterwards, no key recovery and use will be possible.

After confirming the dialog with "OK", keys and certificate will be generated. This may take a couple of seconds if done first time. Once the new principal is available, you'll find it in the device's "Principals" section:



2.4.2 Principal deletion

To delete a principal, navigate down to it...



...and hit the button “Remove principal”. After confirming the action in a next dialog, both the private key and the certificate will be deleted from the database.

2.4.3 Private key security

By default, the private keys are wrapped using AES encryption with 128-bit keys. In order to increase security, you may enforce AES 256-bit encryption by reconfiguring the key size to 32 Byte. See the following snippet:

```
<bean id="dbKeyStoreBasicCipher"
class="de.intarsys.tools.crypto.standard.JcaCipherFactory">
  <property name="encryptionAlgorithmTransformation" value="AES/CBC/PKCS5Padding" />
  <property name="keySize" value="32" />
</bean>
```

Be aware that this should be considered **before** creating any private keys, i.e. principals. Keys generated before changing the key size will not be recoverable by the application afterwards.

2.5 Keystore pooling

As already known from the use of Smartcards, you can also pool a Keystore device, allowing for centralized activation and deactivation.

A custom device can be created using the device provider (e.g. **deviceProvider.pool**) as a factory – but as this is a common task for pools, a special Spring factory bean is provided for Keystore pools.

Using the `de.intarsys.security.device.pool.keystore.KeystorePoolDeviceFactoryBean`, the task of creating a new Keystore pool is greatly simplified.

In Spring syntax, you would add this definition to your bean definition file.

Spring XML fragment

```
<bean class="de.intarsys.security.device.pool.keystore.KeystorePoolDeviceFactoryBean">
  <property name="id" value="demoKeystore" />
  <property name="autostart" value="true" />
  <property name="certificateFilter" value="alias=soft certificate" />
  <property name="timeout" value="30000" />
</bean>
```

2.6 Example

Based on this configuration

spring XML fragment

```
<bean class="de.intarsys.security.device.keystore.device.KeystoreDeviceFactoryBean">
  <property name="id" value="soft" />
  <property name="keyStoreDirectoryName" value="{cloudsuite.config.shared}/keys" />
  <!--
  keyStorePassword=file password
  keyPassword=key password
  -->
  <property name="keyStorePassword" value="plain#ZmlsZSBwYXNzd29yZA==" />
</bean>

<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="demoKeystore" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
value="de.intarsys.security.processor.signature.DocumentSignerFactory" />
      <entry key="documentSigner.args.digestSigner.factory"
value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device" value="soft@keystore" />
    </map>
  </property>
</bean>
```

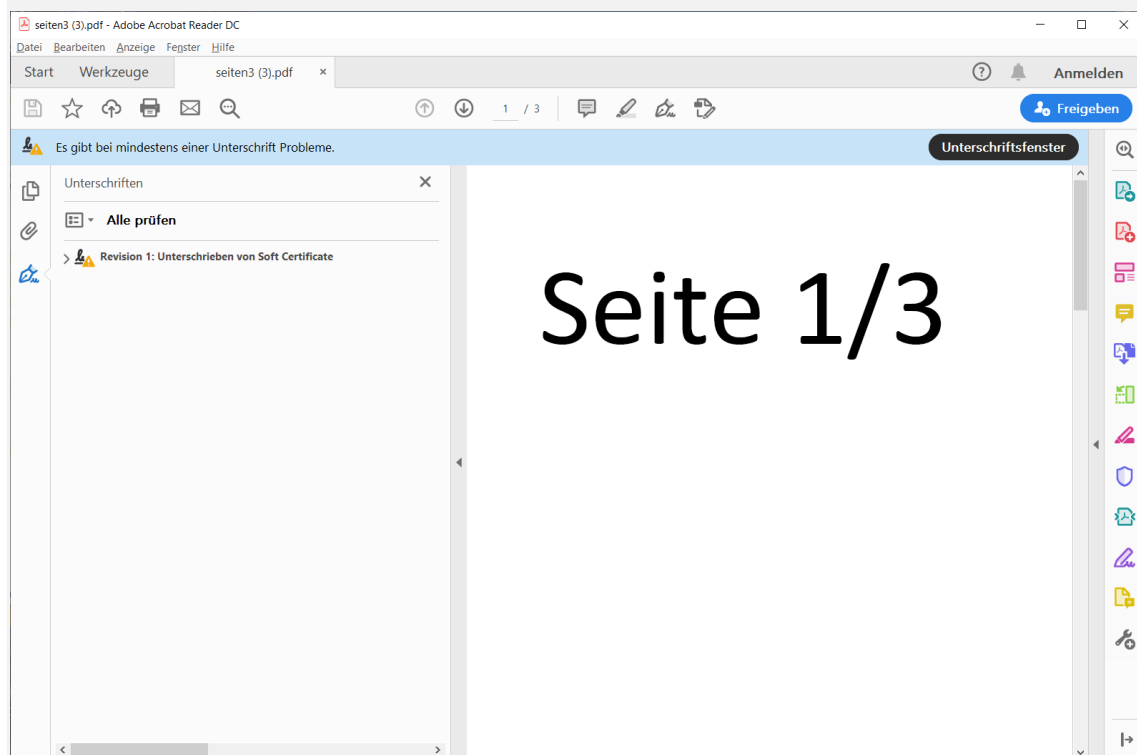
and the keystore from the "incubator/keystore device" folder you can now launch this request to the signer/create endpoint

service call

```
POST /cloudsuite-gears/core/api/v1/flow/signer/create HTTP/1.1
Content-Type: application/json
```

```
{
  "args": {
    "documentSigner": {
      "args": {
        "digestSigner": {
          "args": {
            "signerIdentifier": "alias=soft certificate",
            "signerPassword": "key password"
          }
        }
      }
    }
  },
  "configuration": "demoKeystore",
  ...
}
```

You can check the result in Adobe Reader. For sure the signature cannot be validated successfully as we have used a self-signed certificate, but it is technically a fully featured signature.



3. Modal UI

3.1 Overview

The default behavior for launching asynchronous processes in gears does not lock down the UI. Instead, a list of currently active processes is maintained, allowing (in theory) for multiple backend operations.

This behavior may be confusing for some users – they may need stricter guidance. For this scenario we provide now the "modal" processing.

3.2 Configuration

To enforce modal behavior, you simply add the property **modal** to an action definition.

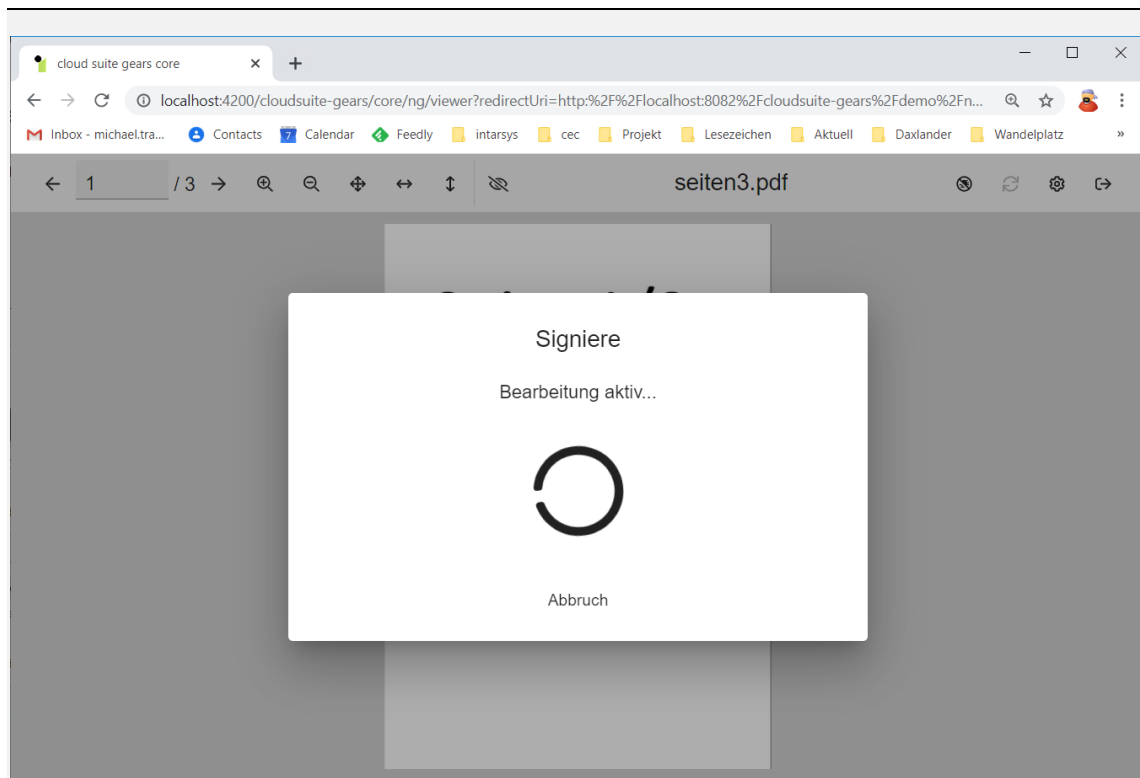
spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="demoModal" />
  <property name="widgets">
    <list>
      <w:widget parent="de.intarsys.widget.toolbar.additions" id="signModal"
label="signModal" icon="eye-slash">
        <w:on event="select" do="Sign">
          <entry key="modal" value="true" />
          <entry key="requireField" value="false" />
          <entry key="requireSignature" value="false" />
          <entry key="signerCreate" value="{flow.variables.signerCreate}" />
        </w:on>
      </w:widget>
    </list>
  </property>
</bean>
```

If you launch the action, the UI will block with a modal dialog – you can experience this behavior best when combined with an out-of-band signature (like demoOtp).

3.3 Example

This is the UI when using the above configuration with an out-of-band signer:



4. Shortcut definition

4.1 Overview

To improve user experience and usability, shortcuts are an important feature for any application.

We now provide a means to add shortcuts that is completely in line with the well-known widget configuration so far.

4.2 Configuration

You can add a shortcut as a special widget in the configuration

spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="demoShortcuts" />
  <property name="widgets">
    <list>
      <w:widget parent="de.intarsys.widget.shortcuts" label="press me">
        <w:property name="key" value="ctrl+x" />
        <w:on event="select" do="Alert">
          <entry key="message" value="hello!" />
        </w:on>
      </w:widget>
    </list>
  </property>
</bean>
```

Pressing "ctrl-x" should present you a browser alert dialog.

4.3 Features

The implementation is based on the popular "mousetrap" library (see <https://craig.is/killing/mice>) and detailed documentation can be found there.

We will summarize the most important features here.

4.3.1 Syntax

Keys are expressed as an "+" separated combination of key tokens

`ctrl+shift+x`

4.3.2 Modifier keys

These are keys that are pressed in combination with others. We have

- shift
- ctrl
- alt
- meta
- mod

This is a special modifier that maps to "command" on Mac and "ctrl" on other platforms.

4.3.3 Special keys

Special keys can be denoted by their names

- backspace
- tab
- enter
- return
- capslock
- esc
- escape
- space
- pageup
- pagedown
- end
- home
- left
- up
- right
- down
- ins
- del
- plus

4.3.4 Function keys

For function keys, simply use "f" and the number:

f12

4.3.5 Other keys

All other keys are referenced by themselves ("k" is "k").

5. Widget visibility

5.1 Overview

To make a view configuration more versatile, we now provide public access to a widget's visibility.

As widgets are hierarchically organized, this can for example be used to simply show or hide the toolbar via configuration, arguments or at runtime.

5.2 Configuration

This configuration is slightly more complex to show the effect of combining some of our incubation features

spring XML fragment

```

<property name="widgets">
  <list>
    <w:widget id="de.intarsys.widget.toolbar">
      <w:property name="visible" value="{flow.variables.widgets.toolbar.visible}" />
    </w:widget>
    <w:widget id="de.intarsys.widget.sidebar">
      <w:property name="visible" value="false" />
    </w:widget>
    <w:widget parent="de.intarsys.widget.shortcuts" label="Toolbar
aktivieren/deaktivieren">
      <w:property name="key" value="f8" />
      <w:on event="select" do="SetVisibility">
        <entry key="mode" value="toggle" />
        <entry key="widget" value="de.intarsys.widget.toolbar" />
      </w:on>
    </w:widget>
  </list>
</property>

```

JSON fragment

```

{
  "widgets": [
    {
      "id": "de.intarsys.widget.toolbar",
      "properties": {
        "visible": "${flow.variables.widgets.toolbar.visible}"
      }
    },
    {
      "id": "de.intarsys.widget.sidebar",
      "properties": {
        "visible": "false"
      }
    },
    {
      "id": "de.intarsys.widget.sidebar",
      "parent": "de.intarsys.widget.shortcuts",
      "callbacks": {
        "select": {
          "factory": "SetVisibility",
          "args": {
            "mode": "toggle",
            "widget": "de.intarsys.widget.toolbar"
          }
        }
      },
      "properties": {
        "key": "f8"
      }
    }
  ]
}

```

First, to address the visibility of a widget, use the **visible** property in your configuration. In this configuration we have mixed a static declaration along with a variable based declaration to show the possibility of defining the visibility from your client call.

Next, we have configured two shortcuts, one to toggle visibility of the toolbar and one to leave the app even when the toolbar is not visible.

5.3 Example

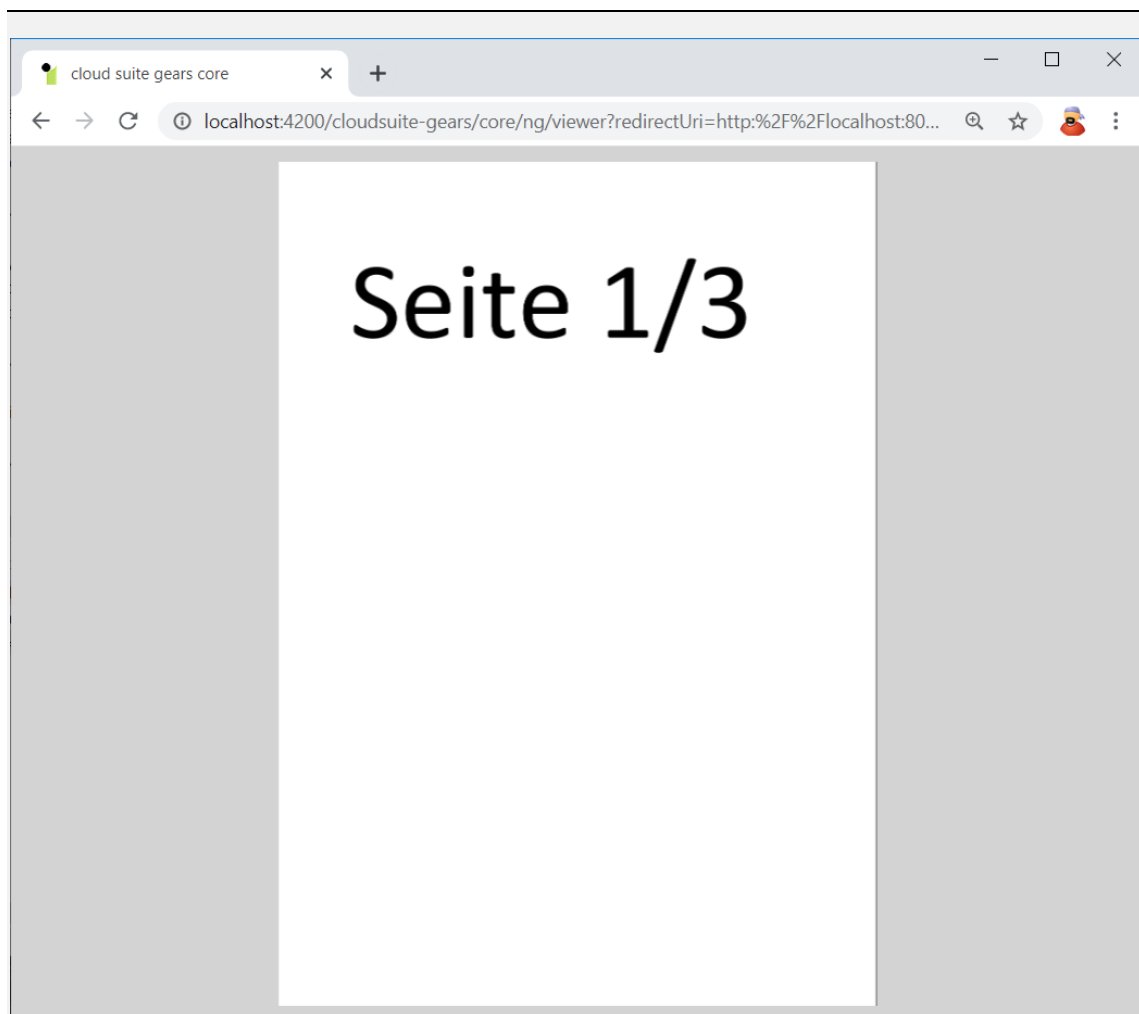
For the toolbar you need now information from your client call in the "variables" property of your call like so

service call

```
POST /cloudsuite-gears/core/api/v1/flow/viewer/create HTTP/1.1
Content-Type: application/json

{
  ...
  "variables": {
    "widgets": {
      "toolbar": {
        "visible": false
      }
    }
  },
  "configuration": "demoVisible",
  ...
}
```

You see that – you see no toolbar...



Pressing "F8" will toggle the toolbar, "Ctrl+X" will leave the component.

6. User defined icons

6.1 Overview

Gears comes with a set of predefined icons that are very common or are already used in the application.

Adding new icons currently involves always program changes – this is due to the way angular packaging and deployment is designed.

On the other hand, adding all possible icons is not desirable as this will put a download burden on all installations.

Here we present a solution for those that really require other icons without the global footprint. It is based on a dynamic plugin technique that will be used in other application areas, too.

6.2 Fontawesome icons

The iconography of gears is based on Fontawesome™, a business specialized in icons.

To see a catalog of the available icons see

<https://fontawesome.com/icons?d=gallery>

The icons are categorized in the following subsets

Name	
de.intarsys.plugin.fontawesome.icons.FreeSolid	Available as plugin
de.intarsys.plugin.fontawesome.icons.FreeRegular	Loaded by default
de.intarsys.plugin.fontawesome.icons.FreeBrands	Available as plugin
de.intarsys.plugin.fontawesome.icons.ProRegular	Available as plugin
de.intarsys.plugin.fontawesome.icons.ProSolid	Available as plugin
de.intarsys.plugin.fontawesome.icons.ProLight	Available as plugin
de.intarsys.plugin.fontawesome.icons.ProDuotone	Available as plugin

To use one of the icons of a specific subset, you must ensure that the respective plugin is loaded in your configuration.

Be sure to respect the icon reference syntax to resolve the correct icon!

6.3 Plugin

A plugin is defined along the configuration for your viewer. You need to assign the list of required plugins to the "plugins" property. A plugin is defined in the "PluginSpec".

spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="..." />
  <property name="plugins">
    <list>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.fontawesome.icons.FreeSolid" />
      </bean>
      ...
    </list>
  </property>
  ...
</bean>
```

6.4 Example

This configuration declares all available icon plugins and installs toolbar items that use an icon of each of them.

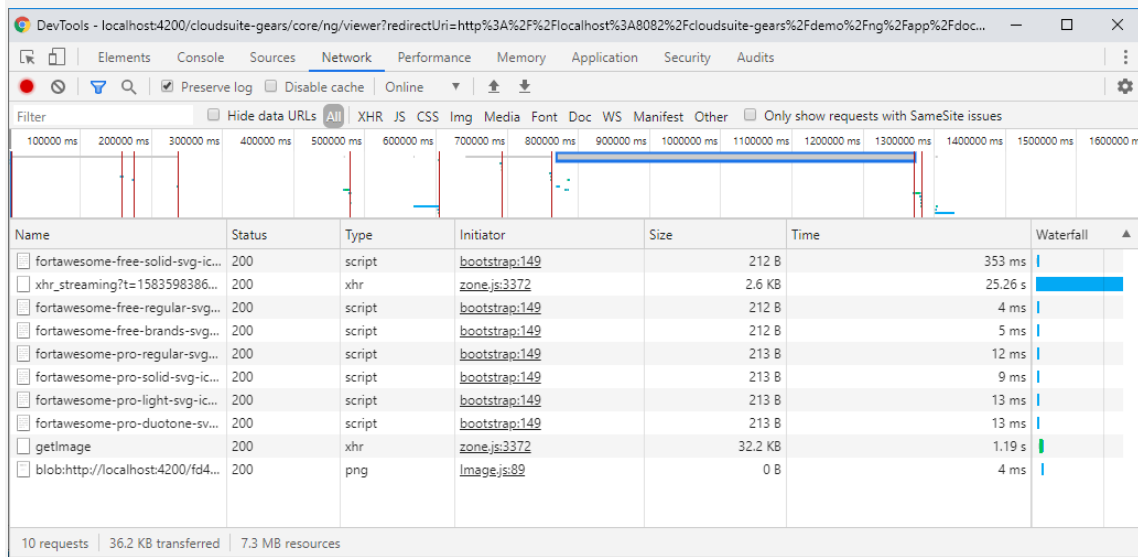
spring XML fragment

```

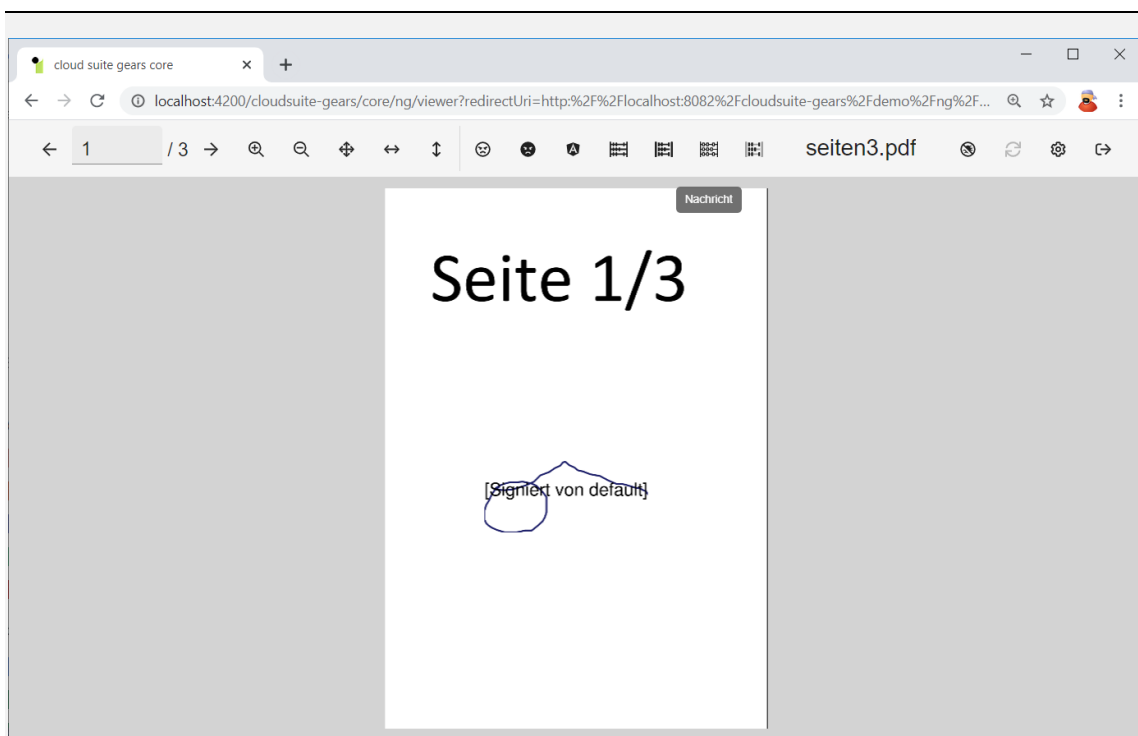
<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="demoPlugin" />
  <property name="plugins">
    <list>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.fontawesome.icons.FreeSolid" />
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.fontawesome.icons.FreeRegular"
/>
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.fontawesome.icons.FreeBrands" />
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.fontawesome.icons.ProRegular" />
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.fontawesome.icons.ProSolid" />
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.fontawesome.icons.ProLight" />
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.fontawesome.icons.ProDuotone" />
      </bean>
    </list>
  </property>
  <property name="widgets">
    <list>
      <w:widget parent="de.intarsys.widget.toolbar.additions" icon="far:angry">
        <w:on event="select" do="Alert">
          <entry key="message" value="loaded FreeRegular" />
        </w:on>
      </w:widget>
      <w:widget parent="de.intarsys.widget.toolbar.additions" icon="fas:angry">
        <w:on event="select" do="Alert">
          <entry key="message" value="loaded FreeSolid" />
        </w:on>
      </w:widget>
      <w:widget parent="de.intarsys.widget.toolbar.additions" icon="fab:angular">
        <w:on event="select" do="Alert">
          <entry key="message" value="loaded FreeBrands" />
        </w:on>
      </w:widget>
      <w:widget parent="de.intarsys.widget.toolbar.additions" icon="far:abacus">
        <w:on event="select" do="Alert">
          <entry key="message" value="loaded ProRegular" />
        </w:on>
      </w:widget>
      <w:widget parent="de.intarsys.widget.toolbar.additions" icon="fas:abacus">
        <w:on event="select" do="Alert">
          <entry key="message" value="loaded ProSolid" />
        </w:on>
      </w:widget>
      <w:widget parent="de.intarsys.widget.toolbar.additions" icon="fal:abacus">
        <w:on event="select" do="Alert">
          <entry key="message" value="loaded ProLight" />
        </w:on>
      </w:widget>
      <w:widget parent="de.intarsys.widget.toolbar.additions" icon="fad:abacus">
        <w:on event="select" do="Alert">
          <entry key="message" value="loaded ProDuotone" />
        </w:on>
      </w:widget>
    </list>
  </property>
</bean>

```

As a result, when starting the viewer, you can see in the network tab of your browser developer tools that the plugin data is loaded separately from the rest of the application.



The visual appearance of your viewer should be like this



7. Embedded use

7.1 Overview

Gears components are primarily designed for standalone use in your web application flow.

That said, a lot of use cases involve the integration of the components tightly in a more rigid container.

To support these use cases, we add a more low-level integration layer. While the initial use case may be best named "process component integration", we now talk about "UI component integration".

7.2 Setup

The component API is initialized by a dedicated plugin. You must add the plugin declaration to your configuration to use it. By default, the component API does nothing.

Plugin ID

```
de.intarsys.plugin.ComponentApi
```

Example

spring XML fragment

```

<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="demoEmbedded" />
  <property name="plugins">
    <list>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.ComponentApi" />
        <property name="args">
          <map>
            <entry key="protocol" value="windows" />
            <entry key="targetOrigin" value="*" />
          </map>
        </property>
      </bean>
      ...
    </list>
  </property>
</bean>

```

JSON fragment

```

{
  "plugins": [
    {
      "factory": "de.intarsys.plugin.ComponentApi",
      "args": {
        "protocol": "windows",
        "targetOrigin": "*"
      }
    }
  ]
}

```

Plugin args

Property	Description
protocol	The protocol ID to be used. Default "noop"
...	More arguments specific to the respective protocol

7.3 Protocol

7.3.1 noop protocol

The "noop" protocol ensures that **no** communication from or to the application takes place. This simplifies security considerations in environments where no communication is needed.

This is the **default**, meaning that you **must** configure the plugin in order to get real component interaction.

noop protocol args

Property	Description
protocol	"noop"

7.3.2 windows protocol

This protocol relies on the plain "window.postMessage" functions.

After configuring the plugin with this protocol, the viewer has the ability to interact with a container window. You can now for example put the viewer component in an iFrame and control its behavior.

The incoming messages are fetched from the "window" property of the application. The outgoing messages are sent to either the "window.parent" or "window.owner".

windows protocol args

Property	Description
protocol	"windows"
targetOrigin	The "targetOrigin" argument to use when calling the window.postMessage function. Default "*"
acceptOrigin	If set, only messages matching this origin are accepted. Default null
strictSource	Flag if we accept messages from only the window that initially opened the application. Default: true

The protocol messages outlined below are sent in the "data" property of the JavaScript MessageEvent.

7.3.3 Protocol messages

The communication is based on protocol messages. Protocol messages are transported as plain JavaScript objects.

All protocol messages support the following properties

Property	Description
type	The type of the message.

	The currently supported types are explained in the chapters below.
--	--

Your code must be prepared for changes in the protocol, especially to ignore new upcoming types of events or new properties!

7.3.3.1 TriggerMessage

A TriggerMessage can be used from the container to start some behavior in the application.

The target of a TriggerMessage is either a widget or an action. Upon receipt of the message, the target is looked up or created and execution is scheduled. After execution, the outcome is communicated to the container as a ReplyMessage **if** an "id" is contained in the initiating TriggerMessage.

TriggerMessage properties

Property	Description
type	"trigger"
id	An optional unique id for the message. This can be used for bi-directional communication (in request/reply scenarios)
sendResult	Flag indicating if in case of success the complete result should be sent to the container. This may cause large serialization objects. Default: false
widget	A widget reference if a widget should be triggered.
action	An action or action reference if an action should be triggered.

Example: Trigger from a container

JavaScript code

```

onEcho() {
  this.postMessage({
    type: 'trigger',
    id: 'echo',
    sendResult: true,
    action: {
      type: "Echo",
      message: "hello world"
    }
  });
}

```

7.3.3.2 ReplyMessage

A ReplyMessage is sent after the process spawned by a TriggerMessage has completed or failed **if** the TriggerMessage has held an **id** property.

ReplyMessage properties

Property	Description
type	"reply"
id	The id of the originating TriggerMessage.
result	The serialized result, if the process completed successfully and "sendResult" was requested – an empty object otherwise.
error.code	An error code, in case the process failed. This is "cancel" in case of cancellation.
error.message	An error message, in case the process failed.

Example: Receive reply in a container

JavaScript code

```

receiveMessage(message: any) {
  const data = message.data;
  if (data.type === 'reply') {
    if (data.id === 'ok' || data.id === 'cancel') {
      this.router.navigate(['/app', 'documents'], { queryParams: { cs_conversation: this.conversation } });
    }
  }
  if (data.type === 'event') {
    // handle event
  }
}

```

7.3.3.3 EventMessage

An EventMessage is sent to notify the container of state changes in the component.

EventMessage properties

Property	Description
type	"event"
event	The event type, e.g. "ready"
source	An optional event source
args	Optional event args

Example: Receive event in a container

JavaScript code

```

receiveMessage(message: any) {
  const data = message.data;
  if (data.type === 'reply') {
    // handle reply
  }
  if (data.type === 'event') {
    // handle event
  }
}

```

7.4 Protocol lifecycle

The gears component will publish events that allow to synchronize the lifecycle.

7.4.1 ready

After the component is completely launched it will publish an EventMessage with event "ready". You can use this as an indication that you can safely send TriggerMessages now.

7.5 Example

We have provided a simple configuration that can be used together with an example container from the "demo" application.

spring XML fragment

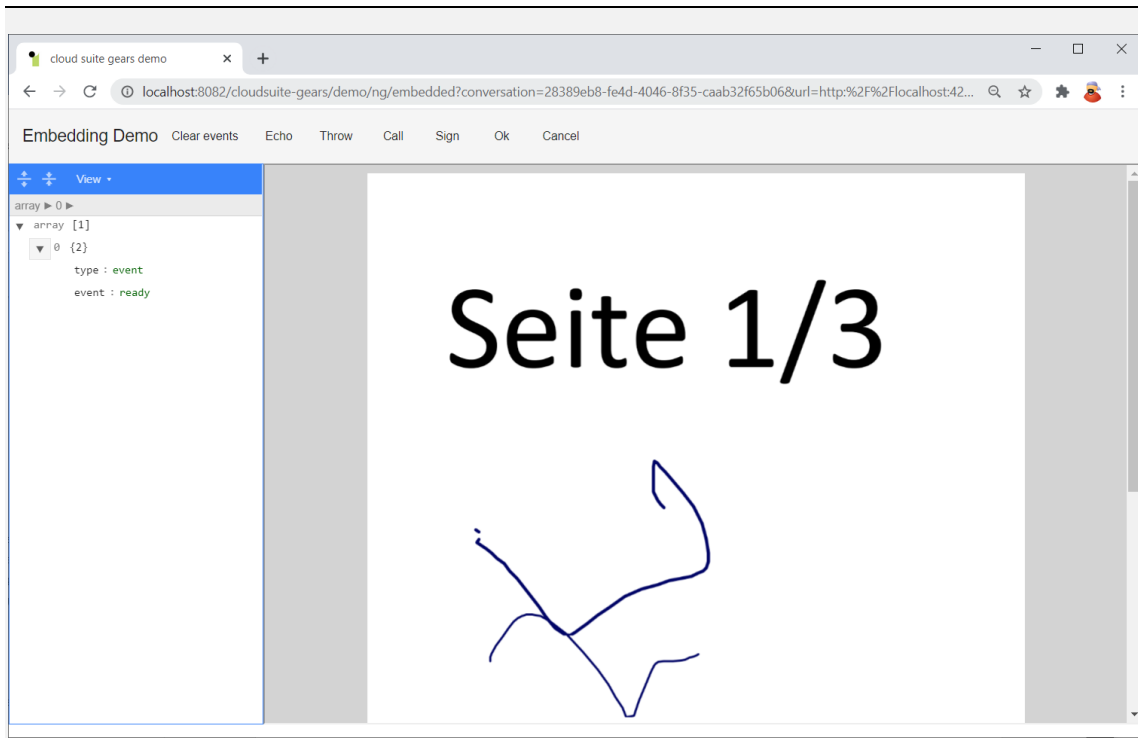
```

<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="demoEmbedded" />
  <property name="plugins">
    <list>
      <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
        <property name="factory" value="de.intarsys.plugin.ComponentApi" />
        <property name="args">
          <map>
            <entry key="protocol" value="windows" />
          </map>
        </property>
      </bean>
      ...
    </list>
  </property>
  <property name="widgets">
    <list>
      <w:widget id="de.intarsys.widget.toolbar">
        <w:property name="visible" value="false" />
      </w:widget>
      <w:widget id="my.widget.sign" register="my.widget.sign">
        <w:on event="select" do="Sign">
          <entry key="requireField" value="true" />
          <entry key="requireSignature" value="true" />
          <entry key="pageRange" value="current" />
          <entry key="signerCreate" value="{flow.variables.signerCreate}" />
        </w:on>
      </w:widget>
    </list>
  </property>
</bean>

```

It contains a "virtual" widget that is not directly used in the component. Instead, it is referenced from your container code.

You can try the embedded demo by selecting "demoEmbedded" for the viewer configuration and then hold the "ctrl" key while pressing "View" in the demo.



We have a container window from the demo app that holds an iFrame with the viewer (w/o toolbar).

You can trigger the widgets from the buttons at the top, sending a "postMessage" to the component that translates to a widget or action trigger. For example, after pressing "Sign" you can start defining the signature rectangle in the viewer component.

Posting from the client is simply implemented as

JavaScript code

```
postMessage(message: any) {
  this.tref.nativeElement.contentWindow.postMessage(message, this.urlString);
}
```

accessing the "window.postMessage" API.

The message is defined as

JavaScript code

```
onSign() {
  this.postMessage({
    type: 'trigger',
    widget: 'my.widget.sign'
  });
}
```

The type must be "trigger", the **widget** property holds the name of the targeted widget. That's it.

The left part is for showing the messages that are posted back as replies or events from the component.

8. Control overlay

8.1 Overview

Interaction with the toolbar may be cumbersome in some circumstances and disturb user experience.

While using keyboard shortcuts for experienced users may be the best way to support usage, bringing the controls nearer to the user may add up to application acceptance.

Many applications today choose to transparently add controls within the document viewing area to minimize mouse interaction. The control overlay will add such a feature to the gears viewer.

8.2 Mechanics

The overlay will add itself invisibly to the rendering queue. If the mouse reaches the sensitive area at the top or bottom of the document viewer, the toolbar will get visible.



If you leave the sensitive area, the toolbar will be hidden again.

If you have a nested toolbar, leaving the sensitive area is not enough – you must click outside of the toolbar to close it.

8.3 Configuration

Adding the control overlay is as simple as defining an overlay within your viewer configuration. Within this overlay widget you can define the dynamic toolbar just as you are used to for the application toolbar.

The identifier for the overlay is "ControlOverlay".

The overlay implementation uses a child widget named "toolbar" to define the dynamic toolbar widgets.

As an additional feature you can use a one-level deep nesting within this "toolbar" child.

8.3.1 Properties

activationBand	
number	Denotes the area that is waiting for interaction. A DIN/A4 PDF page has ca. 595 x 840 points. -1 covers the whole screen.

	Default: 20
offset	
number	Position of the overlay a offset to the upper and lower viewer border. Default: 20
hideAfter	
number	Controls disappear if used or automatically after given millis. Default: 3000
icon.padding	
number	The padding of the icon to the rectangle bounds. Default: 4
icon.size	
number	The size of the icon. Default 24
icon.margin	
number	The margin between icons. Default: 2
icon.fill	
text	The fill color of the icon. Default: black
icon.opacity	
number	The opacity of the icon. Default: 1
box.fill	
text	The fill color of the bounding box. Default: gray
box.opacity	
number	The opacity of the bounding box: Default: 0.1

8.4 Example

spring XML fragment

```

<w:widget parent="de.intarsys.widget.renderer.overlays" id="control"
type="ControlOverlay">
  <w:property name="icon.fill" value="black"/>
  <w:property name="box.fill" value="green"/>
  <w:widget id="toolbar">
    <w:widget icon="arrow-left">
      <w:on event="select" do="SelectPagePrevious" />
    </w:widget>
    <w:widget icon="arrow-right">
      <w:on event="select" do="SelectPageNext" />
    </w:widget>
    <w:widget icon="search">
      <w:widget icon="search-plus">
        <w:on event="select" do="ZoomIn" />
      </w:widget>
      <w:widget icon="search-minus">
        <w:on event="select" do="ZoomOut" />
      </w:widget>
      <w:widget icon="arrows-v">
        <w:on event="select" do="ZoomPageHeight" />
      </w:widget>
      <w:widget icon="arrows-h">
        <w:on event="select" do="ZoomPageWidth" />
      </w:widget>
      <w:widget icon="arrows">
        <w:on event="select" do="ZoomPage" />
      </w:widget>
    </w:widget>
  </w:widget>
</w:widget>

```

This will add a nested toolbar to your viewer giving direct access to the most often used viewer manipulations, using a nested toolbar. The default color is changed to green.



9. Signature shapes

9.1 Overview

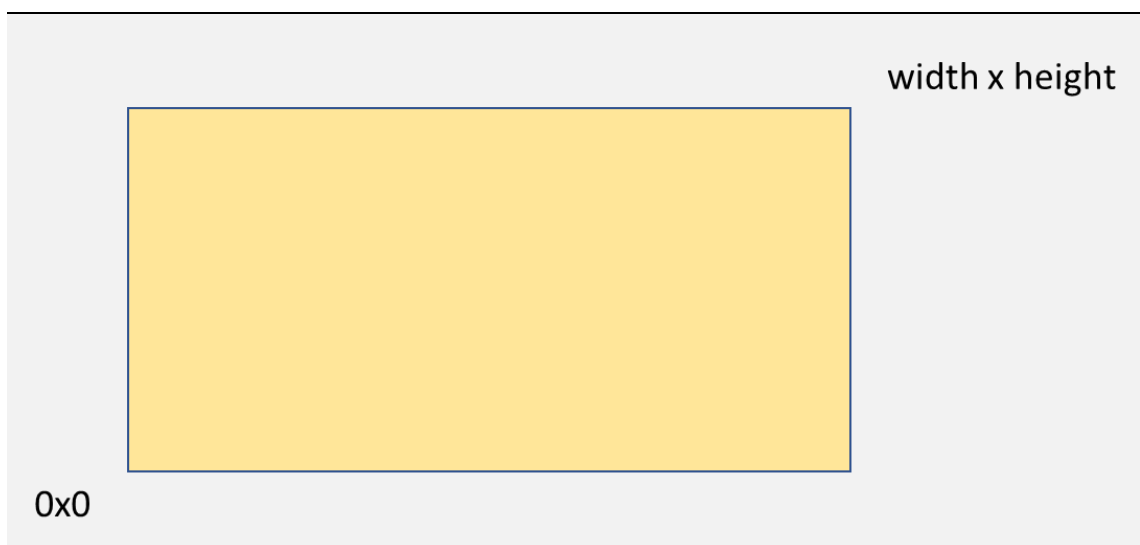
While simple and sufficient for most applications, the "ExtendedDecorator" has not many features and does not allow for detailed appearance definition.

Some use cases require more complex designs. This implementation is based on full-fledged vector graphics kernel. It allows very detailed manipulation of a user defined number of different shapes, while trying to hide associated complexities.

9.2 Graphics model

9.2.1 Container

The base for all drawing applied is the rectangle defined by the signature field.



Its lower left corner is the origin, the width and height define the drawable area. All coordinates are in PDF user space.

9.2.2 Shape

Now we can start adding shapes. A shape is anything that can be drawn on our signature field, like

- Text
- Icon
- Rectangle
- Lines



```
{  
  "type": "text",  
  "text": "Hi world!"  
}
```

Shapes are defined by simple objects. In this example we define a simple text shape.

A shape has always these basic properties

- type (required)

What type of shape do you want to draw. The available types are described below.

The other general properties are described in the chapters below.

The shape specific properties are described in the respective shape chapters.

9.2.3 Position

The first step is to position the shape within the container.



Here we start with a text object that is drawn at position 20 * 10.

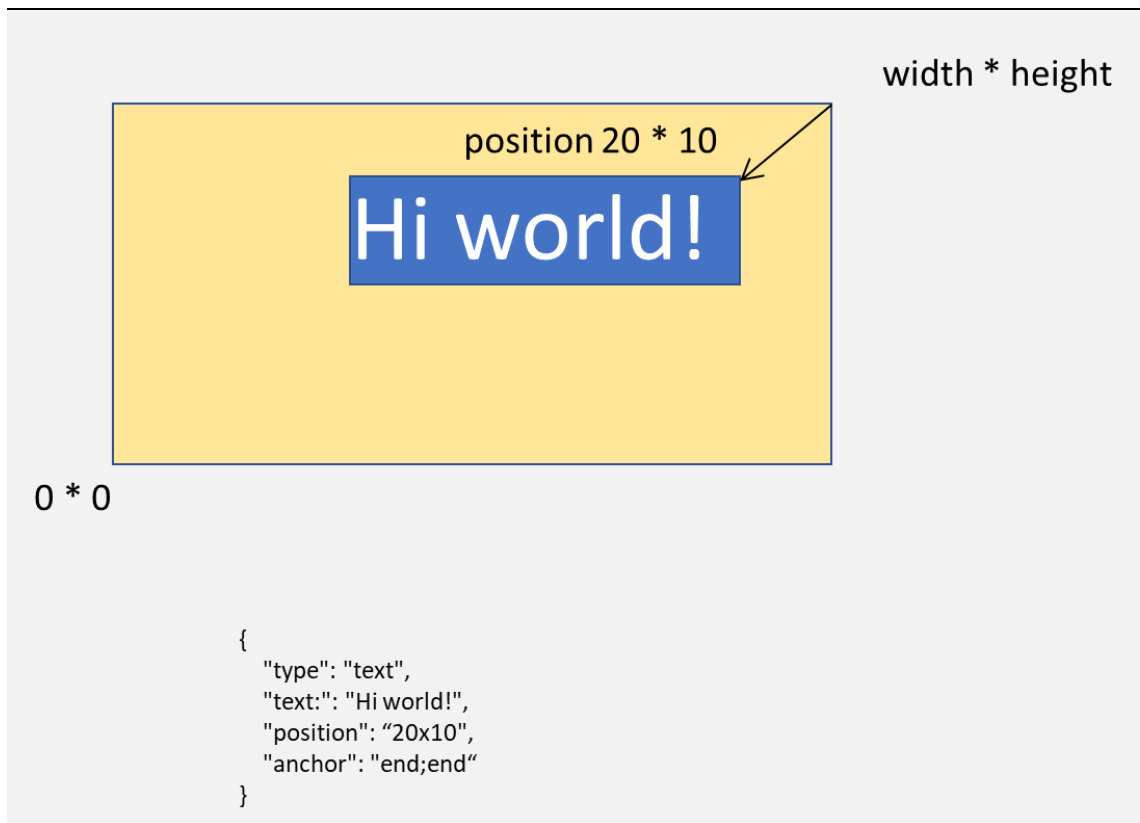
- position (optional)

At what position on the field do we draw the shape

Later we will see that a position can both be absolute and relative with regard to its container.

9.2.4 Anchor

Now we can already operate, but some definitions might be clumsy because we may not know the size of the rectangle in advance. Here comes the concept of an "anchor".



The anchor property has two parts, the horizontal and the vertical rule.

- `anchor ::= horizontal [";" vertical]`
- `horizontal ::= "start" (default) | "center" | "end"`
- `vertical ::= "start" (default) | "center" | "end"`

If only "horizontal" is defined, both directions take the same value.

The anchor property defines for both the container and the shape what is the corner that all dimensions are relative to. For example, with "end;end", the top right corner of the signature field and the top right corner of the shape are used to interpret "position". You should notice that the signs of the axes are inverted, too.

Mathematically we simply move the origins to the declared anchor points and mirror the coordinate system.

Practically you simply position the shape exactly without knowing the field size upfront.

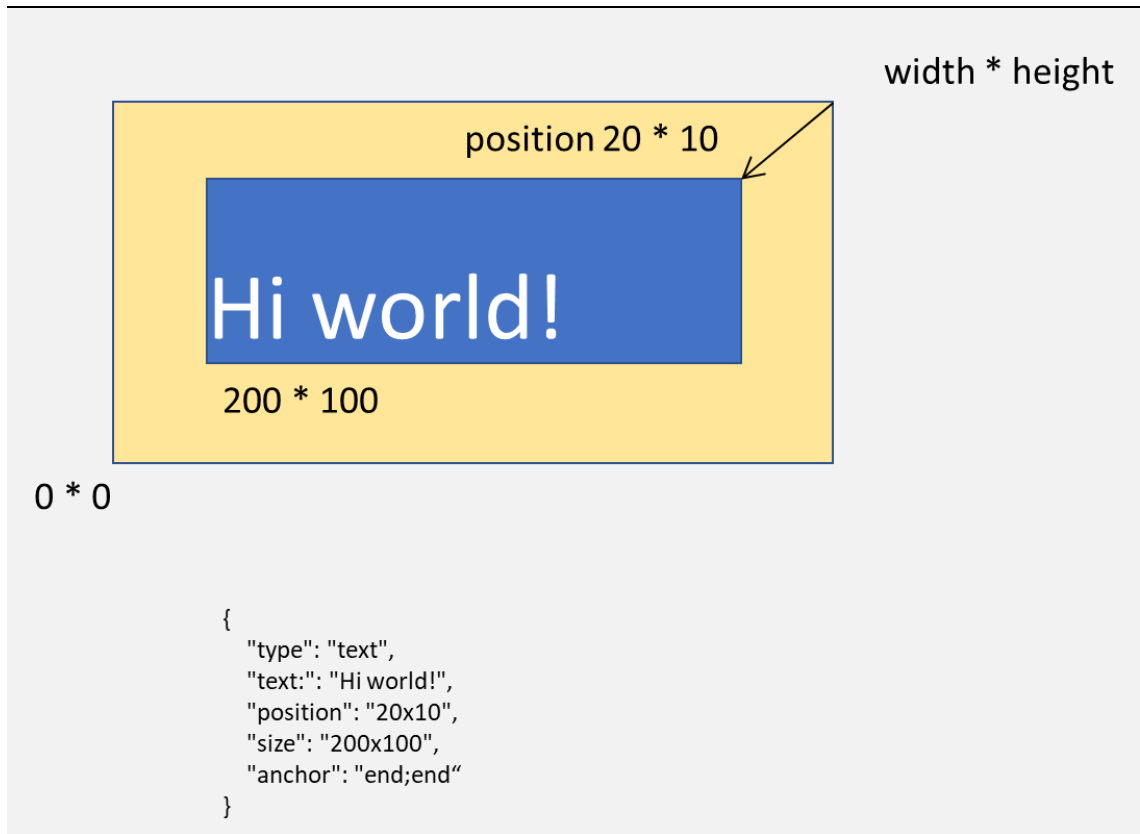
Remember that we switch the anchor points for **both** rectangles.

Upcoming version may support switching independently if necessary.

9.2.5 Size

Now we will have a look at the size of the shape. As we see, in the examples above there is no explicit size definition but the text still has a size. This is because of the fact that a shape computes a default size based

on its content, the text or icon bounding box. But if we want a special size to achieve a special result, we always can define it explicitly.



The text now has a bigger shape – but in this scenario the only effect is that the text is moved a little bit further to the lower left. If we had a rectangle shape the effect would be immediately visible.

Again, we will see later that the size definition can be relative to the container.

It is important to remember that no clipping is applied to the shape rectangle when the content is rendered.

9.2.6 Align

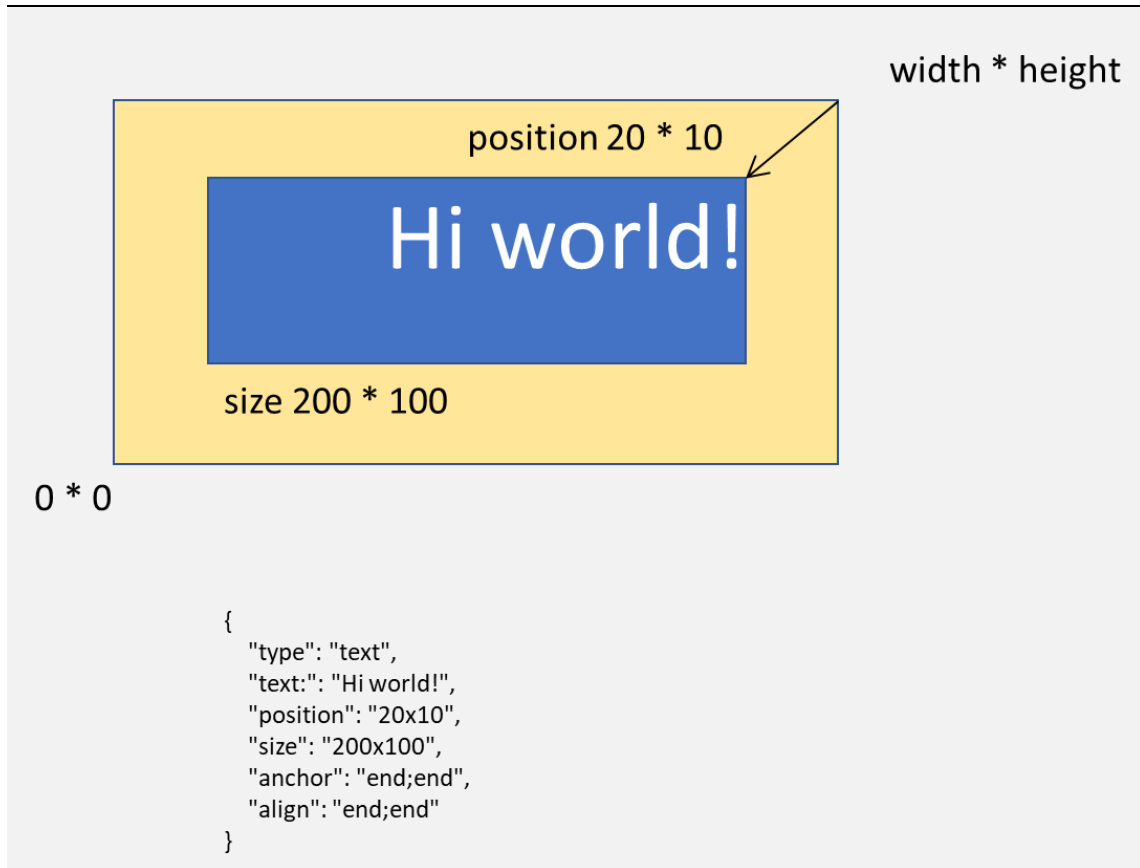
An interesting feature of a shape (especially for text and image) is the ability to align and fit the content within the bounding rectangle defined by the size. The "align" property too has a horizontal and a vertical rule, but in addition we have an option for scaling.

- align ::= horizontal [";" vertical [";" fit]]
- horizontal / vertical ::=
 - "start" (default)
 - "center"
 - "end"
- fit ::=
 - "never" (default)
 - "fill"

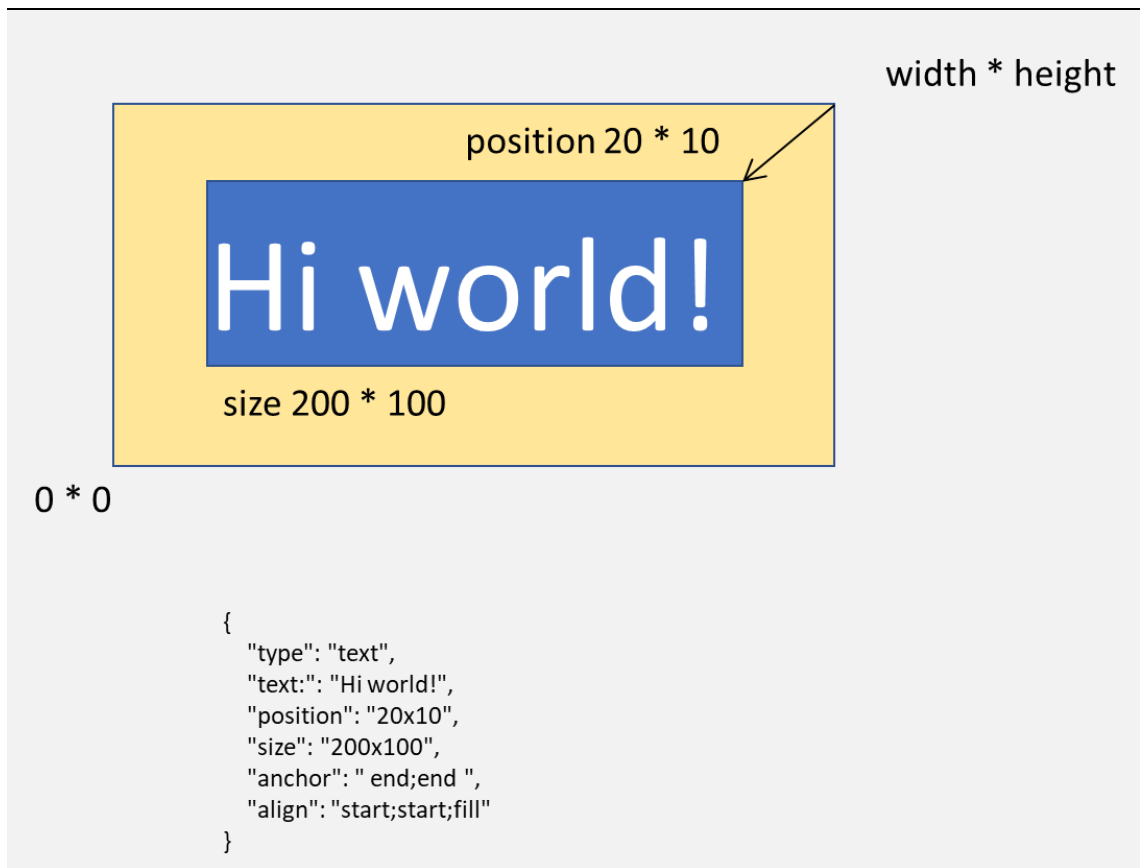
- "grow"
- "shrink"

If only "horizontal" is defined, both directions take the same value.

This property determines what to do with remaining space within the shape between its borders and the content.



We see here an example where the text is aligned top/right within the rectangle.



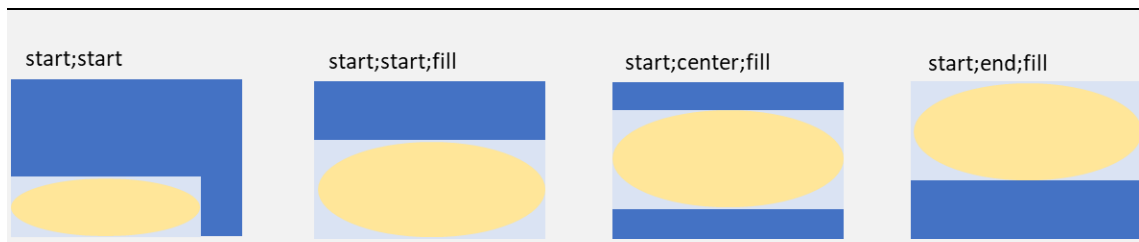
Here we have an example where the text content is scaled up to the shape size.

The alignment rules can have the following values

- start (default)
Move all content to the start position
- center
Center all content
- end
Move all content to the end position

The "fit" rules can have the following values

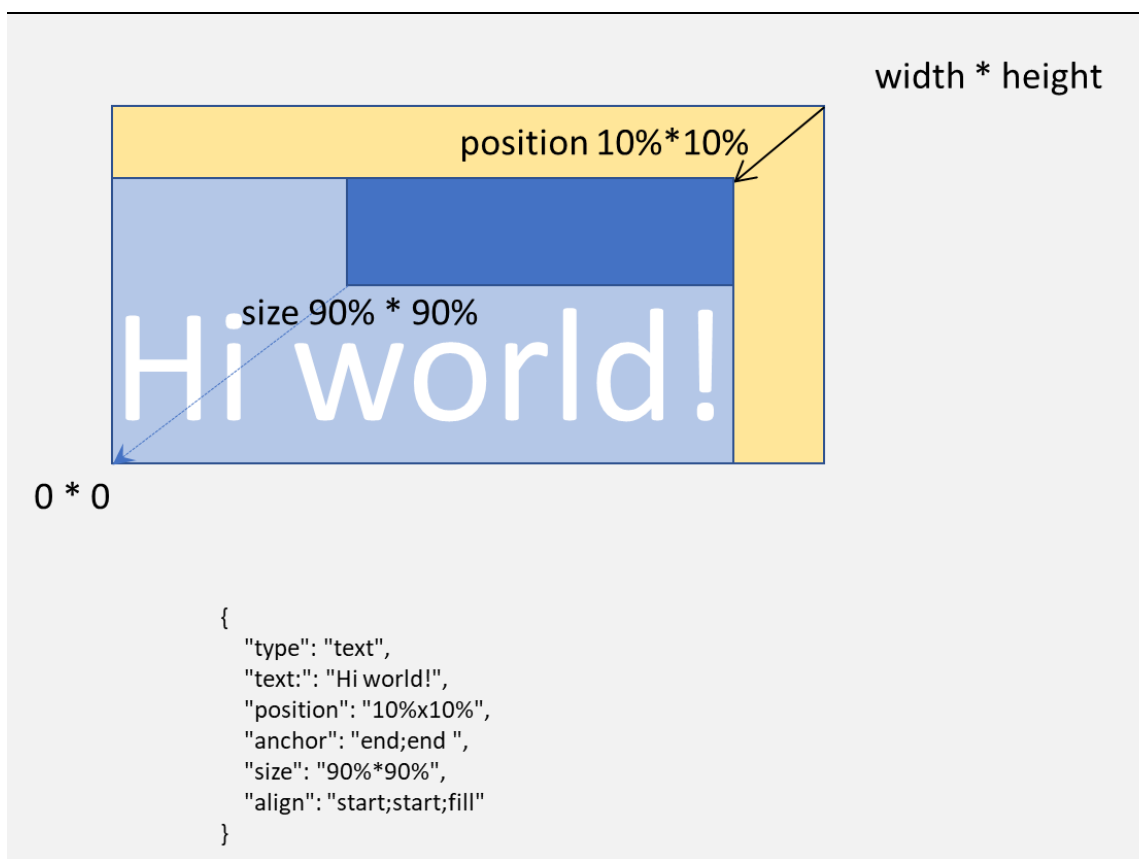
- never (default)
Never scale the content
- fill
Enlarge when content is too small and reduce when too large
- shrink
If the content fits, leave it alone. Reduce if it is too large.
- grow
If the content is smaller, enlarge. Otherwise leave it alone.



Scaling is **always** proportional. If the scaling factors differ, the smaller of both is selected to ensure the overall result can be rendered within the shape size. The other dimension is treated according to the alignment rule. The picture above illustrates the behavior.

9.2.7 Relative coordinates

This leads us to a special feature of the position and size specifications. Both can be expressed as a percentage of the container (normally the signature field rectangle). To do so, just use a "%" sign in the definition of the size or position.



Here the shape size is scaled to 90% of the container after moving it 10%, the text content again is scaled to fit the shape size.

9.3 Shapes

9.3.1 Shape

A shape is any graphical element that can be rendered. All share the properties below (there may be some that do not use certain properties, though).

type	
String	<p>The type of shape to draw. See the type value for the respective shape description</p> <p>Examples</p> <ul style="list-style-type: none"> • "text"
position	
String	<p>The position of the shape anchor point relative to the container anchor point. This is a numeric absolute or percent value of the container.</p> <p>Examples</p> <ul style="list-style-type: none"> • 200*10 • 10%*20%
size	
String	<p>The size of the shape. This is a numeric absolute or percent value of the container.</p> <p>Examples</p> <ul style="list-style-type: none"> • 200*100 • 100%*20%
anchor	
String	<p>The anchor point for the shape and the container.</p> <p>anchor ::= horizontal ";" vertical horizontal / vertical ::= "start" "center" "end"</p> <p>Examples</p> <ul style="list-style-type: none"> • start;start
align	
String	<p>The alignment for the shape within its size rectangle</p> <p>align ::= hAlign [";" vAlign [";" fit]] hAlign, vAlign ::= "start" "center" "end" fit ::= "never" "fill" "grow" "shrink"</p>

	Examples <ul style="list-style-type: none"> • start;start • start;start;fill;fill
nonStrokeColor	
String	The color for non-stroke operations (fill). This is either <ul style="list-style-type: none"> • single float for grayscale, 0-1, 0 is black • 3 floats for RGB, 0-1, 0 is no color for the component (black)
strokeColor	
String	The color for stroke operations (lines). This is either <ul style="list-style-type: none"> • single float for grayscale, 0-1, 0 is black • 3 floats for RGB, 0-1, 0 is no color for the component (black)
strokeWidth	
float	The width for the stroke operation.

9.3.2 Text

The text shape takes the text property and uses a simple layouter to render an internal paragraph (newlines etc. are supported).

The text is rendered using the nonStrokeColor, the strokeColor is ignored.

... all properties from "Shape"	
-	
type	
String	Must be "text"
text	
String	The text to be displayed
textAlign	
String	The alignment of the text in multiline paragraphs textAlign ::= "start" "center" "end" where start is the default. Examples <ul style="list-style-type: none"> • start
fontName	
String	The name of a font on the server to be used. The default is the font defined in the PDF default appearance or Helvetica if none.
fontSize	

String	The size of the font to be used. The default is the size in the default appearance or 10.
--------	---

Example

```
{
  "type": "text",
  "text": "Hi world"
}
```

9.3.3 Icon

This shape can be used to render all internally supported types of image data (like PNG or JPG).

The color properties are ignored.

... all properties from "Shape"	
-	
type	
String	Must be "icon"
icon	
locator	A locator to the icon data

Example

```
{
  "type": "icon",
  "icon": {
    "path":
      "https://upload.wikimedia.org/wikipedia/commons/8/85/Erika_Mustermann_2005.jpg"
  }
}
```

9.3.4 Rectangle

Draw a rectangular shape. The rectangle content uses the nonStrokeColor, the rectangle border uses the strokeColor.

... all properties from "Shape"	
-	
type	
String	Must be "rectangle"

Example

```
{
  "type": "rectangle",
  "strokeWidth": 10,
  "strokeColor": "1;0;0",
  "nonStrokeColor": "0;1;0",
  "size": "100*100",
  "position": "10*10"
}
```

9.3.5 Ellipse

Draw an elliptic shape (remember a circle is an ellipse...) within the shape rectangle. The content uses the nonStrokeColor, the border uses the strokeColor.

... all properties from "Shape"	
-	
type	
String	Must be "ellipse"

Example

```
{
  "type": "ellipse",
  "strokeWidth": 10,
  "strokeColor": "1;0;0",
  "nonStrokeColor": "0;1;0",
  "size": "100*100",
  "position": "10*10"
}
```

9.3.6 Path

Draw a user defined path.

If strokeColor is defined, the path has a border, if nonStrokeColor is defined, the path is filled. The default for strokeColor is black, the default for nonStrokeColor is no color.

... all properties from "Shape"	
-	
type	
String	Must be "path"
points	
array of points	An array of points (x and y coordinates).

Example

```
{
  "type": "path",
  "strokeWidth": 10,
  "strokeColor": "1;0;0",
  "points": [{
    "x": 10,
    "y": 40
  }, {
    "x": 100,
    "y": 40
  }]
}
```

9.4 The decorator

Now it's time to apply all these concepts to signature field appearance creation.

As we already know it from the existing implementations (see [1]), we declare a "decorator".

service call

```
POST /cloudsuite-gears/core/api/v1/flow/signer/create HTTP/1.1
Content-Type: application/json

{
  "args": {
    "documentSigner": {
      "args": {
        "field": {
          "position": "10x10",
          "size": "300x100"
        },
        "decorator": {
          "factory":
"de.intarsys.security.document.type.pdf.signature.shape.ShapeDecoratorFactory",
          "args": {
            "shapes": [{
              "type": "rectangle",
              "size": "100%*100%",
              "nonStrokeColor": "0.9;0.9;0.9"
            }]
          }
        }
      }
    },
    ...
  }
}
```

The `de.intarsys.security.document.type.pdf.signature.shape.ShapeDecoratorFactory` does exactly what we need here.

It takes a list of shapes in the "shapes" argument. Each shape is rendered in the order given in the list to the signature field.

It is important to know, that the shape decorator **expands all its arguments** in the context of the digest signer. This means

- the well known


```
"text": "I am ${digestsigner.subject.CN}"
```

 is working

- all other decorator arguments can be evaluated from variables, too

9.5 Examples

9.5.1 Simple rectangle

service call

```
POST /cloudsuite-gears/core/api/v1/flow/signer/create HTTP/1.1
Content-Type: application/json

{
  "args": {
    "documentSigner": {
      "args": {
        "field": {
          "position": "10x10",
          "size": "300x100"
        },
        "decorator": {
          "factory":
"de.intarsys.security.document.type.pdf.signature.shape.ShapeDecoratorFactory",
          "args": {
            "shapes": [{
              "type": "rectangle",
              "size": "100%*100%",
              "nonStrokeColor": "0.9;0.9;0.9"
            }
          ]
        }
      }
    },
    ...
  }
}
```

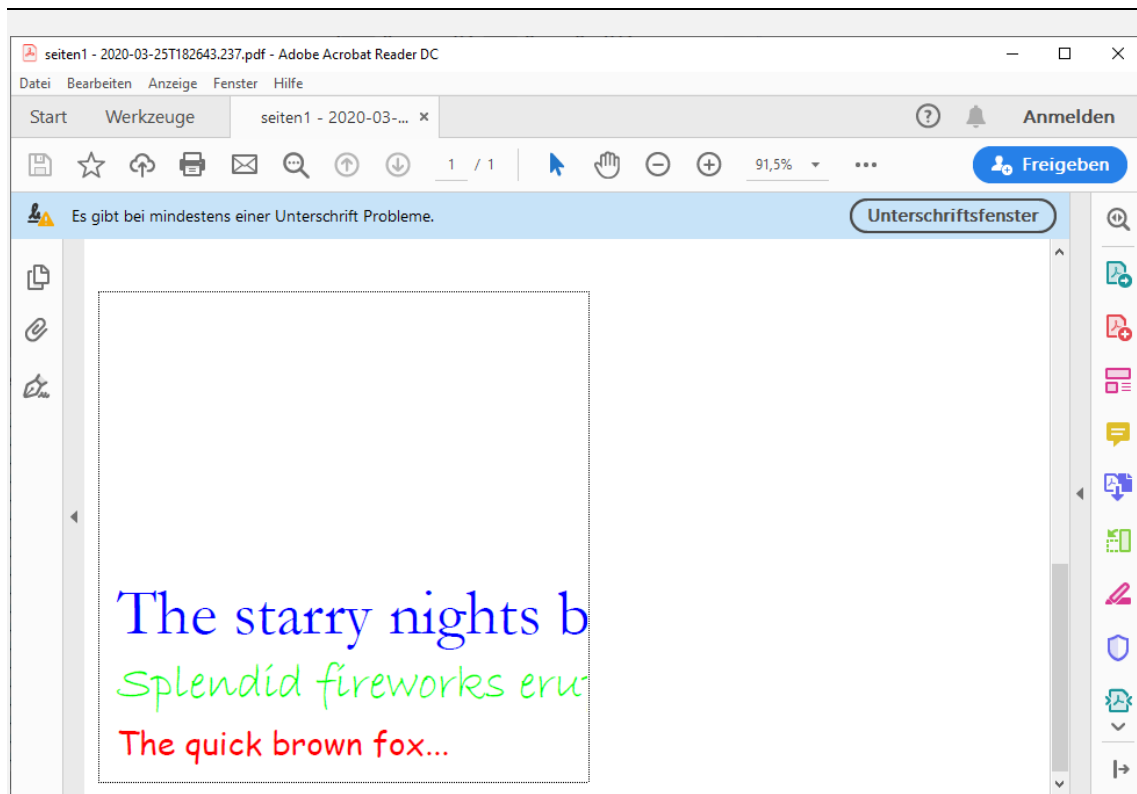
9.5.2 Multiple fonts

service call

```
POST /cloudsuite-gears/core/api/v1/flow/signer/create HTTP/1.1
Content-Type: application/json

{
  "args": {
    "documentSigner": {
      "args": {
        "field": {
          "position": "10x10",
          "size": "300x300"
        },
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        },
        "decorator": {
          "factory":
"de.intarsys.security.document.type.pdf.signature.shape.ShapeDecoratorFactory",
          "args": {
            "shapes": [{
              "type": "text",
              "text": "The quick brown fox...",
              "fontName": "Comic Sans MS",
              "fontSize": 20,
              "nonStrokeColor": "1;0;0",
              "position": "10x10"
            }, {
              "type": "text",
              "text": "Splendid fireworks erupted...",
              "fontName": "Bradley Hand ITC",
              "fontSize": 30,
            }
          ]
        }
      }
    }
  }
}
```

```
        "nonStrokeColor": "0;1;0",  
        "position": "10x40"  
    }, {  
        "type": "text",  
        "text": "The starry nights brightens...",  
        "fontName": "Garamond",  
        "fontSize": 40,  
        "nonStrokeColor": "0;0;1",  
        "position": "10x80"  
    }  
]  
}  
}  
},  
...  
}
```



9.5.3 Multiple icons

service call

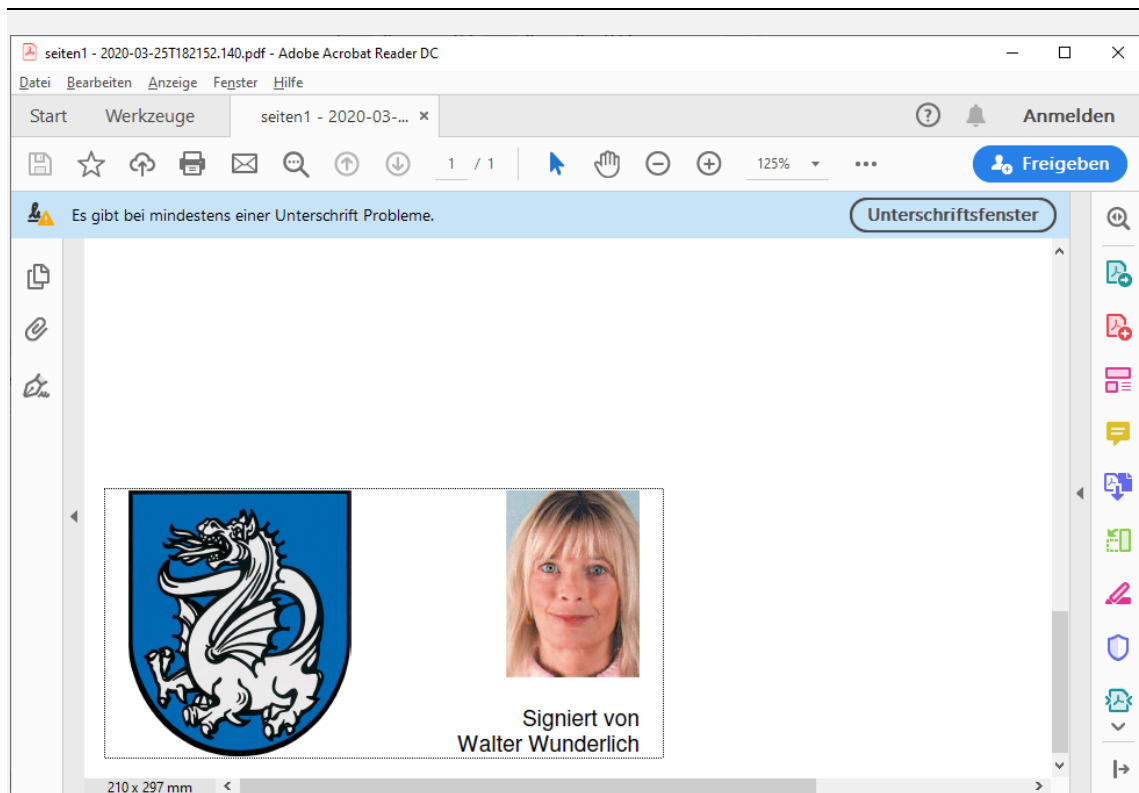
```
POST /cloudsuite-gears/core/api/v1/flow/signer/create HTTP/1.1
Content-Type: application/json

{
  "args": {
    "documentSigner": {
      "args": {
        "field": {
          "position": "10x10",
          "size": "250x120"
        },
        "decorator": {
          "factory":
"de.intarsys.security.document.type.pdf.signature.shape.ShapeDecoratorFactory",
          "args": {
            "shapes": [{
              "type": "icon",
              "icon": {
                "path":
"https://upload.wikimedia.org/wikipedia/commons/7/7b/Wachtberg_bei_Bonn_Wappen.png"
              },
              "position": "10*0",
              "anchor": "start;start",
              "size": "100*100%",
              "align": "start;start;fill"
            }, {
              "type": "icon",
              "icon": {
                "path":
"https://upload.wikimedia.org/wikipedia/commons/8/85/Erika_Mustermann_2005.jpg"
              },
              "position": "10*0%",
              "anchor": "end;end",
              "size": "60*70%",
              "align": "start;start;fill"
            }, {
```

```

        "type": "text",
        "text": "Signiert von\nWalter Wunderlich",
        "position": "10*0",
        "textAlign": "end",
        "anchor": "end;start"
    }
  ]
}
},
...
}

```



10. Additional crypto and security components

10.1 TLS client authentication with signer device

The `ConfigurableSslContextProvider` described in the gears manual support the use of custom and predefined key manager providers.

10.1.1 `IKeyManagerProvider`

[*de.intarsys.security.app.ssl.DeviceBasedKeyManagerProvider*](#)

Create a key manager provider based on a registered signing device. This is especially useful if referring to a pool device, as this will enable you to conveniently activate and deactivate access to the client authentication credentials through the gears control panel.

Properties

signerDevice	
IDevice	A bean reference to a device used to create a signing application. This application will be used in order to perform client authentication during a TLS handshake.
signerArgs	
IArgs	An optional argument map, containing arguments to be passed to the signing application.

Example

```

<bean id="aisSoftAuth"
class="de.intarsys.security.device.keystore.device.KeystoreDeviceFactoryBean">
  <property name="id" value="aisSoftAuth" />
  <property name="keyStoreFileName"
    value="\${ais.tls.keyStoreName:\${ais.clientKeyStorePath:}}" />
  <property name="keyStorePassword"
    value="\${ais.tls.keyStorePassword:\${ais.clientKeyStorePassword:}}" />
</bean>

<bean id="aisSoftSslContextBuilder"
class="de.intarsys.tools.ssl.ConfigurableSslContextProvider">
  <property name="protocol" value="TLS" />
  <property name="trustAll" value="\${ais.tls.trustAll:false}" />
  <!-- retain TLS sessions for 60 seconds -->
  <property name="sslSessionTimeout" value="60" />
  <property name="keyManagerProvider" >
    <bean class="de.intarsys.security.app.ssl.DeviceBasedKeyManagerProvider">
      <property name="signerDevice" ref="aisSoftAuth" />
      <!-- either provide the key password statically here or dynamically via control
panel caching -->
      <!--
      <property name="signerArgs">
        <map>
          <entry key="signerPassword"
            value="\${ais.tls.keyPassword:\${ais.clientKeyPassword:}}" />
        </map>
      </property>
      -->
    </bean>
  </property>
</bean>

```

10.2 OAuth 2.0 login to gears control panel

OAuth 2.0 (OAuth2) based login to the gears control realm assumes a session-less protocol leveraging token-based authentication between a frontend (the gears control panel) and the backend (the gears core control services). The frontend obtains an access token from an Authorization Service (AS), while the backend validates this token, evaluates its contents and makes an access decision.

The OAuth 2.0 protocol defines a variety of possible setups. For the following configuration details, we assume JWS-based token security and offline token validation.

10.2.1 Backend configuration

The backend configuration relies on Spring security and the concepts introduced in the gears manual and security whitepaper.

First of all, we'll have to attach a token-based authentication filter to the gears control realm. This is done by redefining the *securityRealmControlAuthenticationFilter* bean with an appropriate substitute:

```

<!--
The filter detects the bearer token and starts the authentication process if present
-->
<bean id="securityRealmControlAuthenticationFilter"

    class="org.springframework.security.oauth2.server.resource.web.BearerTokenAuthenticatio
nFilter">
    <constructor-arg ref="securityRealmControlAuthenticationManager" />
    <property name="bearerTokenResolver">
        <bean
class="org.springframework.security.oauth2.server.resource.web.DefaultBearerTokenResolve
r" />
    </property>
    </bean>

```

This will ensure that incoming requests to the control services are scanned for bearer tokens where you'd expect them: in the HTTP Authorization header.

Next, we redefine the *securityRealmControlAuthenticationManager* bean and instruct it to derive authentication status from a JWT authentication provider:

```

<security:authentication-manager id="securityRealmControlAuthenticationManager">
    <security:authentication-provider ref="jwtAuthenticationProvider" />
</security:authentication-manager>

```

This authentication provider is the core of this token authentication process:

```

<!--
The authentication provider creates a spring Authentication object along with granted
authorities from the validated token.
-->
<bean id="jwtAuthenticationProvider"

    class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthentica
tionProvider">
    <constructor-arg ref="jwtDecoder" />
    <property name="jwtAuthenticationConverter">
        <bean
class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticat
ionConverter">
            <property name="jwtGrantedAuthoritiesConverter">
                <bean class="de.intarsys.spring.security.JwtGrantedAuthoritiesByRolesConverter">
                    <property name="authoritiesClaimName" value="resource_access.gears.roles" />
                    <!-- alternative for extraction of realm roles
                    <property name="authoritiesClaimName" value="realm_access.roles" />
                    -->
                    <!-- ROLE_ is the default prefix prepended to the IDP user's role
                    <property name="authorityPrefix" value="ROLE_" />
                    -->
                    <!-- gears checks for role ROLE_OPERATOR, so put everything into upper-case -->
                    <property name="authorityToUpperCase" value="true" />
                    <!--false is default here - stick with it
                    <property name="authorityToLowerCase" value="false" />
                    -->
                </bean>
            </property>
        </bean>
    </property>
</bean>

```

The snippet above assumes that

- we use Keycloak for the authorization service part,
- register a client “gears” for the backend application within Keycloak,

- define a role “operator” within the Keycloak client “gears” and
- assign this role within this application to authorized users within your Keycloak realm.

By default, Keycloak assumes the “roles” scope being requested and provides the user’s client roles within the *resource_access.<client>.roles* claim. We could also rely on client-independent realm roles by assigning a value of *realm_access.roles*.

If we only manage operators within the Keycloak realm anyway, we could also drop the role inspection and just assume the operator role for any authenticated user. This would boil the previous example down to the following alternative:

```
<!--
The authentication provider creates a spring Authentication object along with granted
authorities from the validated token.
-->
<bean id="jwtAuthenticationProvider"

    class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthentica
tionProvider">
    <constructor-arg ref="jwtDecoder" />
    <property name="jwtAuthenticationConverter">
        <bean
class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticat
ionConverter">
            <property name="jwtGrantedAuthoritiesConverter">
                <bean class="de.intarsys.spring.security.JwtStaticGrantedAuthoritiesConverter">
                    <property name="authorities" value="ROLE_OPERATOR" />
                </bean>
            </property>
        </bean>
    </property>
</bean>
</property>
</bean>
```

In any case, as a last piece in the setup, we need to instruct gears where to get keys for validation of received tokens:

```
<!--
The JwtDecoder takes the authentication token string, parses and validates.
-->
<bean id="jwtDecoder" class="de.intarsys.spring.security.JwtDecoderFactoryBean">
    <property name="jwkSetUri"
value="https://cloudsuite.intarsys.de/auth/realms/gears/protocol/openid-connect/certs"
/>
</bean>
```

The JWT decoder here fetches the IDP realm’s JWKS and asserts that the token has really been issued by the authorization service itself.

10.2.2 Frontend configuration

The gears control panel frontend application can react to secured backend services in three ways:

1. Transparently, completely relying on the browser.
2. By showing user/password dialog, giving the possibility to log in using HTTP Basic Authentication.

3. By redirecting to an OAuth 2.0 authorization page, requesting the issuance of an access token.

Option 1 is straight-forward and works out-of-the box.

Option 2 is the default behavior for authorization requirements not fulfilled by the browser and thus reaching the frontend application itself.

Option 3 has to be activated as a replacement for option 2 whenever we want to use OAuth 2.0 – as we assume in this section of the book.

In order to activate OAuth 2.0, we place a file named “configuration.json” in the folder `${cloudsuite.config.shared}/ui/ng`. This file should look like the following:

```
{
  "control": {
    "authentication": {
      "scheme": "oauth",
      "parameters": {
        "responseType": "code",
        "issuer": "https://cloudsuite.intarsys.de/auth/realms/gears",
        "clientId": "gears-ui",
        "scope": "openid"
      }
    }
  }
}
```

Setting “control.authentication.scheme” to “oauth” is the basic element for activating OAuth 2.0 in the control panel application. (There’s also “basic” available, which is the default – see option 2 - and thus will not be further discussed here.)

“control.authentication.parameters” contains a set of parameters to be interpreted by the scheme implementation. It will be transparently passed to the initialization part of NodeJS module “angular-oauth2-oidc”, so generally all properties found at <https://manfredsteyer.github.io/angular-oauth2-oidc/docs/classes/AuthConfig.html> can be used.

Note: redirectUri and noRedirectToLogoutUrl are predefined by the application for the sake of proper working and should not be redefined.

For a usual setup, where we have public client with *clientId* registered with the Authorization Service identified and accessed by *issuer*, which may use the OAuth 2.0 Code Grant flow, the sample above is just fine and just needs adjustment of *clientId* and *issuer* to your specific infrastructure setup.

11. Blackening Editor

11.1 Overview

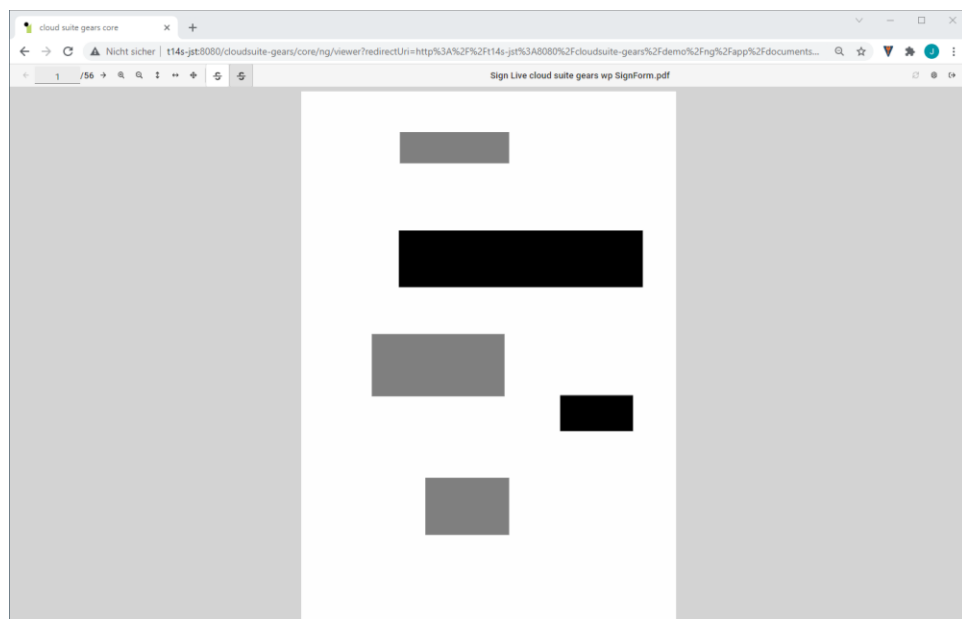
Gears' Blackening Editor prepares a PDF document for a confidential export by reducing the document to its visual parts and allowing the user to select areas to be blackened.

Following rules apply to the resulting document:

- interactive data structures are reduced to their visible representations
- blackened areas cannot be reconstructed
- internal meta data is removed
- all attachments are removed
- biometric data of handwritten signatures is removed

11.2 The Editor

The editor offers an optimized setting and an edit mode, toggled in the toolbar. The optimized setting mode lets the user define one region after the other to be blackened. The edit mode allows to select a blackened area and to remove it completely or change its extension.



11.3 Configuration

The following viewer configuration defines a viewer with blackening functions.

JSON fragment

```
[{
  "plugins": [{
    "factory": "de.intarsys.plugin.BlackeningEditor",
    "args": {}
  }
],
  "widgets": [{
    "parent": "de.intarsys.widget.toolbar.additions",
    "id": "blackening",
    "type": "Toggle",
    "icon": "fas:caret-square-right",
    "callbacks": {
      "select": {
        "factory": "BlackeningEditorSetMode",
        "args": {
          "mode": "define"
        }
      }
    }
  }, {
    "parent": "de.intarsys.widget.toolbar.additions",
    "id": "blackening",
    "type": "Toggle",
    "icon": "fas:pen-square",
    "callbacks": {
      "select": {
        "factory": "BlackeningEditorSetMode",
        "args": {
          "mode": "edit"
        }
      }
    }
  }
]
}]
```

12. Standalone operation

12.1 Overview

Apart from the current standard operation mode as a WAR-packaged application, gears core can be run as a standalone application without the need to install and maintain a servlet container (typically Tomcat).

The standalone application features following operational flavours:

- Shell application
- Service

The configuration of the standalone application follows the same rules as described in the general manuals provided with Sign Live! Cloud suite gears, except that

- Tomcat-specific settings (e.g. Tomcat-inlined TLS) don't apply and
- HTTP server configuration (e.g. port, path) are configured using Spring Boot means.

12.2 Installation

The standalone application is provided as a dedicated distribution package named “cloudsuite-gears-standalone-versionAndBuildIdentifier.zip”.

In order to install the application, copy the folder at “application/cloudsuite-gears-core” from the archive to the desired installation destination. We will refer to this location as *gearsCoreBase* in the following.

Beyond, a pre-installed Java installation is required. Please see the gears manual for additional information on supported Java versions.

12.3 Operation

12.3.1 Shell application

12.3.1.1 On Windows

The application comes with the following scripts:

- gears-core_start.bat
- setenv.bat

Edit “setenv.bat” if you want to provide environment variables to the application. It is called from the application’s start script.

Use “gears-core_start.bat” to start the application. Termination of the application requires termination of the corresponding shell.

12.3.1.2 On Linux

The application comes with the following scripts:

- gears-core_start.sh
- setenv.sh

Edit “setenv.sh” if you want to provide environment variables to the application. It is called from the application’s start script.

Use “gears-core_start.sh” to start the application. Termination of the application requires exiting the corresponding shell.

12.3.2 Service

12.3.2.1 On Windows

The application comes with the following scripts:

- gears-core_service_install.bat
- gears-core_service_start.bat
- gears-core_service_stop.bat
- gears-core_service_status.bat
- gears-core_service_uninstall.bat

Use “gears-core_service_install.bat” to install the Windows Service.

Use “gears-core_service_uninstall.bat” to uninstall the Windows Service.

You can use “gears-core_service_start.bat”, “gears-core_service_stop.bat” and “gears-core_service_status.bat” to modify or request the service status. These operations are typically handled using the Windows service console.

Beyond, the service triggers the shell application scripts introduced in the previous section. Environment variables set in “setenv.bat” are applied accordingly.

12.3.2.2 On Linux

There is currently no explicit support for Linux service operation.

12.4 Configuration

Beyond the well-documented standard properties of the application, standalone operation introduces the need to explicitly set HTTP specifics within the application. The major properties for this purpose are given in the following:

server.port	
string optional	The port to listen at for incoming connections. Default: 8080
server.servlet.context-path	
Instance optional	The context path of the application. Default: /cloudsuite-gears/core

12.5 Docker containerization

12.5.1 Ubuntu image with Tomcat

This variant results in an image comprising the recommended components for running the application and the gears core standalone application itself:

- Ubuntu Linux 22.04 LTS
- Azul Zulu OpenJDK 17

12.5.1.1 Building the image

It is recommended to derive an application image based on the current application version.

Application image: intarsys/gears-core_standalone:latest

The application comes with a predefined Dockerfile which can be used to create the application image. In order to build an image

1. Navigate to <gears_home>/example/docker/gears-core_standalone.
2. Place the folder “cloudsuite-gears-core” into the current folder.
3. If available, copy license file(s) into subfolder “config/licenses”.
4. Place any additional configuration in subfolder “config”.
5. Run the following command line:

```
docker build -t intarsys/gears-core_standalone:latest .
```

This will build the application image and tag it locally as intarsys/gears-core_standalone:latest.

The default configuration for this image directs all log output to the container’s console, thus making it available to standard log processing tools of typical container runtime environments.

12.5.1.2 Running the image

Most probably you will run the image using the container orchestrator of your choice. As a starting point, you can spawn a container and run it in your local Docker engine by executing the following command line:

```
docker run -p 8080:8080 --name gears-core intarsys/gears-core_standalone:latest
```

This will create and run a container from the previously built image `intarsys/gears-core_standalone:latest`, label it with `gears-core` and expose the internal port 8080 to port 8080 of the host system.