

Sign Live! cloud suite gears SignForm whitepaper

August 2024

intarsys GmbH

Sign Live! cloud suite gears SignForm whitepaper

Version 8.13

Editing und processing complex signature definitions

intarsys GmbH
Sign Live! cloud suite gears SignForm whitepaper
Version 8.13

All rights reserved
© 2020 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 describes in detail the SignForm feature. It contains design & API information.

So, this is the document for architects and developers.

■ 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.

E-Mail 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 in no way be 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
▪ Reviews and comments	5
▪ Disclaimer	6
Contents	7
1. Overview	10
2. Data structures	11
2.1 Overview	11
2.2 Value	11
2.2.1 LiteralValue	11
2.2.2 ReferenceValue	12
2.3 VariableDescriptor	12
2.4 Domain	13
2.4.1 BasicDomain	13
2.4.2 EnumerationDomain	14
2.4.3 UnionDomain	14
2.5 Form	15
2.6 FieldType	16
2.7 Field template	16
2.8 Field	16
2.8.1 Common properties	16
2.8.2 Resize options	18
2.8.3 Permissions	21
2.9 Text based field	21
2.10 TextField	22
2.11 CheckboxField	22
2.12 ImageField	23
2.13 SignatureDepictionField	23

Content

2.14	Configurations	24
2.15	FormValue	24
2.16	DtoFieldValues	25
3.	Actions	27
3.1	SignForm action	27
3.1.1	Form generic properties	29
3.1.2	Field generic properties	30
4.	Variables	32
4.1	Basic use	32
4.2	Automatic variable binding	32
4.3	Explicit variable binding	34
4.4	Editor support	36
5.	Editor API	37
5.1	Overview	37
5.2	Editor call	37
5.3	Editor result	38
6.	Editor customization	40
6.1	Plugin arguments	40
7.	Editor NLS	46
7.1	Overview	46
7.2	Message codes	46
8.	Processor API	49
8.1	Overview	49
8.2	Processor call	49
8.3	Processor result	52
8.3.1	Signed document	53
8.3.2	Field/Value pair	53
9.	Processor customization	54
9.1	Plugin arguments	54
9.2	ActionsSidebar	55
10.	Processor NLS	57
10.1	Overview	57
10.2	Message codes	57
11.	Signer/create API	58
11.1	Overview	58
11.2	Usage	58
12.	Demo integration	60
12.1	Overview	60
12.2	Editor activation	60
12.3	Processor activation	61
13.	External References	63

Content

1. Overview

Sign Live! cloud suite gears provides features for defining complex signature representations and their processing.

This is for example a process where a user is prompted to sign an agreement where he enters some information, uploads an image and / or provides a handwritten signature.

This document describes

- the underlying data structure
- the editing mechanics
- the API for entering the form editor
- the properties for customizing the edit component
- the processing mechanics
- the API for entering the form processor
- the properties for customizing the processing component

2. Data structures

2.1 Overview

These are basic data structures that are used to implement the SignForm behavior. They are defined here only in JSON notation, as most often they are used as call arguments or results.

2.2 Value

For later use we start with the basic construct of a value. We need this indirection to differentiate between concrete literal values and references to a variable – just like the difference between a literal and a variable in a programming language.

A **Value** simply resolves to a concrete value at runtime.

It is an abstract superclass

These are the properties for all **Value** implementations

Property	Description
type	The discriminator for a concrete Value implementation, e.g., "literal"

2.2.1 LiteralValue

This value holds a literal JavaScript value.

Property	Description
type	"literal"
value	The concrete value

Example LiteralValue

JSON fragment

```
{
  "type": "literal",
  "value": 5
}
```

Example LiteralValue

JSON fragment

```
{
  "type": "literal",
  "value": "Hello"
}
```

2.2.2 ReferenceValue

This value holds a reference to a context variable. It can be resolved at runtime against the active variables

Property	Description
type	"reference"
name	The variable name

Example ReferenceValue

JSON fragment

```
{
  "type": "reference",
  "name": "myVar"
}
```

2.3 VariableDescriptor

A VariableDescriptor may be used to declare a variable that will be available at runtime.

The user interface may provide facilities to show a list of available variables to the user.

Property	Description
name	The variable name. This will be used to lookup a variable value at runtime
label	A human readable name. This will be displayed in the form editor UI to ease selection of variables

Example VariableDescriptor

JSON fragment

```
{
  "label": "E-Mail Adresse",
  "name": "user.email"
}
```

2.4 Domain

A domain describes the set of possible values for a variable or field value. The discriminator for a concrete domain subtype is "type".

These are the properties for all Domain implementations

Property	Description
type	The discriminator for a concrete domain implementation, e.g., "basic"

The following domains are supported.

2.4.1 BasicDomain

This domain denotes all instances of a basic JavaScript type.

Property	Description
type	"basic"
value	The name of a concrete basic type. This is one of any string number datetime image signature

Example BasicDomain

JSON fragment

```
{
  "type": "basic",
  "value": "string"
}
```

2.4.2 EnumerationDomain

A domain that enumerates all possible items explicitly

Property	Description
type	"enumeration"
items	A list of possible EnumerationItem objects

EnumerationItem

Property	Description
value	A Value object that defines the concrete value (see "Value" above)

Example EnumerationDomain

JSON fragment

```
{
  "type": "enumeration",
  "items": [{
    "value": {
      "type": "literal",
      "value": "Hello"
    }
  }, {
    "value": {
      "type": "reference",
      "name": "myVar"
    }
  }
]
```

2.4.3 UnionDomain

This domain defines the union of all sub-domains

Property	Description
type	"union"
domains	A list of Domain objects

Example UnionDomain

JSON fragment

```

{
  "type": "union",
  "domains": [{
    "type": "basic",
    "value": "image"
  }, {
    "type": "enumeration",
    "items": [{
      "value": {
        "type": "literal",
        "value": {
          "name": "image 1",
          "content": "<base64>"
        }
      }
    }, {
      "value": {
        "type": "literal",
        "value": {
          "name": "image 2",
          "content": "<base64>"
        }
      }
    }
  ]
}
]
}

```

2.5 Form

A form is the description of some properties that can be filled according to some specific rules via some UI at runtime.

The form definition is independent from the interpreter and as such from a concrete rendering.

The form and fields define the PDF signature appearance.

This form has nothing to do with the PDF AcroForm technology.

Property	Description
id	A unique name
fields	A list of Field objects
properties	<p>A generic set of key/value pairs that are associated with a form but that are not directly related with form processing.</p> <p>An example of such a generic property is "annotationType" which is used by gears to control the rendering to a PDF annotation when a signature is created.</p>

Example Form

JSON fragment

```
{
  "fields": [{
    "type": "text",
    "id": "text",
    "label": "Textfeld",
    "pageRange": "0",
    "x": 79.77,
    "y": 519,
    "width": 243.21,
    "height": 85.58,
    "editable": true,
    "required": false,
    "domain": null,
    "initialValue": {
      "type": "literal",
      "value": "tinkerbelle"
    }
  }]
}
```

2.6 FieldType

The field type designates, well, the type of the field.

It is one of

text | number | datetime | checkbox | image | signature

Each type has a dedicated set of properties and UI elements for editing and processing.

2.7 Field template

For some use cases, a field type can be replaced by using a "Field template". This is simply a (partially) defined field instance (see below).

Whenever this field template is "instantiated", a field of the respective type is created where all properties are preset from the field template.

For example, for form editing, the plugin can be parameterized with the available types. Here you can simply use field templates, too. Whenever the user creates a new field, all properties from the template are copied, first.

This is especially useful if you want to change the defaults for some of the field properties, like "required". Just create a "template type" of your own.

2.8 Field

2.8.1 Common properties

A field defines a single user definable property of a form. It can show and collect information from a user in the processing step.

These are the properties that are common to all form fields

Property	Description
id	A unique name
label	A label for the field
description	A description for the field
type	<p>A field has a type. This implies the "control" that allows interaction with the user</p> <p>This is one of</p> <p>"text"</p> <p>"number"</p> <p>"datetime"</p> <p>"checkbox"</p> <p>"image"</p> <p>"signature"</p>
domain	<p>A field has a domain. This is the set of all possible values for the field. Depending on the domain, "null" is a possible member of the set.</p> <p>If no domain is specified, the value is implied (has a default) by the field type.</p> <p>Examples:</p> <ul style="list-style-type: none"> • A "text" field has a default BasicDomain of "string". • A "signature" field has a default UnionDomain of <ul style="list-style-type: none"> ○ "image" BasicDomain ○ "signature" BasicDomain
initialValue	<p>A field can have an initial value which is a "Value" object. It can be assigned in the form editor.</p> <p>If a field has an initial value, it is resolved and copied as the value upon field creation in the form processor</p>
required	Flag if this field needs some content to be valid.
editable	Flag if this field can be changed by the user interactively

x	The x position in user coordinates
y	The y position in user coordinates
width	The width in user coordinates
height	The height in user coordinates
pageRange	<p>The page range where this field is visible. This is 0 based.</p> <p>You can use</p> <ul style="list-style-type: none"> • "first" • "last" • "all" • page number ("3") • page range ("3-5") • list of page number or page range ("2;4-6")
hResize	<p>The horizontal resize strategy that is applied to map the field content into the field rectangle.</p> <p>This is currently one of never grow shrink fill inflate</p> <p>Default fill</p>
vResize	<p>The vertical resize strategy that is applied to map the field content into the field rectangle.</p> <p>This is currently one of never grow shrink fill</p> <p>If not provided, defaults to hResize</p>
properties	<p>A generic set of key/value pairs that are associated with a field but that are not directly related with form processing.</p> <p>An example of such a generic property is "annotationType" which is used by gears to control the rendering to a PDF annotation when a signature is created.</p>

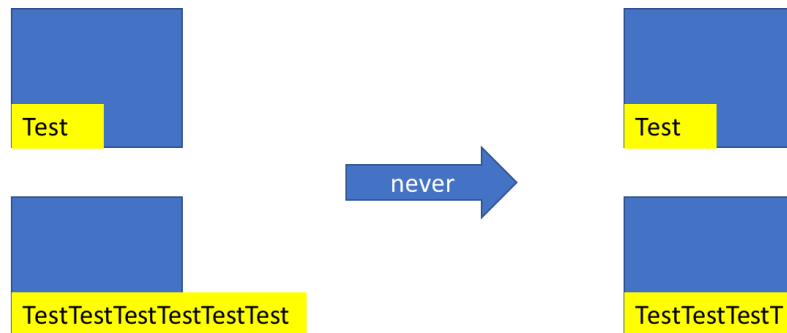
2.8.2 Resize options

The "hResize" and "vResize" options control the rendering of the content within the field rectangle.

The available options are:

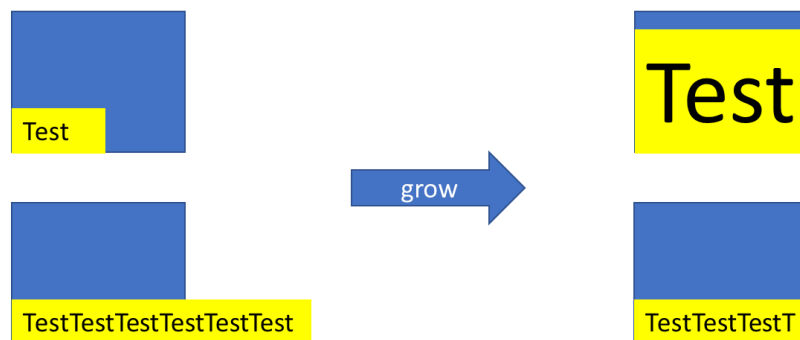
- never

Neither the field content nor the field rectangle are changed



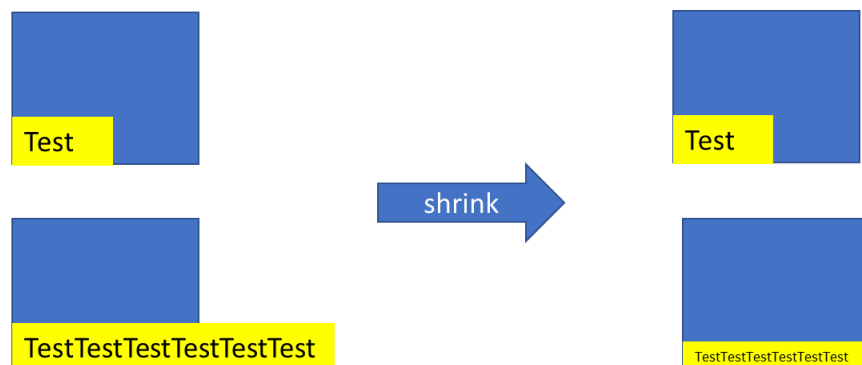
- grow

The field content is enlarged if it is smaller than the field, otherwise left unchanged



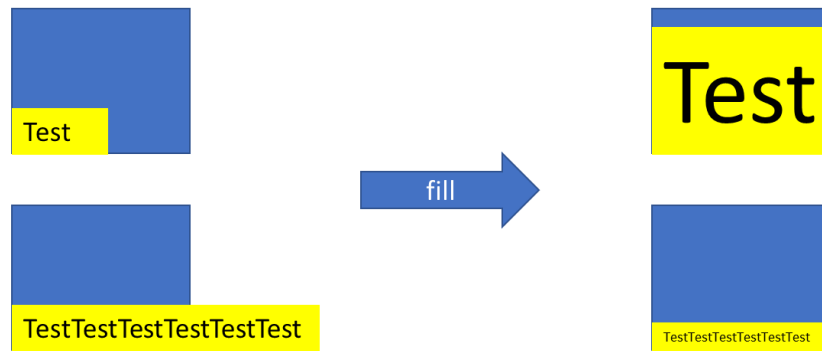
- shrink

The field content is shrunk if it is larger than the field, otherwise left unchanged.

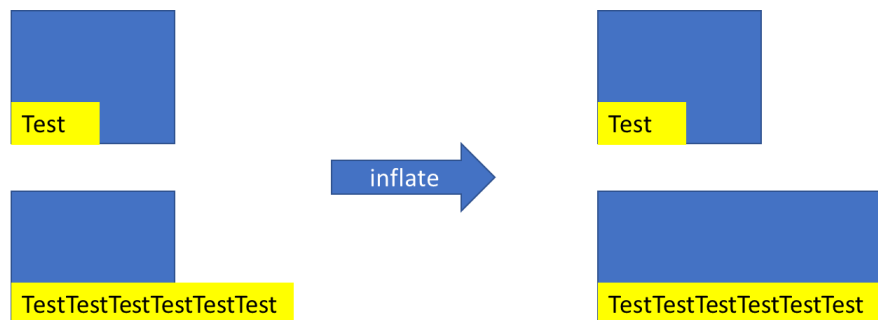


- fill

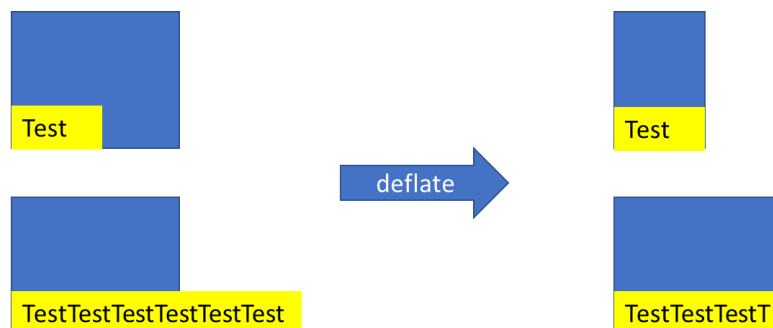
The field content is always resized to fit the field



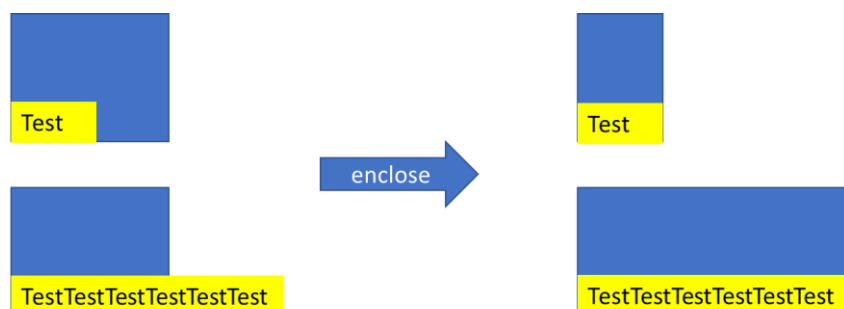
- inflate
The field is enlarged if the content is larger



- deflate
The field is squeezed if the content is smaller



- enclose
The field is always resized to fit the content



2.8.3 Permissions

A field can carry along a "permissions" map that defines what operation can be executed in what context.

The permission syntax is expressed in JSON

```
permission: {
  "<action>": "<context>"
}
```

Valid action tags for our use cases are

- **resize**
The user may resize the field
- **move**
The user may move the field

Valid context tags for the use cases described here are

- **always**
The action can be executed regardless of context
- **never**
The action cannot be executed regardless of context
- **default**
The action can be executed in conformance with the context default (the same as defining no permission)
- **process**
This is the context when we are in the processor

Multiple context tags can be concatenated with a comma ",".

If no permission is defined, a context sensitive default applies.

Example:

This is a text field that can be resized in any context (especially in the form processing step).

```
{
  "type": "text",
  "label": "Resizable Text",
  "permissions": {
    "resize": "always"
  },
  "x": 100,
  "y": 100,
  "width": 200,
  "height": 50
}
```

2.9 Text based field

All text-based field share some common properties:

Property	Description
----------	-------------

fontName	The name of a font that must be available on the backend system for text rendering. If the font is not defined or not found, "Helvetica" is used.
fontSize	The font size in points for the rendering. If this value is = -1 or undefined, the resulting text will by default be automatically scaled to fit both horizontally and vertically into the bounding rectangle of the field (hResize=fill and vResize=fill). If this value is > 0, the text will be rendered at this exact size. No scaling is done. The result will be clipped if it exceeds the bounding rectangle (hResize=never and vResize=never). This scaling behavior can be overridden by setting the hResize respective the vResize properties.

2.10 TextField

A text field provides simple text input.

Text field properties

Property	Description
type	"text"

Example TextField

JSON fragment

```
{
  "type": "text",
  "id": "foo",
  "label": "Foo",
  "pageRange": "all",
  "x": 183.12,
  "y": 349.6,
  "width": 204.43,
  "height": 132.95,
  "editable": true,
  "required": false,
  "initialValue": {
    "type": "literal",
    "value": "Bar"
  }
}
```

2.11 CheckboxField

A checkbox field provides Boolean input capabilities.

checkbox field properties

Property	Description
type	"checkbox"
threeState	Flag if this checkbox support "undefined" state

Example CheckboxField

JSON fragment

```
{
  "type": "checkbox",
  "id": "foo",
  "label": "Foo",
  "pageRange": "all",
  "x": 183.12,
  "y": 349.6,
  "width": 204.43,
  "height": 132.95,
  "editable": true,
  "required": false
}
```

2.12 ImageField

The image field allows the definition of an image in various formats.

Image field properties

Property	Description
type	"image"

Example ImageField

JSON fragment

```
{
  "type": "image",
  "id": "foo",
  "label": "Foo",
  "pageRange": "all",
  "x": 183.12,
  "y": 349.6,
  "width": 204.43,
  "height": 132.95,
  "editable": true,
  "required": false
}
```

2.13 SignatureDepictionField

This field allows to enter representations of the signature. This can be

- an uploaded image
- a handwritten signature
- a selection from some predefined images

SignatureDepictionField field properties

Property	Description
type	"signature"

Example SignatureDepictionField

JSON fragment

```
{
  "type": "signature",
  "id": "foo",
  "label": "Foo",
  "pageRange": "all",
  "x": 183.12,
  "y": 349.6,
  "width": 204.43,
  "height": 132.95,
  "editable": true,
  "required": false
}
```

2.14 Configurations

This is a list of **Configuration** objects. This list is the input and output of the form editor process.

Example

JSON fragment

```
[{
  "id": "foo",
  ...
}, {
  "id": "bar",
  ...
}]
```

When returned from the viewer, it is encapsulated as a result artifact and decorated with the type

```
de.intarsys.cloudsuite.gears.core.service.common.api.DtoConfigurations
```

You can find examples for the data structure in the chapter "Editor API".

2.15 FormValue

This data structure describes the form along with the current field values.

A FormValue object can be used to send it directly to the "signer/create" API as a "field" value.

form	
Form	The complete form description.
values	
Key/Value pairs	The field values

Example FormValue

JSON fragment

```
{
  "form": {
    "fields": [{
      ...
    }, {
      ...
    }
  ],
  "values": {
    "text1": "firlefanfanz",
    "checkbox1": false,
    "img1": {
      "content": "{{base64 data}}"
    }
  }
}
```

2.16 DtoFieldValues

This data structure reflects the result from filling a form. It can be requested explicitly to process the values entered by the user in an external system.

It is a simple key/value pair container where the key is the form field name and the value the form value entered by the user.

When returned from the viewer, it is encapsulated as a result artifact and decorated with the type

```
de.intarsys.cloudsuite.gears.core.service.common.api.DtoFieldValues
```

Example result stage

```

{
  "snapshot": {
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Result",
      "id": "25",
      "result": {
        "@class":
"de.intarsys.cloudsuite.gears.core.service.viewer.api.ResultViewer",
        "value": {
          "documentName": "empty.pdf",
          "artifacts": [{
            "value": {
              "@class":
"de.intarsys.cloudsuite.gears.core.service.common.api.DtoFieldValues",
              "value": {
                "foo": "bar"
              }
            }
          ],
          ...
        ]
      }
    },
    "conversation": "e30c8ad3-0b1b-4b9a-954d-8ddeb15f2567"
  }
}

```

3. Actions

3.1 SignForm action

This specialized action allows the creation of a PDF electronic signature with a set of appearances defined by form fields. Before creating the signature, the user can interact with the form fields to supply individual content (like text or images).

The **type** or **factory** of this action is "**SignForm**".

This action has additional arguments.

form	
object	<p>A Form object.</p> <p>This object can be created in the form editor step described below.</p> <p>It is interpreted and the result is forwarded to create PDF signature content in the processing step.</p>
signerCreate.*	
object	<p>These arguments are directly forwarded to the signer/create gears service implementation.</p> <p>You should take note that to ease handling of this action, the editor adds a default "signerCreate" argument that maps all content from "flow.variables.signerCreate":</p> <hr/> <pre>signerCreate: "\${flow.variables.signerCreate}"</pre> <hr/> <p>You can use the well-known arguments like</p> <ul style="list-style-type: none"> • configuration • args • ...
createArtifact	
Boolean	Flag if this action should create an additional result artifact that holds the key value pairs resulting from form filling

Example **SignForm** action (typed style)

JSON fragment

```
{
  "type": "SignForm",
  "id": "action",
  "label": "Neue Aktion",
  "description": "Tu es!",
  "icon": "far:bell",
  "form": {
    "fields": [{
      "type": "text",
      "id": "text",
      "label": "Textfeld",
      "pageRange": "0",
      "x": 79.77,
      "y": 519,
      "width": 243.21,
      "height": 85.58,
      "editable": true,
      "required": false,
      "initialValue": {
        "type": "literal",
        "value": "tinkerbelle"
      }
    }]
  }
}
```

To leverage all of its features it is important to understand the layers of this action:

- The action references a process that is executed in the viewer app on the client. This process interprets for example the "form" argument locally.
- Upon finalization the process creates a service request to the backend viewer. This is a private API and you do not need to care about its structure.
- The backend viewer process extracts the private information exchanged between client and backend and creates a service request to the signer service.
- The arguments "signerCreate.*" have a special treatment in this course. If available, they are sent all along this chain to the viewer backend. Here they are merged with all other arguments to create the final signature request

Example

JSON fragment

```
{
  "type": "SignForm",
  "id": "action",
  "label": "Neue Aktion",
  "description": "Tu es!",
  "icon": "far:bell",
  "signerCreate": {
    "args": {
      "documentSigner": {
        "args": {
          "digestSigner": {
            "factory": "de.intarsys.security.app.signature.SignerFactory",
            "args": {
              "device": "default@demo"
            }
          }
        }
      }
    }
  },
  "form": {
    "fields": [{
      "type": "text",
      "id": "text",
      "label": "Textfeld",
      "pageRange": "0",
      "x": 79.77,
      "y": 519,
      "width": 243.21,
      "height": 85.58,
      "editable": true,
      "required": false,
      "domain": null,
      "initialValue": {
        "type": "literal",
        "value": "tinkerbells"
      }
    }
  ]
}
}
```

This example explicitly selects the "demo" device for signature.

3.1.1 Form generic properties

These properties that can be attached to a form are interpreted by the SignForm action:

annotationType	
string	<p>An enumeration that determines the type of annotations that result from the form definition.</p> <p>"widget" will enforce all annotations to be widget annotations of the signature field. This will create unexpected results in the adobe reader, as it is the only product that insists all widget annotations have to show the same content. This is considered a bug, but is not currently handled by adobe.</p> <p>"markup" will enforce all annotations to be markup. This means, that an invisible signature is created and all visible artifacts are created as "comments". This is compatible with all known viewers, but may</p>

	<p>confuse users that are not intimate with PDF semantics, as the comments are technically not really part of the signature.</p> <p>"auto" tries to get the best of both worlds by using a widget annotation if only one exists and markup annotations for all other scenarios.</p> <p>In addition, you may dive even deeper and determine for each and every field the intended annotation type in the "field" object.</p> <p>One of : auto widget markup</p> <p>Default: auto</p>
createArtifact	
Boolean	<p>A flag if for this form a "DtoFieldValues" result artifact should be created.</p> <p>This can be set instead of requesting "createArtifact" with the SignForm action.</p>

This example forces the signature to be rendered using widget annotations only:

JSON fragment

```
{
  "type": "SignForm",
  "id": "forceWidget",
  "form": {
    "properties": {
      "annotationType": "widget"
    },
    "fields": [{
      "type": "text",
      ...
    },
    ...
  ]
}
```

3.1.2 Field generic properties

These properties that can be attached to a field are interpreted by the SignForm action:

annotationType	
string	<p>An enumeration that determines the type of annotation that results from the field definition. For more information see the description of "Form generic properties" above.</p> <p>"widget" will force the corresponding annotation to be a widget annotation, regardless of form.properties.annotationType.</p>

	<p>"markup" will force the corresponding annotation to be a markup annotation, regardless of form.properties.annotationType.</p> <p>"auto" is the default as described in "Form generic properties".</p> <p>One of : auto widget markup</p> <p>Default: auto</p>
--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

This example forces the signature to be rendered using a widget annotation for a single field, markup for all the others:

JSON fragment

```
{
  "type": "SignForm",
  "id": "forceWidget",
  "form": {
    "properties": {
      "annotationType": "markup"
    },
    "fields": [{
      "type": "text",
      ...
    }, {
      "type": "image",...
    }, {
      "type": "signature",
      ...,
      "properties": {
        "annotationType": "widget"
      }
    }
  ]
}
```

4. Variables

The SignForm design supports the concept of variables that are defined on the fly for an individual request. The variable values are referenced from a variety of components, e.g., they can be referenced in the string expansion mechanism.

SignForm has a special variable usage convention that allows to rely on variables without extensive explicit reference definitions, i.e., we have some automatic associations set up by default.

4.1 Basic use

Variables are defined as a property of a **Configuration** (see manual).

Example **Configuration** with "variables"

JSON fragment

```
{
  "variables": {
    "x": 1,
    "y": 2,
    "nested": {
      "foo": "bar"
    }
  }
}
```

As with other **Configuration** properties, variables are merged at runtime when multiple configurations are used, the last configuration having the highest priority.

4.2 Automatic variable binding

A **SignForm** action pulls a subset of the context variables and makes some default references based on naming conventions.

First, a **SignForm** action uses the variables subtree starting at "actions.<action id>", we call this subtree the "ActionVariables".

ActionVariables

Variable	Description
form	variables related to the form attached to this action.

The respective subtrees "form.fields.<field id>" are dedicated to variables that are used by a specific form field, this is the "FieldVariables" subtree.

FieldVariables

Variable	Description
initialValue	<p>The initialValue of a field can be overwritten by this variable of type "Value".</p> <p>The form processor will assign this value as the initial value of the form field. If the field is not editable, this will be the final value.</p>
domain	<p>The domain of a field can be overwritten by this value.</p> <p>This can be for example used to inject an enumeration of possible images at runtime.</p>

Example **Configuration** argument with variables: Inject a text value for a text field via variables at runtime

JSON fragment

```
{
  "variables": {
    "actions": {
      "myAction": {
        "form": {
          "fields": {
            "myText": {
              "initialValue": {
                "type": "literal",
                "value": "Hello!"
              }
            }
          }
        }
      }
    }
  }
}
```

Example **Configuration** argument with variables: Inject a choice of images for a signature depiction field via variables at runtime

JSON fragment

```

{
  "variables": {
    "actions": {
      "myAction": {
        "form": {
          "fields": {
            "mySignature": {
              "domain": {
                "type": "union",
                "domains": [{
                  "type": "basic",
                  "value": "image"
                }, {
                  "type": "basic",
                  "value": "signature"
                }, {
                  "type": "enumeration",
                  "items": [{
                    "value": {
                      "type": "literal",
                      "value": {
                        "name": "image 1",
                        "content": "<base64>"
                      }
                    }, {
                      "value": {
                        "type": "literal",
                        "value": {
                          "name": "image 2",
                          "content": "<base64>"
                        }
                      }
                    }
                  ]
                }
              }
            }
          }
        }
      }
    }
  }
}

```

4.3 Explicit variable binding

The automatic binding may be cumbersome to implement in certain scenarios. Let's say the application can prepare a call but does not know anything about the action and form structure it is using.

Example **Configuration** argument with variables: An application starting with a configuration that transports some variables, regardless of usage context.

JSON fragment

```
{
  "variables": {
    "user": {
      "email": "horst@supermail.com",
      "name": "Dr. Helmut Horst",
      "signature": {
        "name": "signature.png",
        "content": "<base64>"
      }
    },
    "process": {
      "label": "I am a process"
    }
  }
}
```

We need now a mechanism to reference these values from the action data structure explicitly. You can assign a **ReferenceValue** to the "initialValue" to do so.

Example: A **SignForm** action with references to the above variables.

JSON fragment

```
{
  "type": "SignForm",
  "id": "sign",
  "label": "Antrag bestätigen",
  "form": {
    "fields": [{
      "type": "signature",
      "id": "signature",
      "label": "Antrag Unterschrift",
      "pageRange": "all",
      "x": 50,
      "y": 20,
      "width": 200,
      "height": 50,
      "editable": true,
      "required": false,
      "initialValue": {
        "type": "reference",
        "name": "user.signature"
      }
    }, {
      "type": "text",
      "id": "ende",
      "label": "Ende",
      "pageRange": "last",
      "x": 183.12,
      "y": 349.6,
      "width": 204.43,
      "height": 132.95,
      "editable": false,
      "required": false,
      "initialValue": {
        "type": "reference",
        "name": "user.name"
      }
    }
  ]
}
```

4.4 Editor support

The "automatic variable binding" scenario can be of great use if the naming convention will work for you.

In this case you must be aware that the editing process may create and/ or change action and field id and you must be prepared to act accordingly in your application.

You may, for example, parse the action before creating your configuration to collect the correct ids.

The "explicit variable binding" scenario does not require this processing – but you must burden the correct assignment of a variable on the form editor user.

*** not yet implemented ***

This can be done by an experienced user by manually typing the variable names. If we have a less experienced user, it is better to provide a list of all possible selections upfront. In chapter "Editor customization" you will see how you can set up this data structure.

5. Editor API

5.1 Overview

The form editor step is activated by calling the gears viewer with the "de.intarsys.plugin.SignFormEditor" plugin in the **Configuration**.

The plugin sets up the viewer context to ensure that we have UI components for the data structures and that upon viewer exit the new configurations list is added as a viewer result object.

5.2 Editor call

Example **Configuration** argument for a form editor launch.

JSON fragment

```
{
  "plugins": [{
    "factory": "de.intarsys.plugin.SignFormEditor",
    "args": {
      ...
    }
  ]
}
```

Example **Configuration** in spring XML

spring XML fragment

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

This plugin provides all functionality needed to create and edit a set of configurations. You can add some fine tuning in the "args" to the plugin, see the chapter below.

The viewer requires the viewer argument

args.configurations

containing the list of configurations to work on. If this argument is empty, a new list with a default configuration is created.

Be aware that "configuration" is used in two roles in this call: One is the "configuration" parameter to the viewer, customizing the viewers process & UI.

The other is as an argument in the "args.configurations" list. This is a plain data structure that is going to be edited in the form editor process and must be an array.

Example: complete call to the viewer for a form editor process

viewer create call

```
POST http: //localhost:8080/cloudsuite-gears/core/api/v1/flow/viewer/create

{
  "options": {
    "redirectUri": "http://localhost:8082/cloudsuite-gears/demo/ng/app/documents"
  },
  "args": {
    "configurations": [{
      "id": "myConfig",
      "label": "Meine Konfiguration"
    }
  ],
  "configuration": {
    "plugins": [{
      "factory": "de.intarsys.plugin.SignFormEditor",
      "args": {}
    }
  ],
  "document": {
    "type": "d",
    "name": "test.pdf",
    "content": "..."
  }
}
```

5.3 Editor result

When the viewer with the form editor plugins is exited, the plugin ensures that the "configurations" data structure is added to list of result artifacts.

The new result artifact has the id "configurations" and a typed value of "de.intarsys.cloudsuite.gears.core.service.common.api.DtoConfigurations".

The value of this typed object is a list of **Configuration** objects.

Example viewer conversation snapshot with a ResultStage

JSON fragment

```

{
  "snapshot": {
    "conversation": "ab9991c5-1601-49cf-a4ea-3e9493eff1b1",
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Result",
      "id": "16",
      "result": {
        "@class":
"de.intarsys.cloudsuite.gears.core.service.viewer.api.ResultViewer",
        "value": {
          "documentName": "seiten2.pdf",
          "artifacts": [{
            "id": "configurations",
            "value": {
              "@class":
"de.intarsys.cloudsuite.gears.core.service.common.api.DtoConfigurations",
              "value": [{
                "id": "myConfig",
                "label": "Meine Konfiguration",
                "actions": [{
                  "type": "SignForm",
                  "id": "newAction0",
                  "label": "New action 0",
                  "form": {
                    "fields": [{
                      "type": "text",
                      "id": "text",
                      "label": "Textfeld",
                      "pageRange": "0",
                      "x": 149.64,
                      "y": 497.19999999999993,
                      "width": 139.98,
                      "height": 57.47,
                      "editable": true,
                      "required": false,
                      "domain": null
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}

```

6. Editor customization

6.1 Plugin arguments

You can customize the form editor component by providing "args" in the plugin definition.

The following arguments are supported for the plugin.

Argument	Description
role	A role may be used to use the components in different scenarios with different NLS resolutions (see chapter "Editor NLS") Default is "default"
expert	Flag if the UI should show features that are intended by expert use. Default is "true"
structureEditor	Configure the sidebar component representing the configurations data structure. Default is {}
variableDescriptors	A list of VariableDescriptor objects. These are selectable in the form editor UI as the initial value of a field. Default is {}
fieldTypes	A list of field types that should be presented to the user in the form editor UI. Instead of a type name, a field template can be used, too. This will ensure that all fields that are created from the template have the initial values copied from the template. Default is null, resulting in all field types Example ["text", "image", "signature"]

"structureEditor" properties

Argument	Description
widgets	Customize some widgets of the structureEditor. The structureEditor is normally displayed on the left sidebar.

"structureEditor.widgets" properties

Argument	Description
configuration.edit.visible	Set to false to hide "edit" button for configuration
configuration.add.visible	Set to false to hide "add" button for configuration
configuration.remove.visible	Set to false to hide "remove" button for configuration
action.edit.visible	Set to false to hide "edit" button for action
action.add.visible	Set to false to hide "add" button for configuration
action.it.remove.visible	Set to false to hide "remove" button for configuration
expert.visible	Set to false to hide "expert" checkbox

Example **Configuration** argument: Switch role of the plugin. This will switch to a different NLS message set.

JSON fragment

```
{
  "plugins": [{
    "factory": "de.intarsys.plugin.SignFormEditor",
    "args": {
      "role": "design"
    }
  }]
}
```

Example **Configuration** spring XML

spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
<property name="plugins">
  <list>
    <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
      <property name="factory" value="de.intarsys.plugin.SignFormEditor"/>
      <property name="args">
        <map>
          <entry key="role" value="dsign"/>
        </map>
      </property>
    </bean>
  </list>
</property>
</bean>
```

Example **Configuration** argument: Switch the expert flag

JSON fragment

```
{
  "plugins": [{
    "factory": "de.intarsys.plugin.SignFormEditor",
    "args": {
      "expert": false
    }
  }]
}
```

Example spring XML

spring XML fragment

```
<property name="plugins">
  <list>
    <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
      <property name="factory" value="de.intarsys.plugin.SignFormEditor"/>
      <property name="args">
        <map>
          <entry key="expert" value="false"/>
        </map>
      </property>
    </bean>
  </list>
</property>
```

Example **Configuration** argument: Switch visibility of specific widgets

JSON fragment

```
{
  "plugins": [{
    "factory": "de.intarsys.plugin.SignFormEditor",
    "args": {
      "structureEditor": {
        "widgets": {
          "configuration": {
            "add": {
              "visible": false
            },
            "edit": {
              "visible": false
            },
            "remove": {
              "visible": false
            }
          },
          "action": {
            "add": {
              "visible": false
            },
            "edit": {
              "visible": false
            },
            "remove": {
              "visible": false
            }
          }
        }
      }
    }
  ]
}
```

Example spring XML

spring XML fragment

```
<property name="plugins">
  <list>
    <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
      <property name="factory" value="de.intarsys.plugin.SignFormEditor"/>
      <property name="args">
        <map>
          <entry key="structureEditor.widgets.configuration.add.visible" value="false"/>
          <entry key="structureEditor.widgets.configuration.edit.visible" value="false"/>
          <entry key="structureEditor.widgets.configuration.remove.visible" value="false"/>
        </map>
      </property>
    </bean>
  </list>
</property>
```

Example **Configuration** argument: Provide some variables to select from.

JSON fragment

```
{
  "plugins": [{
    "factory": "de.intarsys.plugin.SignFormEditor",
    "args": {
      "variableDescriptors": [{
        "label": "E-Mail Adresse",
        "name": "user.email"
      }, {
        "label": "Signatur",
        "name": "user.signature"
      }
    ]
  }
]
}
```

Example spring XML

spring XML fragment

```
<property name="plugins">
  <list>
    <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
      <property name="factory" value="de.intarsys.plugin.SignFormEditor"/>
      <property name="args">
        <map>
          <entry key="variableDescriptors.0.label" value="E-Mail Adresse"/>
          <entry key="variableDescriptors.0.name" value="user.email"/>
          <entry key="variableDescriptors.1.label" value="Signatur"/>
          <entry key="variableDescriptors.1.name" value="user.signature"/>
        </map>
      </property>
    </bean>
  </list>
</property>
```

Example **Configuration** argument: client defined field types.

This will provide two "text" typed field. One uses the built-in "text" type, the other uses a template where the size is already defined.

JSON fragment

```
{
  "plugins": [{
    "factory": "de.intarsys.plugin.SignFormEditor",
    "args": {
      "fieldTypes": [
        "text",
        {
          "type": "text",
          "label": "PresizedText",
          "width": 200,
          "height": 30
        }
      ]
    }
  }
]
}
```

Example spring XML

spring XML fragment

```
<property name="plugins">
  <list>
    <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
      <property name="factory" value="de.intarsys.plugin.SignFormEditor"/>
      <property name="args">
        <map>
          <entry key="fieldTypes.0 " value="text"/>
          <entry key="fieldTypes.1.type" value="text"/>
          <entry key="fieldTypes.1.width" value="200"/>
          <entry key="fieldTypes.1.height" value="30"/>
        </map>
      </property>
    </bean>
  </list>
</property>
```

7. Editor NLS

7.1 Overview

You can customize the NLS messages used for the form editor. The basic mechanics for NLS are described in [1].

All messages for the viewer are in the bundle "de.intarsys.gears.core.ui.messages". So, if you want to redefine messages, you

- create a directory "i18n" in the \${cloudsuite.config.shared} directory
- add subdirectories for each path segment of the bundle (de/intarsys/gears/core/ui)
- add a **messages_<language>.properties** file for each language you want to define
- add a "code=text" entry for each message you want to change.

7.2 Message codes

NLS codes

```

ConfigurationEditorSidebarComponent.label=Configurations
ConfigurationEditorSidebarComponent.tip=Define configurations
ConfigurationEditorSidebarComponent.icon=far:cubes
ConfigurationEditorSidebarComponent.default.configuration.selector.label=Configuration
ConfigurationEditorSidebarComponent.default.configuration.edit.tooltip=Edit
configuration
ConfigurationEditorSidebarComponent.default.configuration.add.tooltip=Add configuration
ConfigurationEditorSidebarComponent.default.configuration.remove.tooltip=Remove
configurations
ConfigurationEditorSidebarComponent.default.configuration.remove.question=Do you really
want to remove configuration '{{label}}'?
ConfigurationEditorSidebarComponent.default.action.edit.label=Edit action
ConfigurationEditorSidebarComponent.default.action.add.label=Add action
ConfigurationEditorSidebarComponent.default.action.remove.label=Remove action
ConfigurationEditorSidebarComponent.default.action.remove.question=Do you really want to
remove action '{{label}}'?
ConfigurationEditorSidebarComponent.default.field.edit.label=Edit field
ConfigurationEditorSidebarComponent.default.field.add.label=Add field
ConfigurationEditorSidebarComponent.default.field.add.help=Create a new field on the
right side for this action.
ConfigurationEditorSidebarComponent.default.field.remove.label=Remove field
ConfigurationEditorSidebarComponent.default.field.remove.question=Do you really want to
remove field '{{label}}'?
ConfigurationEditorSidebarComponent.default.service.expert.label=Expert mode

ConfigurationEditor.AddAction.label=New action

ConfigurationEditorContainerComponent.default.field.add.label=Add field
ConfigurationEditorContainerComponent.default.field.remove.label=Remove field
ConfigurationEditorContainerComponent.default.field.remove.question=Do you really want
to remove field '{{label}}'?

ActionEditorComponent.title.label=Edit action '{{label}}'
ActionEditorComponent.id.label=ID
ActionEditorComponent.label.label=Label
ActionEditorComponent.description.label=Description
ActionEditorComponent.icon.label=Icon

ConfigurationEditorComponent.title.label=Edit configuration '{{label}}'
ConfigurationEditorComponent.id.label=ID
ConfigurationEditorComponent.label.label=Label
ConfigurationEditorComponent.description.label=Description
ConfigurationEditorComponent.icon.label=Icon

FormEditorPaletteComponent.title.label=New field
FormEditorPaletteComponent.unselected.help=Select a field type above to define a new
field
FormEditorPaletteComponent.selected.help=Draw a rectangle using the left mouse button on
the PDF

FormEditorContainerFieldComponent.title.label=Edit field '{{label}}'
FormEditorContainerFieldComponent.id.label=ID
FormEditorContainerFieldComponent.label.label=Label
FormEditorContainerFieldComponent.description.label=Description
FormEditorContainerFieldComponent.required.label=Required
FormEditorContainerFieldComponent.editable.label=Editable
FormEditorContainerFieldComponent.xPosition.label=X Position
FormEditorContainerFieldComponent.yPosition.label=Y Position
FormEditorContainerFieldComponent.width.label=Width
FormEditorContainerFieldComponent.height.label=Height

PageRangeInputComponent.selector.label=Page(s)
PageRangeInputComponent.custom.label=Custom

PageRange.all.label=All pages
PageRange.first.label=First page
PageRange.last.label=Last page
PageRange.custom.label=Custom

FormEditorContainerFieldComponent.tabs.basic.label=General
FormEditorContainerFieldComponent.tabs.value.label=Value
FormEditorContainerFieldComponent.tabs.geometry.label=Geometry

FormEditorTextFieldComponent.value.label=Value

```

```
FormEditorDatetimeFieldComponent.value.label=Value  
  
FormEditorNumberFieldComponent.value.label=Value  
  
FormEditorImageFieldComponent.upload.label=Upload  
FormEditorImageFieldComponent.clear.label=Remove  
  
FormEditorSignatureDepictionFieldComponent.userDefined.label=Upload user defined Image  
FormEditorSignatureDepictionFieldComponent.userDefined.description=The user can upload a  
image from his local device.  
FormEditorSignatureDepictionFieldComponent.signature.label=Capture signature  
FormEditorSignatureDepictionFieldComponent.signature.description=The user can capture a  
handwritten signature.  
FormEditorSignatureDepictionFieldComponent.preDefined.label=Upload a predefined image  
FormEditorSignatureDepictionFieldComponent.preDefined.description=The user can selet one  
of the predefined images.  
FormEditorSignatureDepictionFieldComponent.upload.label=Upload  
FormEditorSignatureDepictionFieldComponent.clear.label=Remove  
FormEditorSignatureDepictionFieldComponent.add.label=Add image
```


8. Processor API

8.1 Overview

The processing step is activated by calling the gears viewer with the "de.intarsys.plugin.SignFormProcessor" plugin in the **Configuration**.

The plugin sets up the viewer context to ensure that we have UI components for a **SignForm** action.

The **SignForm** action itself is part of one of the configurations that we have created in the form editor in the first part.

You can merge the configuration before or just send a list of two or more configurations in the "configuration" parameter. The server will merge them automatically with the higher indices having higher precedence.

8.2 Processor call

Example **Configuration**: activating the form processor

JSON fragment

```
{
  "plugins": [{
    "factory": "de.intarsys.plugin.SignFormProcessor",
    "args": {
      ...
    }
  ]
}
```

Example **Configuration** plugins property in spring XML

spring XML fragment

```
<property name="plugins">
  <list>
    <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
      <property name="factory" value="de.intarsys.plugin.SignFormProcessor"/>
      <property name="args">
        ...
      </property>
    </bean>
  </list>
</property>
```

This plugin provides all functionality needed to process **SignForm** actions.

But activating only the **SignFormProcessor** plugin will have no effect at all – this plugin provides only an interpreter for **SignForm** action definitions. So at least you will have to add a **SignForm** action. Ideally you will use one of the **Configuration** objects created by the **SignFormEditor** step described in the chapters above.

The third component that is missing is the one you are already familiar with: you need a widget that allows the user to trigger the action we have just defined. One possibility may be to use the generic **ActionsSidebar** described below. Another is to directly reference the action from, let's say, a toolbar widget.

These three facets can simply be combined in the **configuration** parameter as an array. You just send an array of configurations or configuration references with the **create**-call, resulting in an effective configuration by merging the elements.

Example **Configuration** argument:

JSON fragment

```
[
  {
    "plugins": [
      {
        "factory": "de.intarsys.plugin.SignFormProcessor",
        "args": {}
      }
    ],
    "widgets": [
      {
        "label": "Start Signature",
        "parent": "de.intarsys.widget.toolbar.additions",
        "callbacks": {
          "select": {
            "ref": "newAction0"
          }
        }
      }
    ]
  },
  {
    "id": "myConfig",
    "label": "Meine Konfiguration",
    "actions": [
      {
        "type": "SignForm",
        "id": "newAction0",
        "label": "New action 0",
        "form": {
          "fields": [
            {
              "type": "text",
              "id": "text",
              "label": "Textfeld",
              "pageRange": "0",
              "x": 149.64,
              "y": 497.19999999999993,
              "width": 139.98,
              "height": 57.47,
              "editable": true,
              "required": false,
              "domain": null
            }
          ]
        }
      }
    ]
  }
]
```

The viewer additionally receives its normal document parameters. The **SignFormProcessor** interprets the form fields when executing the **SignForm** action.

Example: complete call to the viewer for **SignForm** processing.

viewer create call

```

POST http://localhost:8080/cloudsuite-gears/core/api/v1/flow/viewer/create

{
  "options":
  {
    "redirectUri": "http://localhost:8082/cloudsuite-gears/demo/ng/app/documents",
  },
  "configuration": [
    {
      "plugins": [
        {
          "factory": "de.intarsys.plugin.SignFormProcessor",
          "args": {}
        }
      ],
      "widgets": [
        {
          "label": "Start Signature",
          "parent": "de.intarsys.widget.toolbar.additions",
          "callbacks":
          {
            "select":
            {
              "ref": "newAction0"
            }
          }
        }
      ]
    },
    {
      "id": "myConfig",
      "label": "Meine Konfiguration",
      "actions": [
        {
          "type": "SignForm",
          "id": "newAction0",
          "label": "New action 0",
          "form":
          {
            "fields": [
              {
                "type": "text",
                "id": "text",
                "label": "Textfeld",
                "pageRange": "0",
                "x": 149.64,
                "y": 497.19999999999993,
                "width": 139.98,
                "height": 57.47,
                "editable": true,
                "required": false,
                "domain": null
              }
            ]
          }
        }
      ]
    }
  ],
  "document":
  {
    "type": "d",
    "name": "seiten2.pdf",
    "content": "..."
  }
}

```

8.3 Processor result

8.3.1 Signed document

The viewer produces the standard result - the signed document. After the SignForm action, a couple of widgets/annotations representing the field content have been created in the document in addition to the signature itself.

8.3.2 Field/Value pair

If requested, the processor will create a result artifact containing field/value pairs for the form.

To request artifact creation you can

- add an "createArtifact" argument to the SignForm action
- add a generic "createArtifact" property to the form
- add an argument "overlay.createArtifact" argument to the plugin

9. Processor customization

9.1 Plugin arguments

You can customize the form processor component by providing "args" in the plugin definition.

The following arguments are supported for the plugin.

Argument	Description
role	A role may be used to use the components in different scenarios with different NLS resolutions (see chapter "Editor NLS") Default is "default"
expert	A flag if the UI should show features that are intended by expert use. Default is "true"
formProcessorWizard	Configure the form processor wizard component. Default is {}
overlay	Configure the form processor overlay Default is {}

" formProcessorWizard " properties

Argument	Description
layout	The direction of the stepper. horizontal vertical
widgets	Customize some widgets of the component.

" formProcessorWizard.widgets" properties

Argument	Description
cancel.visible	Set to false to hide "cancel" button for wizard

" overlay " properties

Argument	Description
createArtifact	Flag if a result artifact should be created when the action is triggered via the overlay containing field/value pairs for the form.

Example **Configuration** argument: Switch visibility of specific widgets

JSON fragment

```
{
  "plugins": [{
    "factory": "de.intarsys.plugin.SignFormProcessor",
    "args": {
      "formProcessorWizard": {
        "widgets": {
          "cancel": {
            "visible": false
          }
        }
      }
    }
  }
]
```

Example spring XML

spring XML fragment

```
<property name="plugins">
  <list>
    <bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
      <property name="factory" value="de.intarsys.plugin.SignFormProcessor"/>
      <property name="args">
        <map>
          <entry key="formProcessorWizard.widgets.cancel.visible" value="false"/>
        </map>
      </property>
    </bean>
  </list>
</property>
```

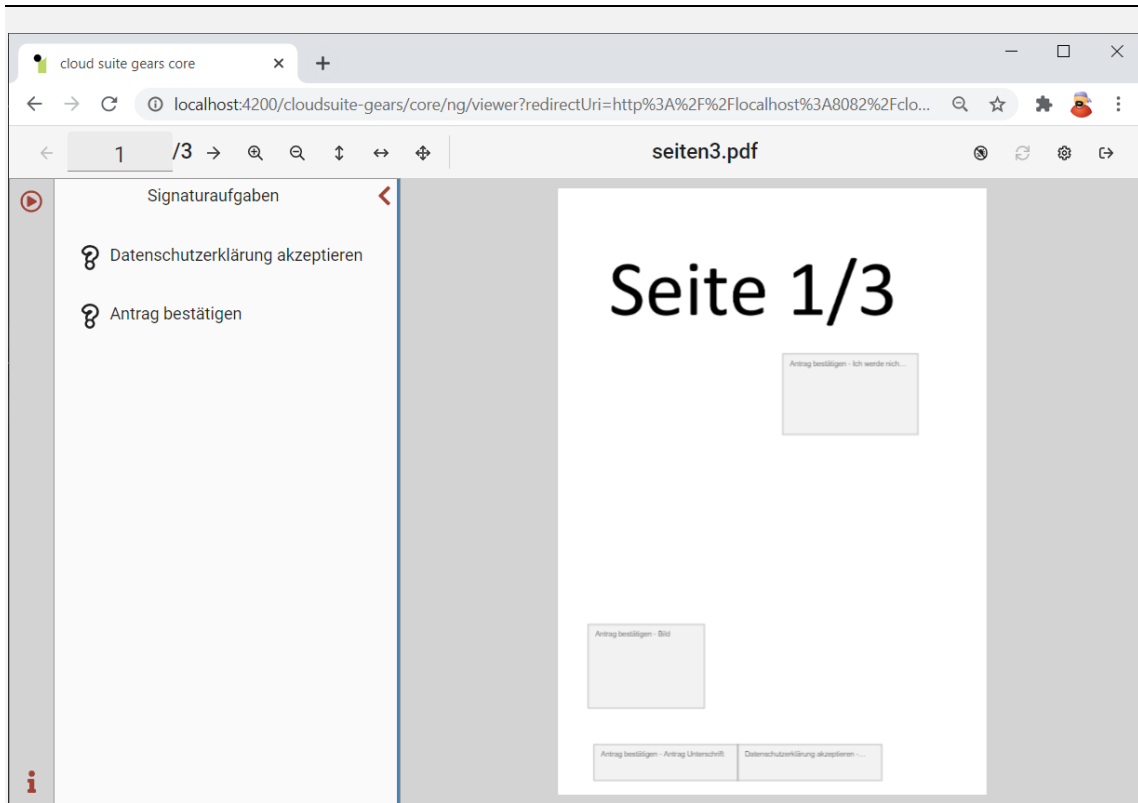
9.2 ActionsSidebar

Surely you can attach the actions defined in your **Configuration** to a widget callback. But this has to be done manually and for each action you want to provide. And, you have to take care that the id used in the form editor and the reference from the widget are the same.

When the form processor plugin is loaded, a new sidebar is made available to launch **SignForm** actions defined in a **Configuration**. If visible, this sidebar shows automatically **SignForm** actions that are available in the current **Configuration**.

If you need a generic UI component to show all available actions, give the sidebar a try.

Example: Sidebar in action in the viewer.



The sidebar is registered using the id

ActionsSidebar

just like any other sidebar you have already defined.

Example JSON

JSON fragment

```
{
  ...
  "widgets": [
    {
      "id": "actions",
      "parent": "de.intarsys.widget.sidebar.components",
      "type": "ActionsSidebar"
    }
  ]
  ...
}
```

Example **Configuration** XML definition

spring XML fragment

```
<property name="widgets">
  <list>
    <w:widget parent="de.intarsys.widget.sidebar.components" id="actions"
type="ActionsSidebar">
    </w:widget>
  ...
</list>
</property>
```


10. Processor NLS

10.1 Overview

You can customize the NLS messages used for the form editor. The basic mechanics for NLS are described in [1].

All messages for the viewer are in the bundle "de.intarsys.gears.core.ui.messages". So if you want to redefine messages, you

- create a directory "i18n" in the \${cloudsuite.config.shared} directory
- add subdirectories for each path segment of the bundle (de/intarsys/gears/core/ui)
- add a **messages_<language>.properties** file for each language you want to define
- add a "code=text" entry for each message you want to change.

10.2 Message codes

NLS codes

```
FormProcessorWizardComponent.default.step.submit.label=Submit
FormProcessorWizardComponent.default.back.label=Previous
FormProcessorWizardComponent.default.next.label=Next
FormProcessorWizardComponent.default.cancel.label=Cancel
FormProcessorWizardComponent.default.submit.label=Submit
FormProcessorWizardComponent.default.summary.ok.label=All required information was
collected. You can start the action by clicking the "Submit" button.
FormProcessorWizardComponent.default.summary.error.label=Some fields are not correct.
Please check your input.

FormProcessorImageFieldComponent.default.upload.label=Upload
FormProcessorImageFieldComponent.default.clear.label=Remove

FormProcessorSignatureDepictionFieldComponent.default.sign.label=Sign
FormProcessorSignatureDepictionFieldComponent.default.upload.label=Upload
FormProcessorSignatureDepictionFieldComponent.default.clear.label=Remove
FormProcessorSignatureDepictionFieldComponent.default.value.selector.label=Select image
type
```

11. Signer/create API

11.1 Overview

The result of the “SignForm” action in the viewer is a “signer/create” call along with the form and values.

If you do not use the viewer to fill a form, you can use the data structure in a signer/create call directly.

11.2 Usage

When using the signer/create API, the “documentSigner.args.field” argument is used to derive the appearance of the signature.

Instead of defining this content indirectly using the “decorator” argument, you can directly specify the content using a FormValue. You can compare this to the even more direct appearance definition using shapes.

In this case you have to create a FormValue object (see above), i.e. the form definition along with its current values and send it to the “signer/create” API as a property in the “field” argument.

Example

service call

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

```
{
  "args": {
    "documentSigner": {
      "factory": "de.intarsys.security.processor.signature.DocumentSignerFactory",
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        },
        "field": {
          "formValue": {
            "form": {
              "fields": [{
                ...
              }, {
                ...
              }
            ]
          },
          "values": {
            "text1": "firlefanfanz",
            "checkbox1": false,
            "img1": {
              "content": "{{base64 data}}"
            }
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<base64 content>"
  }
]
```

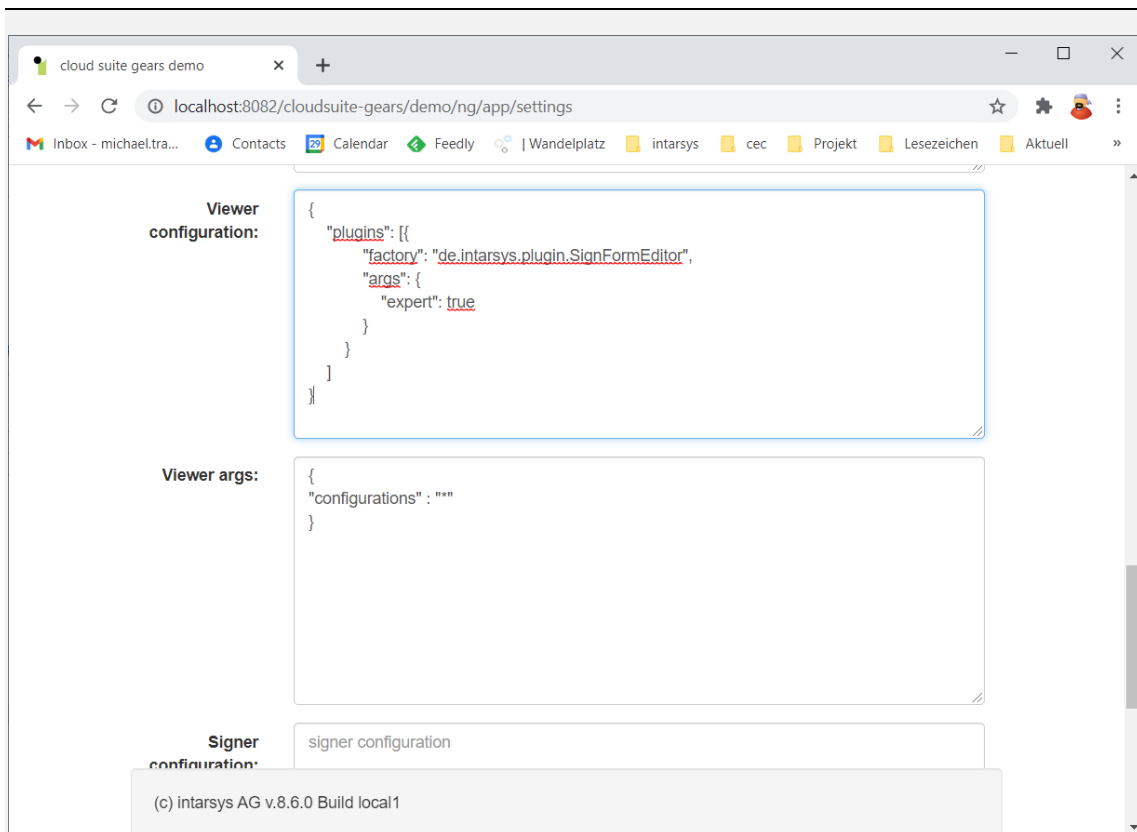
12. Demo integration

12.1 Overview

SignFormEditor and **SignFormProcessor** can easily be tested using the demo application. It was enhanced to interpret the special "DtoConfigurations" artifact.

12.2 Editor activation

You can activate the **SignFormEditor** mode by requesting the plugin in the configuration:



If you do not add 'configuration' (or an empty list) to the viewer arguments, a default configuration will be created.

Example viewer args, resulting in default configuration.

JSON fragment

```
{
  "configurations": []
}
```

If you want, you can directly add one or more Configuration object in the 'configurations' argument.

Example viewer args with configurations.

JSON fragment

```
{
  "configurations": [
    {
      "id": "config1",
      "actions": [...]
    },
    {
      "id": "config2",
      "actions": [...]
    }
  ]
}
```

The demo backend will persist all DtoConfigurations artifacts that result from a **SignFormEditor** activation with the user. This will allow you to reference these configurations later in future calls to the form editor **and** in calls to the **SignFormProcessor**. Referencing the configurations is done by using the **id** of one of the required configurations within the array 'configurations'.

Example viewer args with configuration references.

JSON fragment

```
{
  "configurations": [
    "config2",
    "config2"
  ]
}
```

Finally "*" is a wildcard supported by the demo only that will get replaced by all configurations known to the demo application.

So, the settings in the hardcopy above allows you to repeatedly call the form editor with all the configurations you have available.

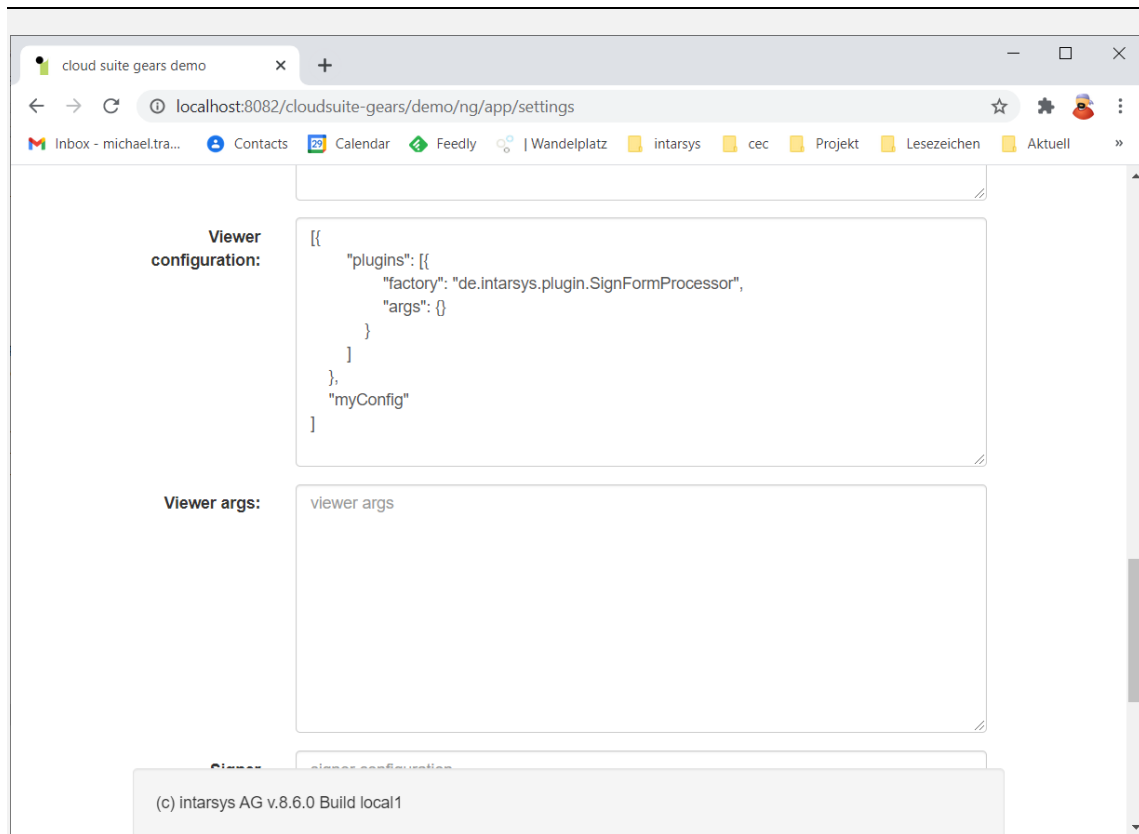
Example viewer args with configuration references.

JSON fragment

```
{
  "configurations": "*"
}
```

12.3 Processor activation

SignFormProcessor activation is even more easy:



You simply send a composite configuration that consists of the plugin and the second one that references one of the configurations you just edited. The demo backend will replace the reference with the one it persisted from the form editor result.

13. External References

- [1] intarsys GmbH, Sign Live! cloud suite gears manual.
- [2] intarsys GmbH, Sign Live! cloud suite gears cookbook.
- [3] intarsys GmbH, Sign Live! cloud suite gears incubator.
- [4] intarsys GmbH, Sign Live! cloud suite gears tutorial.
- [5] intarsys GmbH, Sign Live! cloud suite gears wp security.
- [6] intarsys GmbH, Sign Live! Security Applications Developers Guide.