

# Sign Live! cloud suite gears cookbook

Oktober 2025



intarsys GmbH

Sign Live! cloud suite gears cookbook

Version 8.14

cloud suite gears usage recipes

intarsys GmbH  
Sign Live! cloud suite gears cookbook  
Version 8.14

All rights reserved  
© 2021 intarsys GmbH  
[www.intarsys.de](http://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 is intended as a jump start for developers and may give a rough overview over special features of the product for architects and designers.

- Reviews and comments

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

Email [support@intarsys.de](mailto:support@intarsys.de)

Website [www.intarsys.de](http://www.intarsys.de)



# Contents

Preface	5
▪ Author and company	5
▪ Trademarks	5
▪ Who should read this book	5
▪ Reviews and comments	5
Contents	7
1. Overview	13
2. Self-help recipes	14
2.1 Manuals	14
2.1.1 Manual	14
2.1.2 Security applications developers guide	14
2.1.3 Cookbook	14
2.1.4 Incubator	14
2.1.5 WP architecture	14
2.1.6 WP security	15
2.1.7 WP AIS QSeal	15
2.2 Code examples	15
3. Installation recipes	16
3.1 Security configuration	16
3.1.1 Configuring the application “cryptdec”	16
3.2 Containerization	16
3.2.1 Ubuntu image with Tomcat	16
3.2.2 Red Hat image with Tomcat	18
4. Signer recipes	20
4.1 PAdES	20
4.2 PDF with invisible field	20
4.3 PDF with visible field	20
4.3.1 The field definition	20
4.3.2 The field label	22
4.3.3 The field label with certificate info	24

## Content

4.3.4	The field icon	25
4.3.5	More decorator options	29
4.3.6	Dynamic (floating) fields	30
4.3.7	Dynamic field positioning	32
4.3.8	Shape based appearance	36
4.4	PDF with timestamp	36
4.5	XAdES, enveloped signature (internal)	37
4.6	XAdES-T	38
4.7	XAdES-LT	39
4.8	Multiple internal XML Signatures	40
4.9	XAdES, detached signature	41
4.10	XAdES, enveloping signature (embedded)	42
4.11	CMS signature	44
4.12	CMS multiple signatures	44
4.13	Digest signature	45
4.14	Document specific arguments	46
4.15	Document specific tags	49
5.	Viewer recipes	50
5.1	Add alert dialog	50
5.2	Add multi action button	51
5.3	Overwrite default toolbar	53
5.4	Clear default toolbar	54
5.5	Switch off toolbars	55
5.6	Style toolbar buttons	56
5.7	Annotation overlay with dynamic fields	57
5.7.1	Overview	57
5.7.2	The "annotations" widget container	58
5.7.3	And even more dynamic...	59
5.7.4	Declarative annotations	61
5.7.5	Hints	63
5.8	Request placement of a signature field	64
5.9	Collect multiple information chunks for a complex signature appearance	65
5.10	Use a toggle button and session state	67
6.	Device recipes	69
6.1	Demo device	69
6.2	Bridge with arguments	69
6.3	Bridge exclude/include devices	70
6.4	Bridge include demo device	71
6.5	Bridge with session (comfort signature)	72
6.6	AIS device	73
6.6.1	Look up RAS evidence id	73
6.7	Sign-Me Device	74
6.7.1	Seal creation	74
7.	Integration recipes	77
7.1	Log observers	77



## Content

7.2	Webhook observers	78
7.3	More on filters	80
7.4	More on views	81
7.5	More on log files	83
7.6	More on webhooks	84
7.7	Creating an audit log	86
7.8	Plain Logback Filtering	89
7.9	Graylog	90
8.	Security recipes	93
8.1	Token-based authorization on gears flow	93
8.1.1	Application-Level Authentication with Self-Contained JWS Tokens	93
8.1.2	User-Level Authentication with Authorization Code Grant	94
8.1.3	User Authorization Verification	95
8.1.4	Enabling Introspection for Gears Core Authentication	96
9.	External References	98







# 1. Overview

---

Sign Live! cloud suite gears provides a set of components to be embedded in web application or web service environments.

This book is intended to collect and present useful recipes for the usage of the application.

## 2. Self-help recipes

---

### 2.1 Manuals

Well, you are reading this manual, so you already know that gears comes with a folder with a lot of documentation. Here is a short intro to each of them.

#### 2.1.1 Manual

The manual contains the complete description of gears, with details on installation, configuration and usage.

#### 2.1.2 Security applications developers guide

This manual contains the complete reference of the signature API that is used within gears “signer/create”.

#### 2.1.3 Cookbook

This is for the guys that either do not want to read the two books mentioned above or still need more hands-on examples.

The book contains blueprints with snippets how to implement typical scenarios using gears.

#### 2.1.4 Incubator

This manual holds information on upcoming or not-yet completely public features in the product.

You are welcome to spy on the future and comment on improvements.

#### 2.1.5 WP architecture

This document gives an overview where gears is placed in a typical system infrastructure and what challenges may arise.

### 2.1.6 WP security

A special manual how to use and configure gears to comply to special security requirements.

### 2.1.7 WP AIS QSeal

This describes the special needs when using the AIS qualified seals.

## 2.2 Code examples

Along with gears you will find a complete demo application, separated in a Java backend and an angular frontend, that implements a complete client to gears.

Feel free to use for your own needs “as is”.

## 3. Installation recipes

### 3.1 Security configuration

#### 3.1.1 Configuring the application “cryptdec”

By default, the application “cryptdec” applies 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="cryptoKeyCryptdec" class="de.intarsys.tools.crypto.bytes.DerivedByteProvider">
  <property name="kdf" ref="cryptoKdfMaster" />
  <property name="inputProvider">
    <bean class="de.intarsys.tools.crypto.bytes.StaticByteProvider">
      <property name="text" value="cryptdec" />
    </bean>
  </property>
  <property name="size" value="32" />
</bean>
```

### 3.2 Containerization

We assume the use of Docker for creation of a gears container image. The instructions in the following sections require a local installation of Docker. It is expected that you are familiar with the execution of container workloads in your operations environment.

The creation of a gears container image assumes use of a base image which provides the required Java runtime environment and a servlet container.

#### 3.2.1 Ubuntu image with Tomcat

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

- Ubuntu Linux 22.04 LTS
- Azul Zulu OpenJDK 17
- Apache Tomcat 10

Building the image

It is recommended to create the final image in two steps:

1. Build the base image providing the runtime environment.



2. Derive an application image based on the current application version.

Base image: `intarsys/tomcat:10-jdk17-zulu`

The application comes with a predefined Dockerfile which can be used to create the base image. This step requires access to Docker Hub, the official docker registry at *registry.hub.docker.com*. In order to build an image

1. Navigate to `<gears_home>/example/docker/tomcat_10-jdk-17-zulu`
2. Run the following command line:

---

```
docker build -t intarsys/tomcat:10-jdk17-zulu .
```

---

This will build the base image and tag it locally as *intarsys/tomcat:10-jdk17-zulu*.

Application image: `intarsys/gears-core: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`.
2. Copy “cloudsuite-gears#core.war” 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:latest .
```

---

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

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:latest
```

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

### 3.2.2 Red Hat image with Tomcat

This variant results in an image meant to be run on the Red Hat OpenShift platform. It builds on RHEL 9 with OpenJDK 17 and adds the following components:

- Apache Tomcat 10

Building the image

It is recommended to create the final image in two steps:

1. Build the base image providing the runtime environment.
2. Derive an application image based on the current application version.

Base image: *intarsys/tomcat:10-jdk17-rhel\_ubi9\_openjdk*

The application comes with a predefined Dockerfile which can be used to create the base image. This step requires access to Red Hat's open docker registry at *registry.access.redhat.com*, as the image is derived from Red Hat's Universal Base Image (UBI).

In order to build an image

1. Navigate to *<gears\_home>/example/docker/tomcat\_10-jdk\_17-rhel\_ubi9\_openjdk*
2. Run the following command line:

```
docker build -t intarsys/tomcat:10-jdk17-rhel_ubi9_openjdk .
```

This will build the base image and tag it locally as *intarsys/tomcat:10-jdk17-rhel\_ubi9\_openjdk*.

Application image: *intarsys/gears-core:rhel-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\_rhel*
2. Copy "cloudsuite-gears#core.war" into to 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:rhel-latest .
```

This will build the application image and tag it locally as *intarsys/gears-core:rhel-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 your container runtime environment.

### Running the image

It is expected that you have a valid Red Hat subscription and that you run the image on a Red Hat OpenShift platform.

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_rhel intarsys/gears-core:rhel-latest
```

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

## 4. Signer recipes

### 4.1 PAdES

That is easy – we only create signatures compliant to PAdES baseline (PAdES-B-B, ETSI EN 319 142-1).

So, this is already a valid call:

#### 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"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<base64 content>"
  }]
}
```

### 4.2 PDF with invisible field

The call from the recipe "PAdES" already creates an invisible field – you are done.

### 4.3 PDF with visible field

#### 4.3.1 The field definition

For a visible field you can define a bunch of properties (see [1]). The most important ones are

name
------

string	The name of the field to be created. It must be unique within the document.
position	
integer tuple	A tuple of two integers, separated by "*" or "x". The integers are the x and y coordinates of the lower left corner of the field in PDF user space.  Example 100x100
size	
integer tuple	A tuple of two integers, separated by "*" or "x". The integers are the width and height of the field in PDF user space.  Example 300x200
pageRange	
page definition	The definition on which page to render the signature appearances. This can be either 0 based page numbers, page ranges or a list thereof or predefined logical page names.  number   number – number   "first"   "last"   "all"  Default is "all"  Examples 0        // first page 1;5     // second and 6 <sup>th</sup> page 0-3;8   // first to 4 <sup>th</sup> page and 9 <sup>th</sup> page last    // the last page

This is the simplest definition for a newly created visible field. This will create a field with its appearance created by the default behavior.

```
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": {
          "position": "50*150",
          "size": "300*100"
        }
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.pdf",
      "content": "<base64 content>"
    }
  ]
}
```

### 4.3.2 The field label

The simplest way to add a field label is to define "signatureLabel" with the document signer. The built-in decorators are set up to use this value as their default text.

## 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": {
        "position": "50*150",
        "size": "300*100"
      },
      "signatureLabel": "My label"
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<base64 content>"
  }]
}
```

Beginning with later versions, more fine grained control over the signature appearance was introduced. While the defaults for this new features are derived from the existing arguments like "signatureLabel", you can use them directly, too. This may result in a better readable argument structure.

```
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": {
          "position": "50*150",
          "size": "300*100"
        },
        "decorator": {
          "factory":
"de.intarsys.security.document.type.pdf.signature.ExtendedDecoratorFactory",
          "args": {
            "text": "My label"
          }
        }
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.pdf",
      "content": "<base64 content>"
    }
  ]
}
```

### 4.3.3 The field label with certificate info

In this scenario we create a visible signature field where the text contains content from the signature certificate.

You use the "string expansion" feature to accomplish this. Details on this feature are documented in [2].

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": {
        "position": "50*150",
        "size": "300*100"
      },
      "decorator": {
        "factory":
"de.intarsys.security.document.type.pdf.signature.ExtendedDecoratorFactory",
        "args": {
          "text": "Signed by ${digestsigner.subject.CN}"
        }
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.pdf",
      "content": "<base64 content>"
    }
  ]
}
```

4.3.4 The field icon

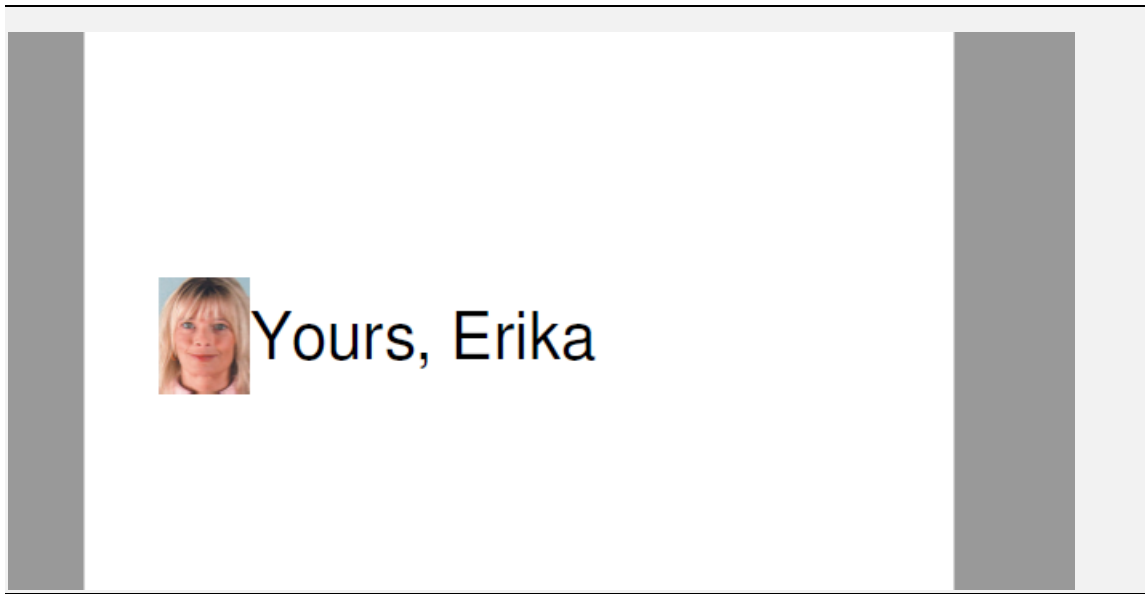
A field can have an icon, too. The "extended decorator" has a bunch of options to add text and image information. Detailed information can be found in [1].

Here we shortly repeat the relevant arguments

text	
string	The literal text to be displayed. The text is expanded before use.
textScaleWhen	
string	How to scale the text within the field. always   never   toobig   toosmall
textScaleProportional	
string	Flag if the scaling is performed proportional, <b>true</b> or <b>false</b> .

textVAlign	
String	How to align the text within the field vertically. bottom   center   top
textHAlign	
string	How to align the text horizontally. left   center   right
icon	
string	A locator to the icon to be displayed in the field. In the simplest case this is the name of an image file that will be included.
iconScaleWhen	
string	How to scale the image within the field. This is one of always   never   toobig   toosmall
iconScaleProportional	
String	Flag if the scaling is performed proportional, <b>true</b> or <b>false</b> .
iconVAlign	
string	How to align the image within the field vertically. bottom   center   top
iconHAlign	
String	How to align the image horizontally. left   center   right
layout	
string	How to merge the text and image section of the appearance. overlay   textAboveIcon   textBelowIcon   textLeftOfIcon   textRightOfIcon

Here is an example with a famous icon from the internet



## 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": {
          "position": "50*150",
          "size": "300*100"
        },
        "decorator": {
          "factory":
"de.intarsys.security.document.type.pdf.signature.ExtendedDecoratorFactory",
          "args": {
            "text": "Yours, Erika",
            "icon": {
              "path":
"https://upload.wikimedia.org/wikipedia/commons/8/85/Erika_Mustermann_2005.jpg"
            },
            "layout": "textRightOfIcon"
          }
        }
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.pdf",
      "content": "<base64 content>"
    }
  ]
}
```

The icon is defined as a locator object, allowing you to transmit the data literally (base 64 encoded), too. Just switch the "path" to a "content" parameter, holding the base64 encoded icon.

---

**service call args fragment**

---

```
...
  "decorator": {
    "factory":
"de.intarsys.security.document.type.pdf.signature.ExtendedDecoratorFactory",
    "args": {
      "text": "Hi!",
      "icon": {
        "content": "<base64>"
      },
      "layout": "textRightOfIcon"
    }
  }
...

```

### 4.3.5 More decorator options

You may want to further improve the appearance, so we give here additional examples for fine tuning.

Apply font size and color

---

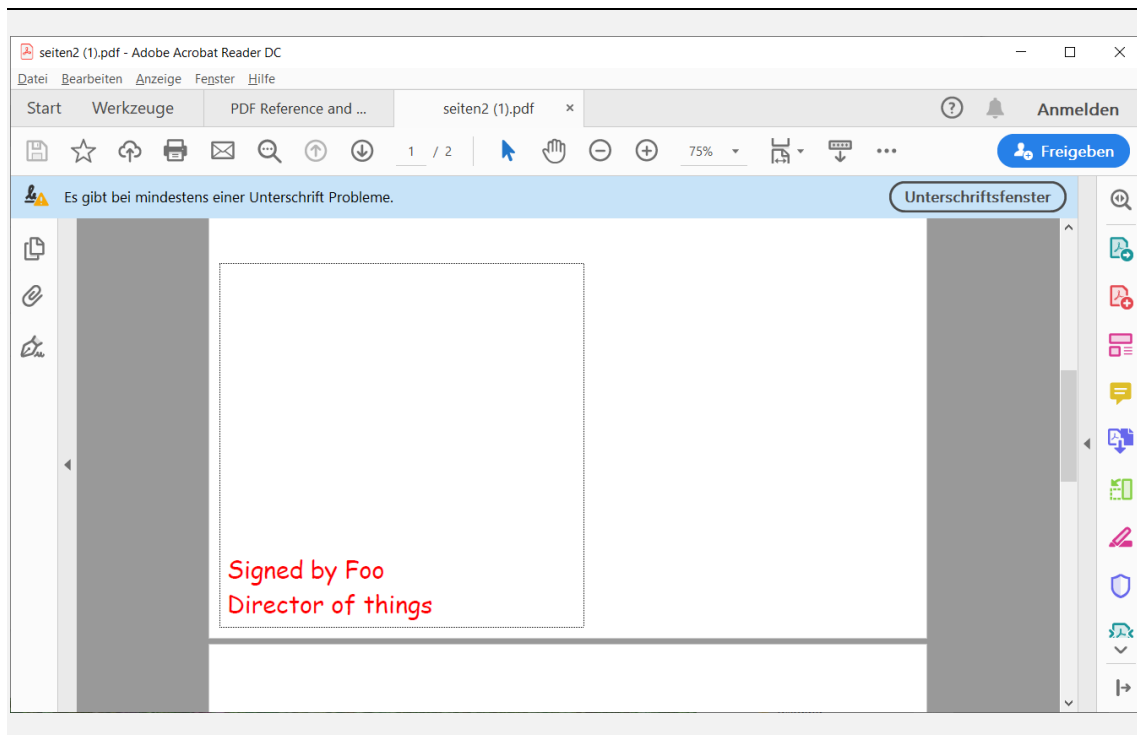
**service call**

---

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

{
  "args": {
    "documentSigner": {
      "args": {
        "field": {
          "position": "10x10",
          "size": "300x300",
          "font": {
            "fontName": "Comic Sans MS",
            "fontSize": 20,
            "fontColor": "1.0;0;0"
          }
        },
        "decorator": {
          "factory":
"de.intarsys.security.document.type.pdf.signature.ExtendedDecoratorFactory",
          "args": {
            "text": "Signed by Foo\nDirector of things",
            "textScaleWhen": "never",
            "textHAlign": "left",
            "textVAlign": "bottom"
          }
        }
      }
    }
  },
  ...
}
```

Here we set an explicit `fontSize` – it is important to pair this with the `"textScaleWhen=never"` option from the decorator to stop him from zooming the text.



#### 4.3.6 Dynamic (floating) fields

An especially interesting feature with visible signatures is the ability to dynamically define field parameters.

Imagine a word document that should have a signature field at the end of the last paragraph. It is hard to accept for your marketing department to put this at a static position, maybe sometimes overlapping with the document text.

The solution is to extract tags from the document itself. Detailed information for this you can find in [1] and [2].

First, you will need to instrument your source document with "invisible text" – you can use white foreground on white background for this. You embed tags, enclosed by the escape characters "@@". Within the escapes you include text in the format "key=value".

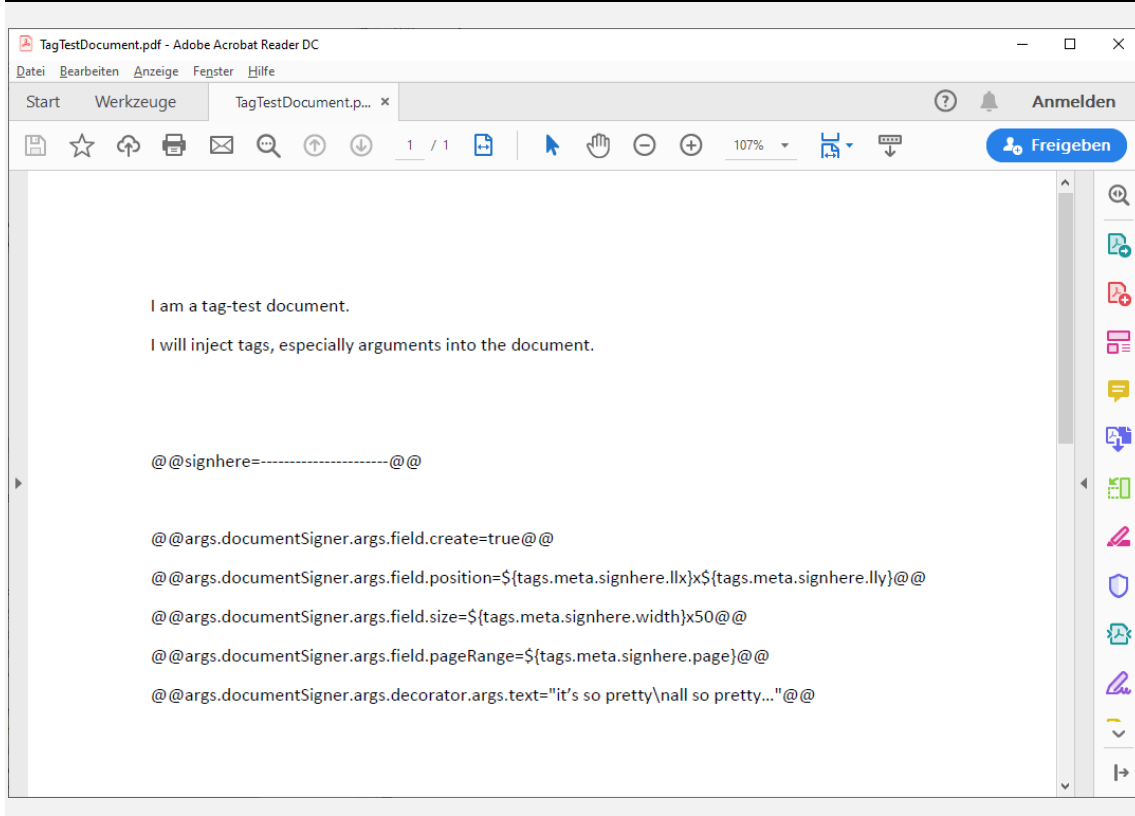
You start with the marker field itself, let's say "signhere". This tag is only there to define a location for the signature.

Then you add a bunch of tags that define arguments to the "documentSigner". These arguments references meta data derived from the initial "signhere" tag. The meta information used here is

- tags.meta.signhere.llx  
"lower left x" coordinate of the field "signhere"
- tags.meta.signhere.lly  
"lower left y" coordinate of the field "signhere"

- tags.meta.signhere.width  
"width" of the field "signhere"
- tags.meta.signhere.height  
"height" of the field "signhere"
- tags.meta.signhere.page  
"page" of the field "signhere"

This is an example for a document – for sure you should hide the text (white on white) in real world scenarios



Now you activate tag detection for the document and the "args" will be injected in the "documentSigner" and expanded with the real tag values.

For sure the expansion will work when the arguments come from the client, too. You only need the "signhere" tag and activate the tag detection as a minimum.

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

{
  "args": {
    "documentTagDetector": {
      "factory": "de.intarsys.document.app.pdf.tags.PDFContentTagDetectorFactory",
      "args": {
        "syntax": "separated"
      }
    },
    "documentSigner": {
      "factory": "de.intarsys.security.processor.signature.DocumentSignerFactory",
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<base64 content>"
  }]
}
```

### 4.3.7 Dynamic field positioning

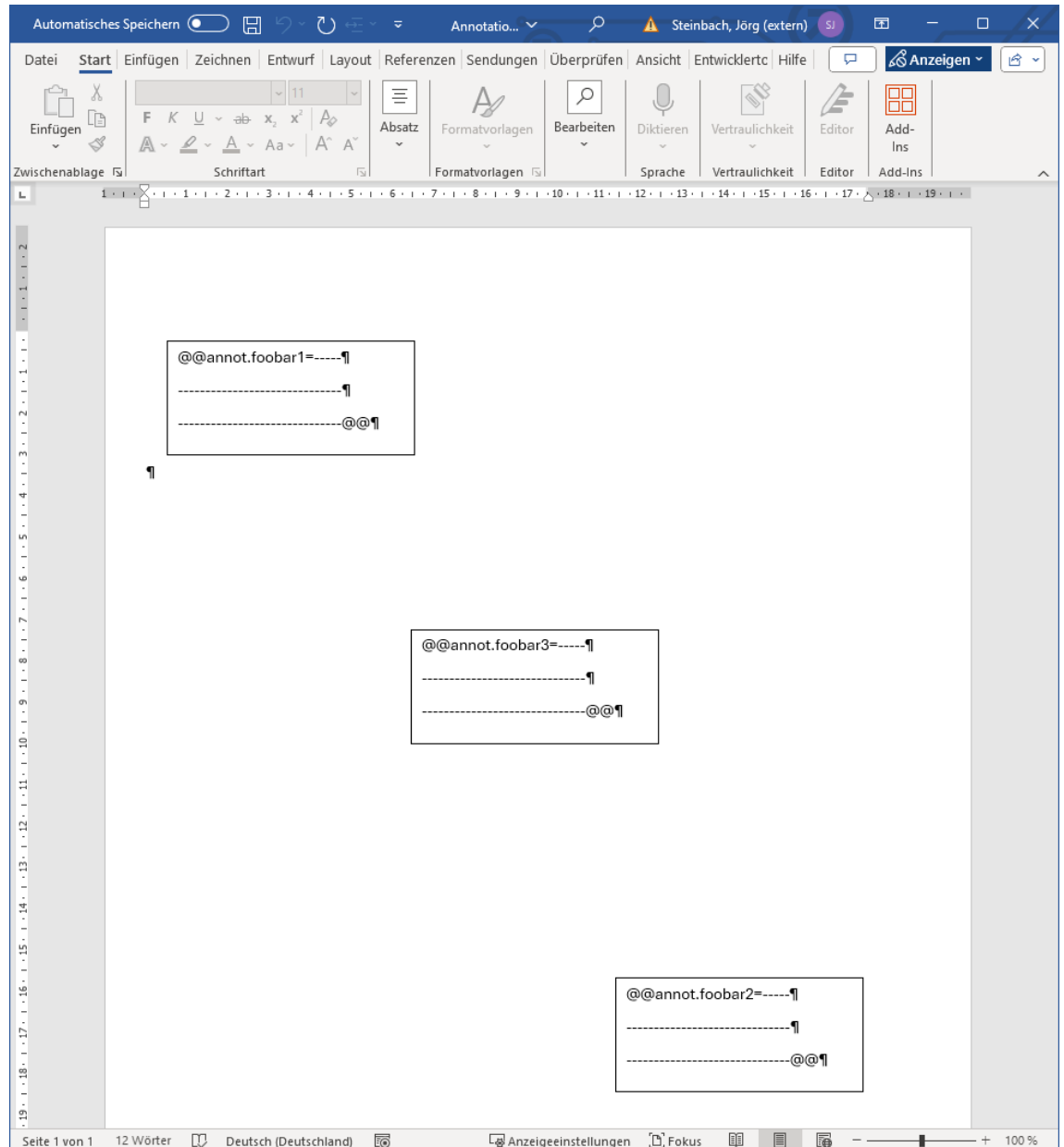
Let's assume you have a document to be signed visibly by an arbitrary number of persons and don't want to deal with the job of calculating, which field has to be filled by the next signer. Idea is to define enumerated fields or a template field with parameters, which define how to create the next field.

The following methods are available.

#### Enumerated fields

Create a PDF document with some marker fields. The positioning of the fields is up to your needs. The fields' names has to be constructed beginning with a sequence of characters and ending with an ascending, seamless numbering starting with 1.





This is the job which always chooses the next free field - with the document above up to three times:

```

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

{
  "documentTagDetector": {
    "factory": "de.intarsys.document.app.pdf.tags.PDFContentTagDetectorFactory",
    "args": {
      "syntax": "separated"
    }
  },
  "documentSigner": {
    "factory": "de.intarsys.security.processor.signature.DocumentSignerFactory",
    "args": {
      "field": {
        "name": "foobar",
        "apply": {
          "type": "enumerated"
        }
      },
      "digestSigner": {
        "factory": "de.intarsys.security.app.signature.SignerFactory",
        "args": {
          "device": "default@demo"
        }
      }
    }
  }
}

```

## Template field by tags

Create a PDF document with one marker field containing an apply parameter of type 'template'.



The following job called the first time will create a signature field at the position defined by the marker field and following calls will create signature fields relatively (deltaX, deltaY) to the first field:

## service call

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

{
  "documentTagDetector": {
    "factory": "de.intarsys.document.app.pdf.tags.PDFContentTagDetectorFactory",
    "args": {
      "syntax": "separated"
    }
  },
  "documentSigner": {
    "factory": "de.intarsys.security.processor.signature.DocumentSignerFactory",
    "args": {
      "field": {
        "name": "foobar"
      },
      "digestSigner": {
        "factory": "de.intarsys.security.app.signature.SignerFactory",
        "args": {
          "device": "default@demo"
        }
      }
    }
  }
}
```

## Template field by args

This method is the same as “template field by tags”, but doesn't need a specific PDF. Everything is defined in the call.

The first call will create a signature field at (x, y), following calls will create signature fields relatively (deltaX, deltaY) to the first field.

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

{
  "documentSigner": {
    "factory": "de.intarsys.security.processor.signature.DocumentSignerFactory",
    "args": {
      "field": {
        "name": "foo",
        "create": true,
        "x": 5,
        "y": 5,
        "width": 200,
        "height": 50,
        "apply": {
          "type": "template",
          "deltaX": "200",
          "deltaY": "500"
        }
      },
      "digestSigner": {
        "factory": "de.intarsys.security.app.signature.SignerFactory",
        "args": {
          "device": "default@demo"
        }
      }
    }
  }
}
```

### 4.3.8 Shape based appearance

If the simple extended decorator approach does not fit your complex design needs, you can revert to individually define your graphics primitives yourself. Detailed information and examples can be found in [3].

## 4.4 PDF with timestamp

Adding a timestamp to a signature is about adding evidence that the signer has performed the signature at a certain point in time. Do not mix this up with a "document timestamp" which is only declaring that this document existed in a certain state at a certain point in time.

To add a timestamp to your signature, you must define a "timestampDevice". Available timestamp devices are documented in [1].

Keep in mind that there are different sources of "time" for a PDF document:

- there is the PDF last modification timestamp
- we have a signature date field within the PDF signature object (/M), derived from local time
- we have a "SigningTime" signed attribute in the CMS, derived from local time.

- we have a "Timestamp" unsigned attribute in the CMS, which is determined by the "timestampDevice"

This example uses a free timestamp server:

#### 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"
          }
        },
        "timestampDevice": {
          "factory":
"de.intarsys.security.device.httptimestamp.device.HttpTimestampDeviceProvider",
          "args": {
            "url": "https://freetsa.org/tsr"
          }
        }
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.pdf",
      "content": "<base64 content>"
    }
  ]
}
```

The "timestampDevice" argument will work only if the CMS for the signature is created within the gears server. Certain signature devices (like AIS), creating ready to use CMS structures, are not affected by this argument. These devices **may** support other internal arguments for timestamp creation.

## 4.5 XAdES, enveloped signature (internal)

Creating an XML signature is a little bit different from PDF or CMS. For PDF and CMS exist well defined "defaults" what to do when we encounter a signature operation with no more context information.

With XML, we depend on the target scheme that is to be created – creating an XML signature at the root, using some random transformation, is just as good or bad as interpreting the whole document as binary and creating a detached signature – and this is what we decided to define as the default. So, an XML document is by default handled just like any other binary document and will create a detached CMS signature.

To switch to an XML DSig signature you will have to explicitly request this signature method via arguments. The signature itself is automatically

created in the XAdES format using XML DSig 1.1. Canonicalization is applied by default.

#### 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.document.type.xml.signature.XMLDocumentInternalSignerFactory",
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.xml",
    "content": "PGZvby8+" // this is <foo/>
  }
]
```

## 4.6 XAdES-T

Adding a timestamp is basically the same procedure as described in 4.4 PDF with timestamp.

To add a timestamp to your signature, you must define a "timestampDevice". Available timestamp devices are documented in [1]. Eventually this will add the XAdES defined property "SignatureTimeStamp".

This example uses a free timestamp server:

#### 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.document.type.xml.signature.XMLDocumentInternalSignerFactory",
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        },
        "timestampDevice": {
          "factory":
"de.intarsys.security.device.httptimestamp.device.HttpTimestampDeviceProvider",
          "args": {
            "url": "https://freetsa.org/tsr"
          }
        }
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.xml",
      "content": "PGZvby8+" // this is <foo/>
    }
  ]
}
```

## 4.7 XAdES-LT

The "LT" conformance level adds long term validation data to the signature in addition to the "T" conformance level.

### 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.document.type.xml.signature.XMLDocumentInternalSignerFactory",
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        },
        "timestampDevice": {
          "factory":
"de.intarsys.security.device.httptimestamp.device.HttpTimestampDeviceProvider",
          "args": {
            "url": "https://freetsa.org/tsr"
          }
        }
      },
      "includeRevocationInfo": "all"
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.xml",
    "content": "PGZvby8+" // this is <foo/>
  }]
}
```

## 4.8 Multiple internal XML Signatures

To apply multiple signatures in parallel, ignoring signatures already applied, you have to define your own reference ("reference" is the definition of a subtree of the original document that is included in the signature).

The reference applied in the example below says:

Take anything below the document root, filter away any element that is an XML DSig "Signature" and apply canonicalization after that.



## service call

```

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

{
  "args": {
    "documentSigner": {
      "factory": "de.intarsys.security.document.type.xml.signature.XMLDocumentInternalSignerFactory",
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        },
        "references": [
          {
            "transforms": [
              {
                "algorithm": "http://www.w3.org/2002/06/xmldsig-filter2",
                "filter": "subtract",
                "expression": "//*[namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and local-name()='Signature']"
              },
              {
                "algorithm": "http://www.w3.org/2001/10/xml-exc-c14n#"
              }
            ],
            "uri": "#xpointer(/)"
          }
        ]
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.xml",
      "content": "PGZvby8+" // this is <foo/>
    }
  ]
}

```

## 4.9 XAdES, detached signature

You can create a "detached" signature in XML (XAdES) format, too. You explicitly request this signature method via arguments. The signature itself is automatically created in the XAdES format using XML DSig 1.1.

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.document.type.xml.signature.XMLDocumentExternalSignerFactory",
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "PGZvby8+" // this is <foo/>
  }
]
```

---

## 4.10 XAdES, enveloping signature (embedded)

An enveloping signature contains the signed content literally in the XML document. You can create this signature similar to the "detached" one by adding the flag "embedDocument" to the arguments.

The signature itself is automatically created in the XAdES format using XML DSig 1.1.

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.document.type.xml.signature.XMLDocumentExternalSignerFactory",
      "args": {
        "embedDocument": true,
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "PGZvby8+" // this is <foo/>
  }
]
```

## 4.11 CMS signature

A CMS signature is by default selected if the document type does not support an internal signature (like PDF). To document or enforce this behavior you can always request a CMS signature explicitly by using the “documentSigner.factory”

```
de.intarsys.security.document.type.pkcs7.signature.PKCS7DocumentSignerFactory
```

### 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.document.type.pkcs7.signature.PKCS7DocumentSignerFactory",
    "args": {
      "digestSigner": {
        "factory": "de.intarsys.security.app.signature.SignerFactory",
        "args": {
          "device": "default@demo"
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "Zm9v" // this is "foo"
  }
]
```

## 4.12 CMS multiple signatures

If you need to create multiple CMS signatures on the same target document, you must provide the existing CMS as an attachment to the document to be signed.

### 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.document.type.pkcs7.signature.PKCS7DocumentSignerFactory",
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "Zm9v", // this is "foo"
    "attachments": [
      {
        "type": "d",
        "name": "test.txt.p7s",
        "content": "<base64 encoded CMS>"
      }
    ]
  }
]
```

For the naming of the attachment and the CMS signature created see the respective documentation in [1].

By default you need to append “p7s” to the document name to indicate this is the CMS signature that belongs to the target document.

## 4.13 Digest signature

There are times when the document is not able (e.g. because of size restrictions) or allowed (e.g. because of security considerations) to be sent to the server completely.

With gears you can send only the hash value and will receive a signature that can be incorporated into the document locally.

The hash value must be a "binary document" with a valid ASN.1 encoded digest (take care to use the .digest extension).

The result is a CAdES compliant CMS data structure.

---

**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"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.digest",
    "content": "<base64 content>"
  }
]
```

## 4.14 Document specific arguments

For your convenience we repeat here a chapter from the manual that might be missed – how to provide individual information, especially args, with an individual document in the document list. As an example, we use the "signer/create" call.

The "signer/create" call is defined using the collection of documents along with the "documentSigner" arguments that are applied to each of them.

Example

## service call

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

{
  "documents": [{
    "type": "d",
    "name": "mydoc.txt",
    "content": "<base64 content>"
  }
],
  "args": {
    "documentSigner": {
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        }
      }
    }
  }
}
```

Now we may have per-document requirements, e.g. to define a specific label for each document.

To support this, before performing the signature, all tags starting with "args.documentSigner.args." are extracted and merged with the overall signature arguments (be aware that this usage may interfere with authorization configuration that may disallow args coming from the client).

Example

## service call

```

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

{
  "documents": [{
    "type": "d",
    "name": "mydoc.txt",
    "content": "<base64 content>",
    "properties": {
      "tags": {
        "args": {
          "documentSigner": {
            "args": {
              "decorator": {
                "args": {
                  "text": "My document"
                }
              }
            }
          }
        }
      }
    }, {
      "type": "d",
      "name": "yourdoc.txt",
      "content": "<base64 content>",
      "properties": {
        "tags": {
          "args": {
            "documentSigner": {
              "args": {
                "decorator": {
                  "args": {
                    "text": "Your document"
                  }
                }
              }
            }
          }
        }
      }
    }
  ], {
    "args": {
      "documentSigner": {
        "args": {
          "digestSigner": {
            "factory": "de.intarsys.security.app.signature.SignerFactory",
            "args": {
              "device": "default@demo"
            }
          },
          "decorator": {
            "factory": "de.intarsys.security.document.type.pdf.signature.ExtendedDecoratorFactory",
            "args": {
              "text": "Default text"
            }
          }
        }
      }
    }
  }
}

```

In this example we provide the argument **documentSigner.args.decorator.args.text** for each of the input documents. It will automatically get merged in the actual arguments because of the



position in the **tags.args** properties, overwriting the default value "Default text" in the argument template.

## 4.15 Document specific tags

Remember that instead of direct arguments you may have added plain tags to the document. These can be addressed later in a string expansion expression

Example

service call

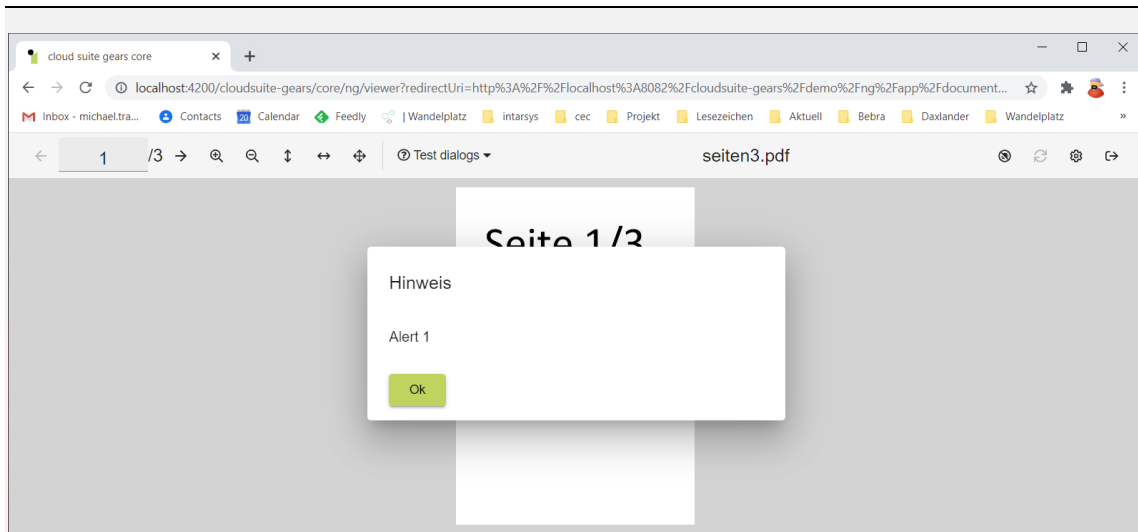
```
POST /cloudsuite-gears/core/api/v1/flow/signer/create HTTP/1.1
Content-Type: application/json

{
  "documents": [{
    "type": "d",
    "name": "mydoc.txt",
    "content": "<base64 content>",
    "properties": {
      "tags": {
        "foo": "My document"
      }
    }
  }, {
    "type": "d",
    "name": "yourdoc.txt",
    "content": "<base64 content>",
    "properties": {
      "tags": {
        "foo": "Your document"
      }
    }
  }
],
  "args": {
    "documentSigner": {
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        },
        "decorator": {
          "factory":
"de.intarsys.security.document.type.pdf.signature.ExtendedDecoratorFactory",
          "args": {
            "text": "this is ${tags.foo}"
          }
        }
      }
    }
  }
}
```

## 5. Viewer recipes

### 5.1 Add alert dialog

We want to show feedback to the user using a dialog box like this:



You need to

- create a Spring definition file in your configuration folder
- create a viewer configuration bean in the definition file
- define a widget in the viewer configuration
- call "viewer/create" using the configuration you just defined

Complete bean configuration example:

## Spring XML file

```

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:w="http://www.intarsys.de/schema/widget"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.intarsys.de/schema/widget http://www.intarsys.de/schema/widget/widget.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
">

<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="myConfig" />
  <property name="widgets">
    <list>
      <w:widget parent="de.intarsys.widget.toolbar.additions" label="MyAlert">
        <w:on event="select">
          <w:action factory="Alert">
            <entry key="message" value="Alert 1" />
          </w:action>
        </w:on>
      </w:widget>
    </list>
  </property>
</bean>

</beans>

```

To show a viewer with this button added to the toolbar, use the "myConfig" viewer configuration.

When you press the button, a non-blocking modal dialog is displayed.

## 5.2 Add multi action button

You can define an action that is composed of multiple other actions.

You need to

- create a Spring definition file in your configuration folder
- create a viewer configuration bean in the definition file
- define a widget in the viewer configuration
- call "viewer/create" using the configuration you just defined

Complete bean configuration example:

## Spring XML file

```

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:w="http://www.intarsys.de/schema/widget"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.intarsys.de/schema/widget http://www.intarsys.de/schema/widget/widget.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
">

<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="myConfig" />
  <property name="widgets">
    <list>
      <w:widget parent="de.intarsys.widget.toolbar.additions" label="MySign">
        <w:on event="select">
          <w:action factory="Alert">
            <entry key="message" value="I will sign now" />
          </w:action>
          <w:action factory="Sign">
            <entry key="requireField" value="false" />
            <entry key="requireSignature" value="false" />
            <entry key="signerCreate.configuration"
value="'{flow.variables.signerCreate.configuration}'" />
            <entry key="signerCreate.options" value="'{flow.variables.signerCreate.options}'"
/>
            <entry key="signerCreate.args" value="'{flow.variables.signerCreate.args}'" />
          </w:action>
          <w:action factory="Alert">
            <entry key="message" value="I have signed now" />
          </w:action>
          <w:action factory="Ok">
          </w:action>
          <w:catch factory="Greet">
            <entry key="message" value="I failed" />
          </w:catch>
          <w:finally factory="Greet">
            <entry key="message" value="I finished" />
          </w:finally>
        </w:on>
      </w:widget>
    </list>
  </property>
</bean>

</beans>

```

To show a viewer with this button added to the toolbar, use the "myConfig" viewer configuration.

When you press the button, multiple actions are executed in order. In this case,

- you will first see an alert box
- then the signature is launched
- then another alert "I have signed now" will show
- then the flow is committed, initiating return to your application
- but before another alert "I finished" will show up.

After this sequence you are returned to your calling application (because of the "Ok" action).

The "w:catch" definition will trigger if a failure arises in the action sequence. For example, you can "cancel" the signature action. In this case, the actions following will no longer be performed, instead the error handler executes. The "finally" action will execute always, resulting in:

- you will first see an alert box
- then the signature is launched (you should cancel here)
- the alert "I failed" will show
- the alert "I finished" will show up.

The viewer will not be terminated.

## 5.3 Overwrite default toolbar

If you do not add any widget definitions to your viewer configuration, you will end up with the default viewer toolbar.

You know that you can add your own widgets to toolbar sections, but you can overwrite the default completely, too!

In this case, if you add any widget to the section "de.intarsys.widget.toolbar.right", any default contribution will be omitted completely.

Example

```

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:w="http://www.intarsys.de/schema/widget"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.intarsys.de/schema/widget http://www.intarsys.de/schema/widget/widget.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
">

  <bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
    <property name="id" value="myConfig" />
    <property name="widgets">
      <list>
        <w:widget parent="de.intarsys.widget.toolbar.right">
          <w:on event="select" do="Ok" />
        </w:widget>
      </list>
    </property>
  </bean>

</beans>

```

You will end up with only the "Ok" button in the right toolbar section.

## 5.4 Clear default toolbar

We have learned now how to change the appearance of a default toolbar section by simply adding widget to it yourself – all default elements will be removed.

But this still leaves the question: how to completely clear a built-in section.

You can achieve this by a simple trick: you add something invisible. In this case, you add a "space" widget to the section

"de.intarsys.widget.toolbar.right". As a result, the default content will be omitted (because you have content of your own) but the content is not visible.

Example

## Spring XML file

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:w="http://www.intarsys.de/schema/widget"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.intarsys.de/schema/widget http://www.intarsys.de/schema/widget/widget.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
">

    <bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
    <property name="id" value="myConfig" />
    <property name="widgets">
    <list>
    <w:widget parent="de.intarsys.widget.toolbar.additions"
type="de.intarsys.ui.control.Space">
    <w:property name="width" value="30px" />
    </w:widget>
    </list>
    </property>
    </bean>

</beans>
```

## 5.5 Switch off toolbars

On some devices you want to reserve the whole screen for your document.

Simple solution is to switch off the toolbars and add control overlays for page turn and exit functions.

Example

```

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:w="http://www.intarsys.de/schema/widget"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.intarsys.de/schema/widget http://www.intarsys.de/schema/widget/widget.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
">

  <bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
    <property name="id" value="myConfig" />
    <property name="widgets">
      <list>
        <w:widget id="de.intarsys.widget.toolbar">
          <w:property name="visible" value="false"/>
        </w:widget>
        <w:widget id="de.intarsys.widget.sidebar">
          <w:property name="visible" value="false"/>
        </w:widget>
        <w:widget parent="de.intarsys.widget.renderer.overlays" id="control"
type="ControlOverlay">
          <w:widget id="toolbar">
            <w:widget icon="arrow-left">
              <w:on event="select" do="SelectPagePrevious" />
            </w:widget>
            <w:widget icon="window-close" >
              <w:on event="select" do="Ok" />
            </w:widget>
            <w:widget icon="arrow-right">
              <w:on event="select" do="SelectPageNext" />
            </w:widget>
          </w:widget>
        </w:widget>
      </list>
    </property>
  </bean>
</beans>

```

## 5.6 Style toolbar buttons

Widgets with type "Button" support additional styling.



Example:

### Spring XML file

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:w="http://www.intarsys.de/schema/widget"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.intarsys.de/schema/widget http://www.intarsys.de/schema/widget/widget.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
">

<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="myConfig" />
  <property name="widgets">
    <list>
      <w:widget parent="de.intarsys.widget.toolbar.additions" type="Button">
        <w:icon descriptor="far:smile" css="background-color: red; color: white; font-size: xx-large;" />
      </w:widget>
      <w:widget parent="de.intarsys.widget.toolbar.additions" type="Button">
        <w:icon descriptor="far:smile" css="display: block; margin-bottom: -20px;" />
        <w:label message="Say hi" />
      </w:widget>
      <w:widget parent="de.intarsys.widget.toolbar.additions" type="Button">
        <w:icon descriptor="far:smile" css="color: blue;" />
        <w:label message="Say hi" css="color: green; font-family: serif; font-weight: bolder; padding-left: 5px;" />
      </w:widget>
    </list>
  </property>
</bean>

</beans>
```

Yields:



## 5.7 Annotation overlay with dynamic fields

### 5.7.1 Overview

The annotation overlay shows all (filtered) annotations from the document and allows direct interaction. In addition, virtual annotations can be defined as children of the "annotations" property.

The annotation overlay is described in detail in [2].

## 5.7.2 The "annotations" widget container

As you know from the documentation, all widgets defined in the branch "annotations" are treated as "virtual" annotations. They are not yet contained in the document, but you can interact with them in the same way as with existing annotations.

This example uses dynamic widgets to add a widget in the "annotations" branch of the annotations overlay. Remember to properly define the "annotations" container in your overlay.

You do not need a callback definition, as the events are handled by the "canvas" widget of the overlay (just the same way as plain standard annotations).

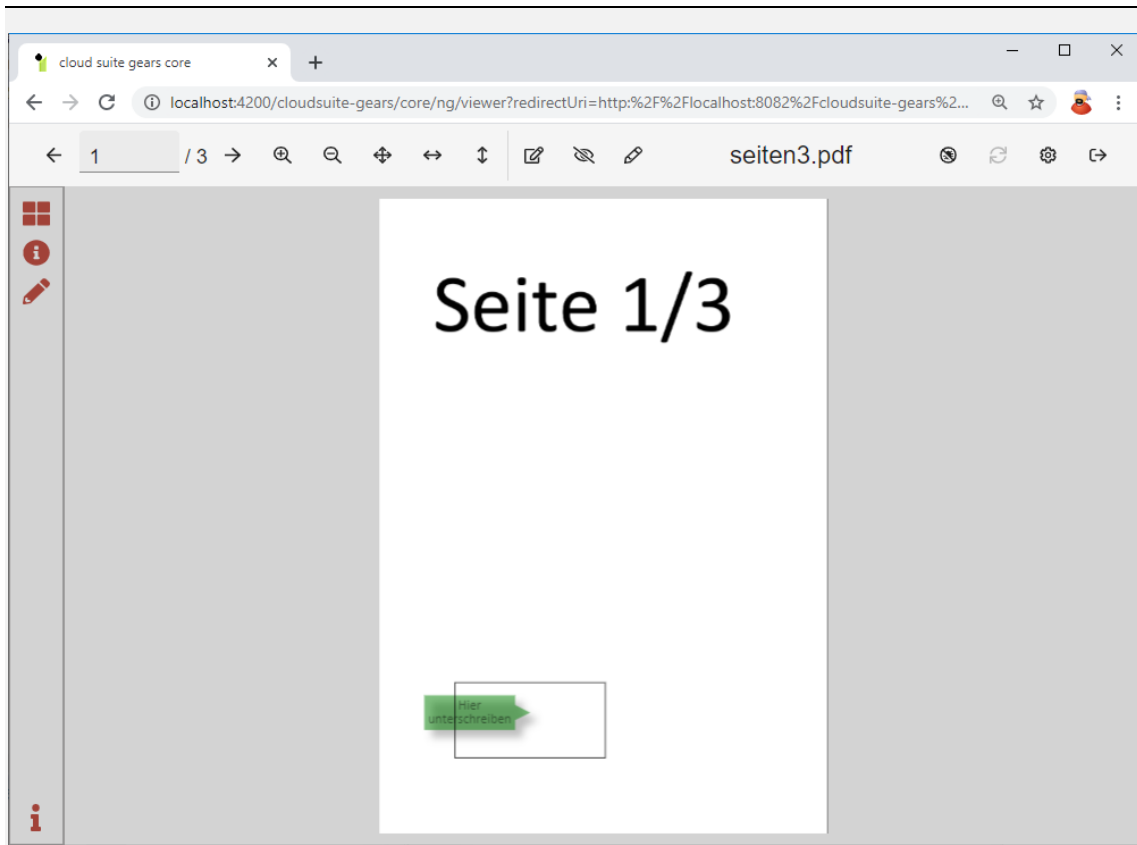
---

### Spring XML fragment

```
<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotations"
type="AnnotationsOverlay">
  <w:widget id="annotations">
    <w:widget>
      <w:property name="name" value="MySignatureField"/>
      <w:property name="type" value="Widget"/>
      <w:property name="subType" value="Sig"/>
      <w:property name="position" value="100x100"/>
      <w:property name="size" value="200x100"/>
      <w:property name="pageRange" value="all"/>
    </w:widget>
  </w:widget>
</w:widget>
```

---

The viewer shows up like this:



The virtual annotation renders on each page in the lower left, just as we have defined it.

### 5.7.3 And even more dynamic...

This is not the end of the possible combinations of gears features. Maybe you remember the "tag detection" and the "floating fields" use case from chapter "Dynamic (floating) fields"?

There we create a field with its rectangle defined within the document itself using meta tags.

Now we even have the possibility to use document defined fields in the viewer, simply by using variables in the widget definition we have made above.

This is an "imperative" approach, where you explicitly denote any signature required for the document. See the next chapter for a more declarative approach that may suit your needs.

For this example, we reuse the "TagTestDocument" from the gears examples folder.

**Spring XML fragment**

```
<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotations"
type="AnnotationsOverlay">
  <w:widget id="annotations">
    <w:widget>
      <w:property name="name" value="MySignatureField"/>
      <w:property name="type" value="Widget"/>
      <w:property name="subType" value="Sig"/>
      <w:property name="position"
value=""?{tags.meta.signhere.llx}*?{tags.meta.signhere.lly}"/>
      <w:property name="size" value=""?{tags.meta.signhere.width}*50"/>
      <w:property name="pageRange" value=""?{tags.meta.signhere.page}"/>
    </w:widget>
  </w:widget>
</w:widget>
```

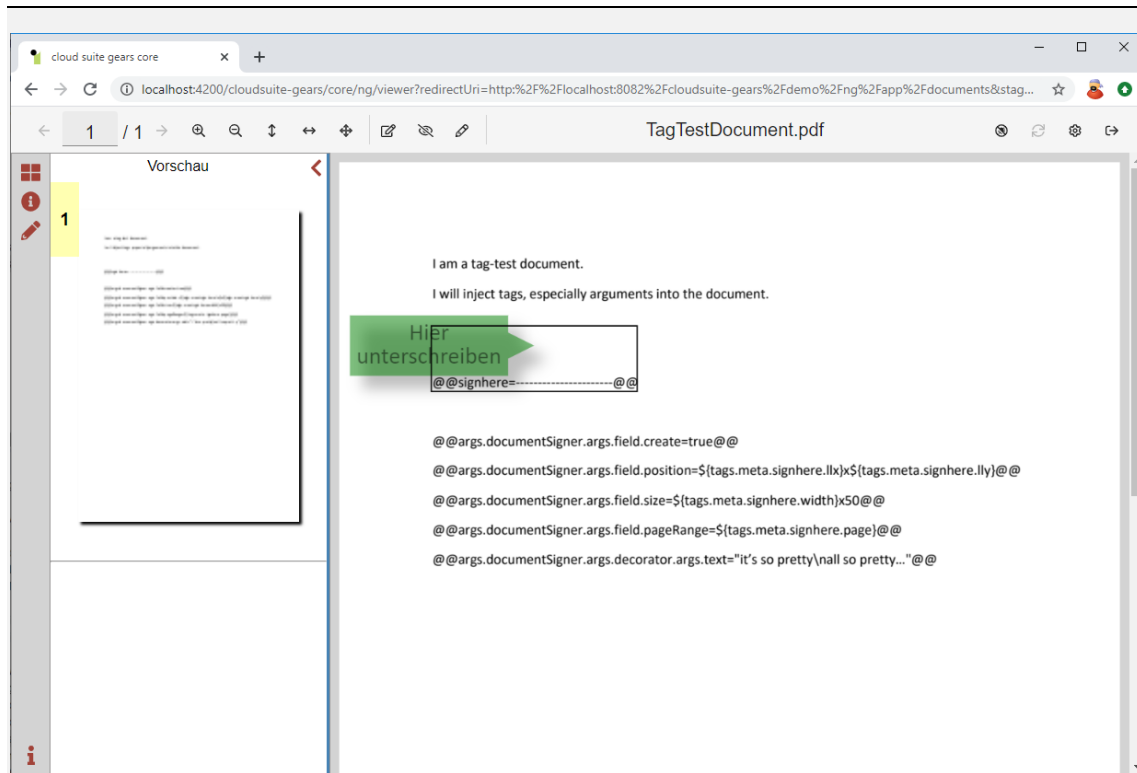
In addition, you need to activate tag detection by adding the "documentTagDetector". Then you can access the tag meta data in your widget definition...

**Example call****service call**

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

{
  "args": {
    "documentTagDetector": {
      "factory": "de.intarsys.document.app.pdf.tags.PDFContentTagDetectorFactory",
      "args": {
        "syntax": "separated"
      }
    },
  },
  "document": {
    "type": "d",
    "name": "test.pdf",
    "content": "<base64 content>"
  }
}
```

This is what you will see – nice, isn't it.



You should keep in mind an important caveat, though:

Arguments in the document overwrite arguments in the call!

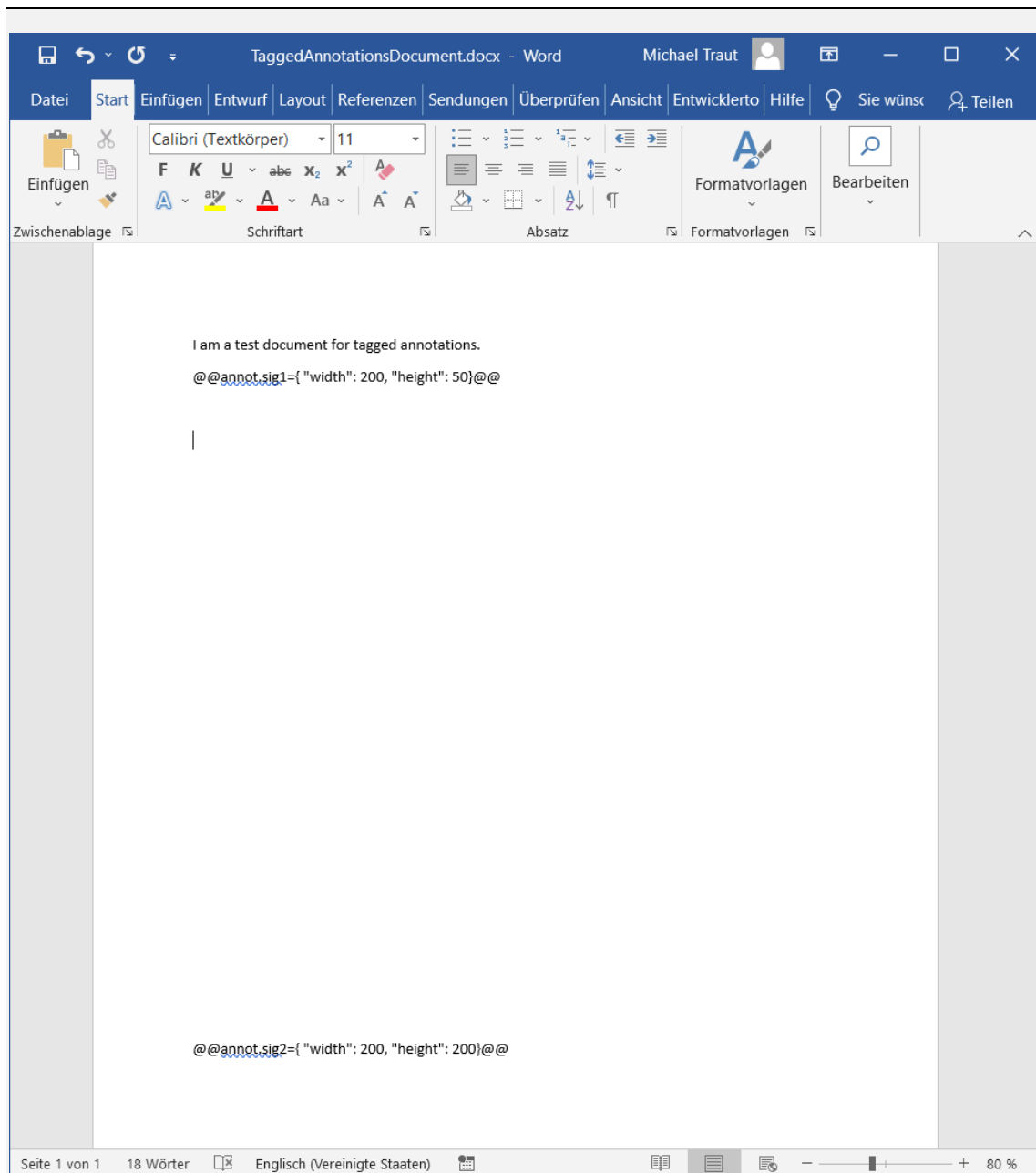
So if, like in the example above, you re-use a document that will be used in direct signer scenarios, too (without viewer interaction) and you have embedded arguments for "blind-processing" (like "args.documentSigner..." in this example) you must be sure that the "documentTagDetector" is **off** for the signer called from the viewer. The arguments build from the annotation widget would be overwritten otherwise.

## 5.7.4 Declarative annotations

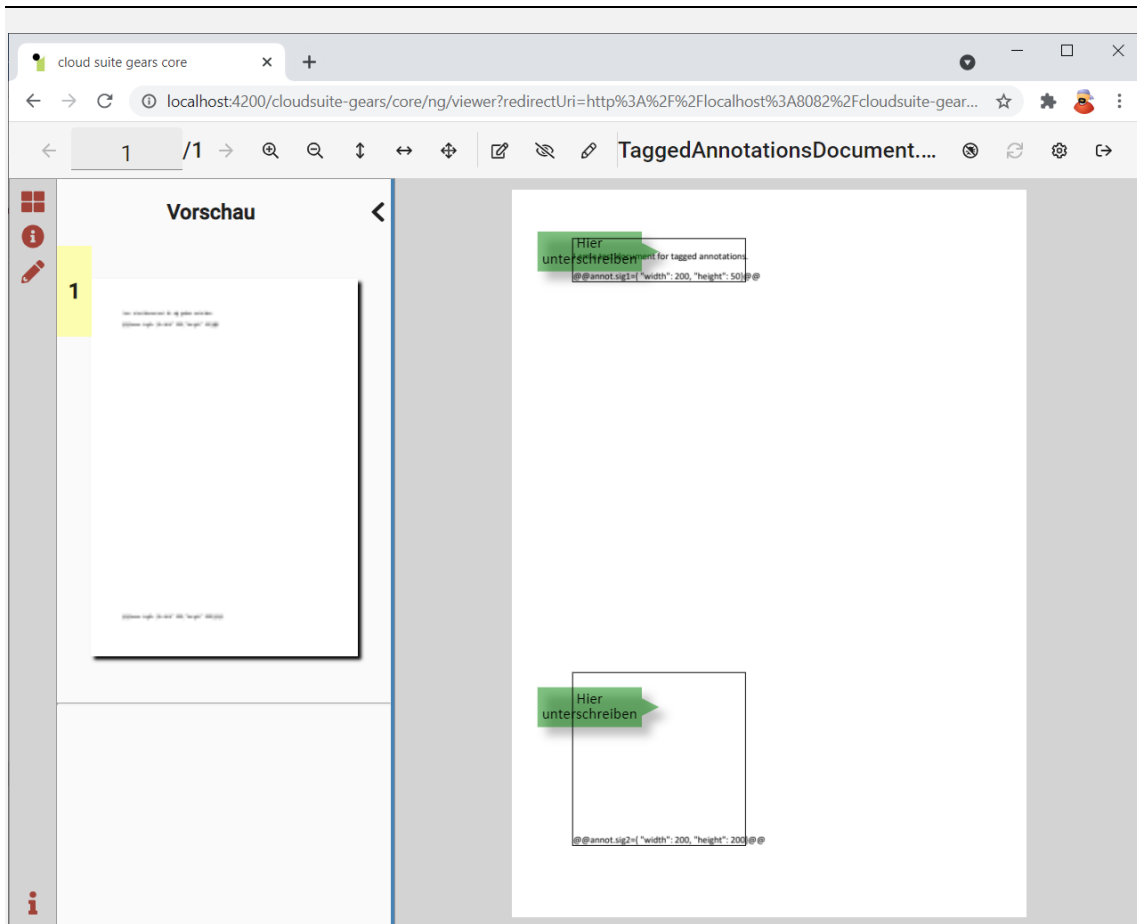
A more declarative approach allows to define "virtual" annotations in the document without the need to go for sophisticated PDF tools. It is based on document tags, exactly like in the chapter before.

Now you include a tag with properties that completely define the new annotation. The AnnotationOverlay will automatically collect these virtual annotations, just as if they were PDF annotations.

Consider this word document:



Now create a PDF - when scanned with the PDF tag detector active, the viewer will start like this (when the AnnotationOverlay is active).



This document in Word and PDF format is part of the gears deployment (in the "example/documents/tag" folder).

### 5.7.5 Hints

If you add widgets to the "annotations" branch of the overlay, you must ensure that this branch is properly defined in your overlay definition. Normally this is just an empty widget block.

#### Spring XML fragment

```
<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotations"
type="AnnotationsOverlay">
  <w:widget id="annotations">
  </w:widget>
...

```

The "callbacks" for the widget definition does not play any role here. An "annotations" subwidget automatically delegates its trigger to the "canvas".

## 5.8 Request placement of a signature field

You can configure a widget to start a signature process by defining a visible signature field from scratch, optionally predefining field dimensions and location.

To achieve this, you need to

- define a widget configuration in a Spring definition file, triggering a Sign action and
- set the action property “requireField” to true and
- optionally add a property “field” with a rectangle definition String as denoted below.

Depending on the “field” value, rectangle dimensions and initial placement will vary as follows:

- “field” is undefined
  - The user defines field rectangle from scratch within the current page’s boundaries.
- “field” is a String containing height and width parameters
  - The rectangle is initialized to the defined dimensions and placed in the default location (upper-left viewer corner)
  - The user may drag the field to a location of his choice within the current page’s boundaries.
- “field” is a String containing x, y, height and width parameters
  - The rectangle is initialized to the defined dimensions and placed in the page location given by x and y.
  - The user may drag the field to a location of his choice within the current page’s boundaries.

Configuration example,



## Spring XML file

```

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:w="http://www.intarsys.de/schema/widget"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.intarsys.de/schema/widget http://www.intarsys.de/schema/widget/widget.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
">

  <bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
    <property name="id" value="myConfig" />
    <property name="widgets">
      <list>
        <w:widget parent="de.intarsys.widget.toolbar.additions"
id="signVisibleWithPredefinedRect">
          <w:on event="select" do="Sign">
            <entry key="requireField" value="true"/>
            <entry key="requireSignature" value="true"/>
            <entry key="requireInput" value="true"/>
            <entry key="inputTitle"
value="?{nlsmg.de.intarsys.gears.core.demo.messages#btnSignVisible.inputTitle}"/>
            <entry key="pageRange" value="current"/>
            <entry key="field" value="x=20;y=10;width=100;height=60"/>
            <entry key="signerCreate" value="?{flow.variables.signerCreate}"/>
          </w:on>
        </w:widget>
      </list>
    </property>
  </bean>
</beans>

```

## 5.9 Collect multiple information chunks for a complex signature appearance

For your workflow you may want to collect multiple text or image chunks that will find their way into the signature appearance.

You can implement this requirement by a sequence of actions, each pushing its result to a different execution context property and then injecting the values into the signature call.

Here is an example configuration that can be used for the viewer:

```

{
  "widgets": [{
    "label": "Sign",
    "parent": "de.intarsys.widget.toolbar.additions",
    "callbacks": {
      "select": [{
        "factory": "CollectDepiction",
        "args": {
          "=": "myDepiction"
        }
      }, {
        "factory": "CollectInput",
        "args": {
          "title": "collect input 1",
          "=": "myText1"
        }
      }, {
        "factory": "CollectInput",
        "args": {
          "title": "collect input 2",
          "=": "myText2"
        }
      }, {
        "factory": "Sign",
        "args": {
          "signerCreate": "${flow.variables.signerCreate}",
          "signerCreate.args.documentSigner.args": {
            "field": {
              "position": "10x10",
              "size": "200x100"
            },
            "decorator": {
              "factory":
"de.intarsys.security.document.type.pdf.signature.shape.ShapeDecoratorFactory",
              "args": {
                "shapes": [{
                  "type": "rectangle",
                  "position": "0*0",
                  "size": "100%*100%",
                  "strokeWidth": 5
                }, {
                  "type": "text",
                  "text?": "myText1",
                  "position": "2*2"
                }, {
                  "type": "text",
                  "text?": "myText2",
                  "position": "2*50"
                }, {
                  "type": "canvas",
                  "content?": "myDepiction",
                  "position": "100*2",
                  "size": "50%x100%"
                }
              ]
            }
          }
        }
      }
    ]
  }
}

```

First, you assign the result of the "CollectXXX" actions using the "=" syntax. In the shapes then you access this context using the "key?" syntax – that's all.

## 5.10 Use a toggle button and session state

This is a simple example of how to combine a toggle button and internal session state.

A toggle button can attach 3 standard callbacks instead of only the "select" callback:

- selectChecked
- selectUnchecked
- isChecked

```

{
  "actions": [{
    "id": "doUnchecked",
    "try": [{
      "factory": "Alert",
      "args": {
        "message": "select unchecked"
      }
    }, {
      "factory": "SetValue",
      "args": {
        "name": "$session.myvar",
        "value": true
      }
    }
  ]
}, {
  "id": "doChecked",
  "try": [{
    "factory": "Alert",
    "args": {
      "message": "select checked"
    }
  }, {
    "factory": "SetValue",
    "args": {
      "name": "$session.myvar",
      "value": false
    }
  ]
}, {
  "factory": "GetValue",
  "args": {
    "id": "isChecked",
    "name": "$session.myvar"
  }
}
],
"widgets": [{
  "type": "Toggle",
  "parent": "de.intarsys.widget.toolbar.additions",
  "icon": "far:pen-square",
  "label": "toggle me",
  "tip": "toggle me and i will be a prince ...",
  "callbacks": {
    "selectChecked": "doChecked",
    "selectUnchecked": "doUnchecked",
    "checked": "isChecked"
  }
}, {
  "type": "Toggle",
  "parent": "de.intarsys.widget.toolbar.additions",
  "icon": "far:pen-square",
  "label": "i just show that state is shared!",
  "tip": "toggle me and i will be a prince ...",
  "callbacks": {
    "selectChecked": "doChecked",
    "selectUnchecked": "doUnchecked",
    "checked": "isChecked"
  }
}
]
}

```

## 6. Device recipes

### 6.1 Demo device

gears comes along with a simple device that can be very efficiently be used for testing purposes.

The demo device represents an identity local to the gears installation that is automatically created upon installation, using a RSA key pair and a self-signed certificate. The security applications of the device do not need authentication by default, so you just have to ask for a signature.

Example

---

#### service call

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

{
  "args": {
    "documentSigner": {
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@demo"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "text": "foo"
  }]
}
```

---

### 6.2 Bridge with arguments

As described in [2], you can send along arguments to the bridge and the bridglet via gears.

The arguments to the bridglet are inserted at the path

---

```
documentSigner.args.digestSigner.args.bridglet.args
```

---

Here you can use any argument you know from [1].

### Example

#### service call

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

{
  "args": {
    "documentSigner": {
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@bridge",
            "bridge": {
              "license": "<base64>"
            },
            "bridglet": {
              "args": {
                "ui": {
                  "reduced": "true",
                  "forceDocumentSelection": "true",
                  "showDocumentSelection": "true"
                }
              }
            }
          }
        }
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.txt",
      "text": "foo"
    }]
  }
}
```

## 6.3 Bridge exclude/include devices

As the bridge agent evolves, more and more devices become available (e.g. smartcards, PKCS#11 tokens, Windows crypto provider). On the downside, the end user may be prompted with a lot of principals that are not intended to be used for the current scenario.

This is why you can explicitly exclude (or include) devices.

The default is to show every supported device, with the exception of the "demo" device.

The example shows how to exclude the windows device, too.

Example

## service call

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

{
  "args": {
    "documentSigner": {
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@bridge",
            "bridglet": {
              "args": {
                "deviceProviderExclude": "windows"
              }
            }
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "text": "foo"
  }]
}
```

## 6.4 Bridge include demo device

As within gears, the demo device, signing with a local dynamic private key and a self-signed certificate is available for testing purposes. The device is excluded by default, but all you have to do is just "activate" it using the "deviceProviderInclude" argument.

Example

## service call

```

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

{
  "args": {
    "documentSigner": {
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@bridge",
            "bridglet": {
              "args": {
                "deviceProviderInclude": "demo"
              }
            }
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "text": "foo"
  }]
}

```

## 6.5 Bridge with session (comfort signature)

An interesting feature for the bridge is the "comfort signature". If you have a suitable license and a suitable token (e.g. a batch signature card), you can persist the security state between multiple bridge calls and omit PIN entry.

To do so, you must indicate the session context to the bridge, where session creation itself is lazy. This means you must invent a random session id and send it to the bridge, along with the "lazy" argument. The bridge will not find the session the first time, create one and switch to "session" mode. All security applications created from here are stored in this session.

The session id itself should really be long and random – let's say a 32-byte random string at least. Any client that is able to guess this id is able to use your security token unattended.

The session will be timed out after 5 minutes by default – then PIN re-entry is required. Depending on license constraints session disposal may occur more often, by default without license the session is disposed after 2 uses.

### Example

This example ensures that a session is lazily established that expires after 10 seconds of idle time.



## service call

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

{
  "args": {
    "documentSigner": {
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@bridge",
            "bridge": {
              "license": {
                "content": "<base64 license for session use>"
              },
              "session": {
                "id": "1111-2222-3333-4444-5555",
                "lazy": "true",
                "expire": {
                  "type": "timeout",
                  "value": "10000"
                }
              }
            }
          },
          "bridglet": {
            "args": {
              "deviceProviderInclude": "demo"
            }
          }
        }
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.txt",
      "text": "foo"
    }]
  }
}
```

## 6.6 AIS device

### 6.6.1 Look up RAS evidence id

Swisscom All-in Signing service requires the evidence id from RA service to be present in the DN for some requests. Sign Live! cloud suite gears provides a mechanism to look up its value in RA service using the provided MSISDN. If the DN contains a variable placeholder prefixed with "rasEvidence", e.g. "\${rasEvidence.evidenceId}", Sign Live! cloud suite gears calls the RA service and replaces the placeholder with the corresponding result value from the service.

```

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

{
  "args": {
    "documentSigner": {
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@ais",
            "mode": "ondemand",
            "distinguishedName" : "CN=Jane
Doe, GIVENNAME=Jane, SURNAME=Doe, C=CH, EMAILADDRESS=jane.doe@swisscom.com, SERIALNUMBER=RAS$
{rasEvidence.evidenceId}",
            "stepUpMSISDN" : "+41791234567",
            "stepUpMessage" : "Please confirm signature",
            "stepUpLanguage" : "EN"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "text": "foo"
  }]
}

```

## 6.7 Sign-Me Device

### 6.7.1 Seal creation

Within the sealing process, the sensitive cryptographic operations take place at the TSP and the partner organization operating gears needs to authorize the application of qualified seal through a two-factor authentication using a cryptographic token. We need to configure the usage of a soft token file for challenge signing during the authorization process.

Assuming that you have properly created your qualified seal at D-Trust – either using D-Trust’s CSM platform or in exchange with a sales representative – the basic steps for a quick use of your seal are

1. Place the authentication token file in the gears configuration folders.
2. Configure the sign-me default device.
3. Configure the default signer configuration “signMeQualifiedSeal”.
4. Request a seal.

The following sections will go into the details of each step.

## Token file storage

The partner system uses the token file as second factor to authorize the application of a qualified seal which is expected in the folder “\${cloudsuite.config.shared}/devices/signMe/<device-id>/authorization-tokens” by default. For example, if we use the device “myDevice@signMe”, the path to the token directory will be “\${cloudsuite.config.shared}/devices/signMe/myDevice/authorization-tokens”. You can customize the storage location by explicitly redefining the default device property “signme.authorization.tokenPath”.

## Default device configuration

Define the default device properties required to access the sign-me service using your partner account. This will most likely look like:

---

### Spring properties

```
signme.service.address=https://cloud-ref-sp.sign-me.de:443/api/v2
signme.service.username=serviceuser
signme.service.password=d83jd9fnnc3fj3i4kd

signme.partner.username=partneruser
signme.partner.password=k84jdg74hwbs9tjn4g3zsdjdjnkdocubzvfzvbzd
signme.partner.role=PARTNER
```

---

## Default signer configuration

Provide the properties for the signer configuration “signMeQualifiedSeal”, defining the authentication data for your seal.

This could likely look like:

---

### Spring properties

```
signme.sealer.authorization.tokenId=TestOrg_CSM9832.p12"
signme.sealer.authorization.tokenPassword=U4KMTL0AJ1
signme.sealer.authorization.organizationName=TestOrg
signme.sealer.text=Sealed by seal-me?{entity.nl}?{digestSigner.subject.cn}
```

---

## Seal creation request

Send a signature request, applying the signer configuration “signMeQualifiedSeal”:

---

**service call**

---

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

```
{
  "configuration": "signMeQualifiedSeal",
  "documents": [{
    "type": "d",
    "name": "testfile.pdf",
    "content": "QUJDLi4us"
  }]
}
```

That's it! You should now receive your first qualified seal.

## 7. Integration recipes

### 7.1 Log observers

Here we set up an observer that will log each license withdrawal to a dedicated file.

This is a simple observer definition:

#### Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="license"/>
  <property name="appender">
    <bean class="ch.qos.logback.core.FileAppender">
      <property name="name" value="FILE"/>
      <property name="file" value="${cloudsuite.log.dir}/license.log"/>
      <property name="append" value="true"/>
      <property name="encoder">
        <bean class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
          <property name="pattern" value="[%d{HH:mm:ss.SSS}] %msg [%args]%n"/>
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

It will log any **license** related event to a file named "license.log" in the gears log folder.

The pattern definition has a small extra: The **"args"** token is an addition to logback that resolves to a string with all observation properties, separated by ",".

This is an example for what you may see in the log file.

```
[08:20:51.959] {loaded}
[created=1619590851959;source=license;code=loaded;text={loaded};locator=C:\ProgramData\c
loudsuite\config\licenses\TEST-GEARS-Fernsignatur-10Tsd_BE-2019-3110_PRO1461-31-12-
2021.lic;product=de.intarsys.security.device.signme;validFrom=10/20/2019;validTo=12/31/2
021;state={EnumLicenseState.valid};properties=id=de.intarsys.security.device.signme;
de.intarsys.security.app.sign.account=10000;year; ]
[08:20:51.960] {loaded}
[created=1619590851959;source=license;code=loaded;text={loaded};locator=C:\ProgramData\c
loudsuite\config\licenses\TEST-GEARS-Fernsignatur-10Tsd_BE-2019-3110_PRO1461-31-12-
2021.lic;product=de.intarsys.security.device.ais;validFrom=10/20/2019;validTo=12/31/2021
;state={EnumLicenseState.valid};properties=id=de.intarsys.security.device.ais;
de.intarsys.security.app.sign.account=10000;year; ]
```

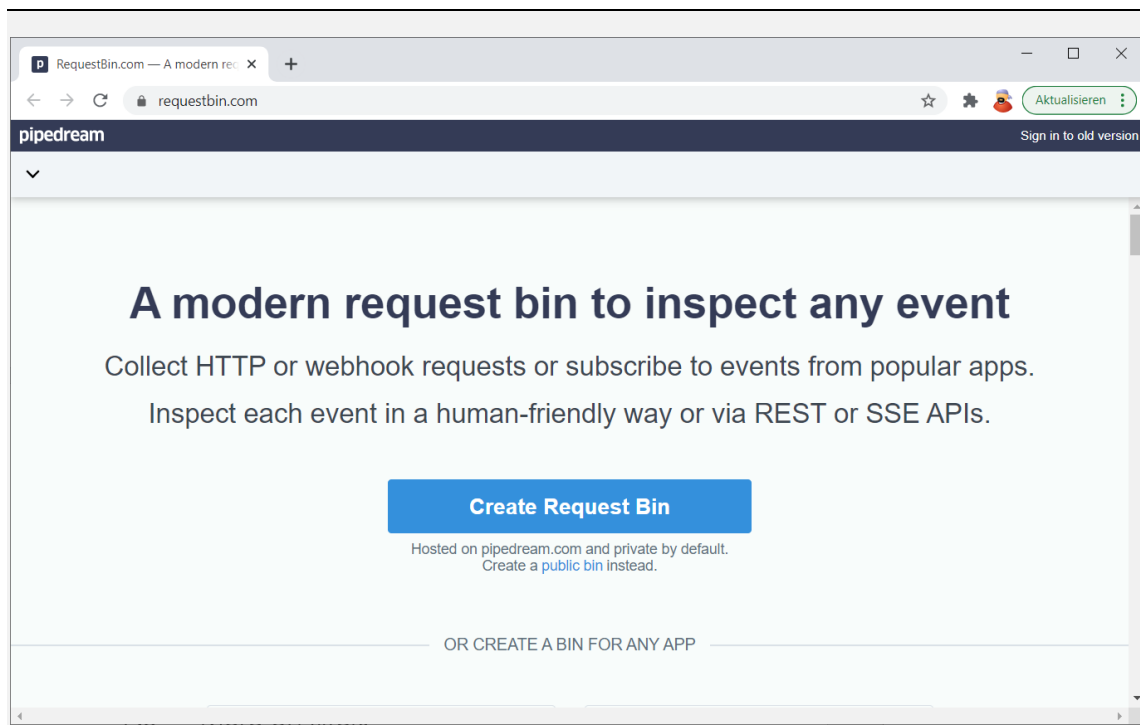
Be aware that different observations (like "loaded" and "withdraw") have different properties.

## 7.2 Webhook observers

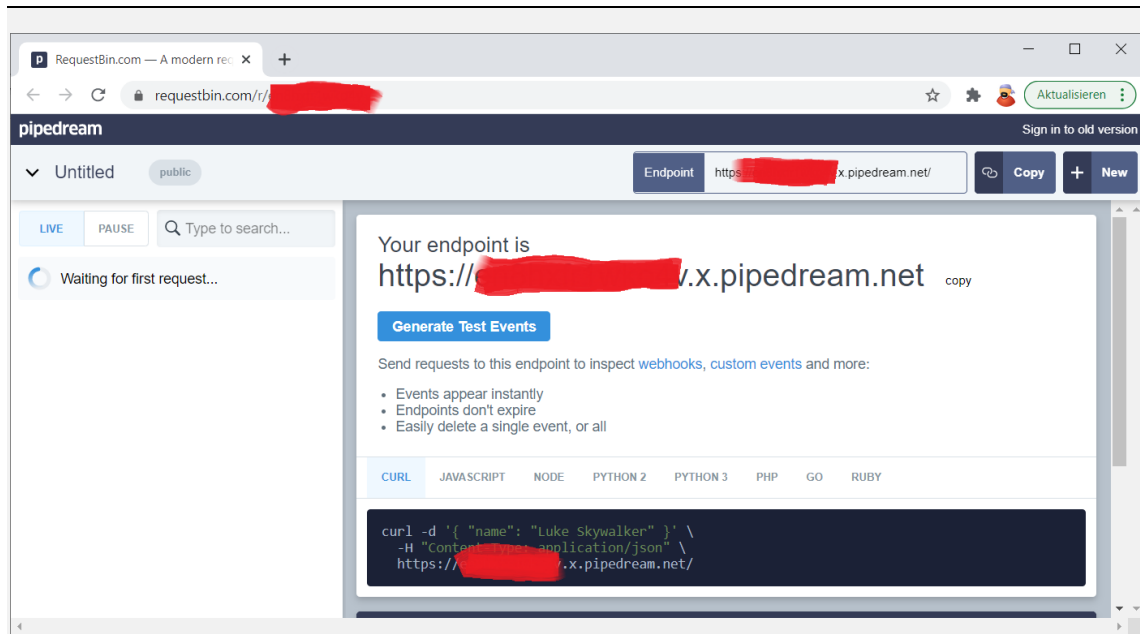
There's an ingenious tool that eases life when it comes to webhooks:

<https://requestbin.com/>

At this portal you can setup a generic endpoint that allows easy testing and inspection of your integration.



On this page you should select the small "Create a public bin" if you are not yet a registered user to proceed here.



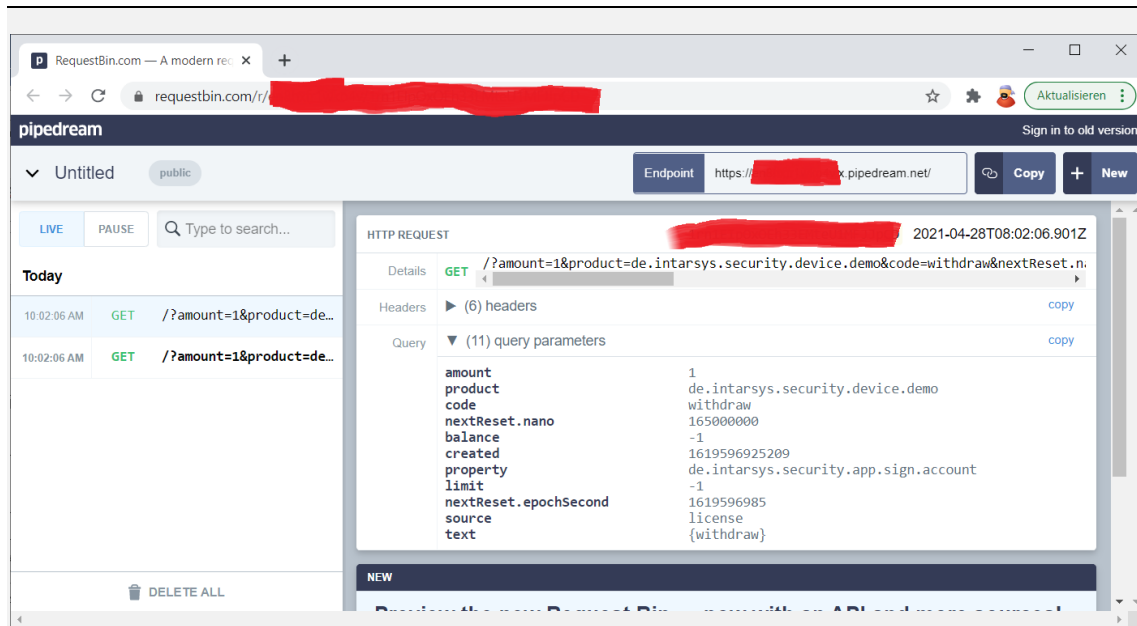
And - that's all. You have a ready to play endpoint that you can register with gears. Be aware that you should not make any assumptions regarding this endpoint's integrity or confidentiality!

Now we create an observer that pushes to this endpoint:

### Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="license"/>
  <property name="appender">
    <bean class="de.intarsys.tools.webhook.logback.WebhookAppender">
      <property name="webhook">
        <bean class="de.intarsys.tools.webhook.impl.StandardWebhookStub">
          <property name="endpointUrl" value="https://.....pipedream.net/">
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

After creating a signature, you should be able to see the event in the browser like this:



## 7.3 More on filters

Filters allow to specify which observations the observer wants to receive.

Here we create an example that only sends "device" events that carry the code "sign.signme.\*", which is a regular expression for "any event that starts with "sign.signme"".

This time we have opted for a POST style binding.

### Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device"/>
  <property name="filter">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AcceptPredicate">
        <property name="expr" value="{observation.code}"/>
        <property name="pattern" value="sign.signme.*"/>
      </bean>
    </list>
  </property>
  <property name="appender">
    <bean class="de.intarsys.tools.webhook.logback.WebhookAppender">
      <property name="webhook">
        <bean class="de.intarsys.tools.webhook.impl.StandardWebhookStub">
          <property name="method" value="POST"/>
          <property name="endpointUrl" value="https://...pipedream.net/">
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

Some important hints:

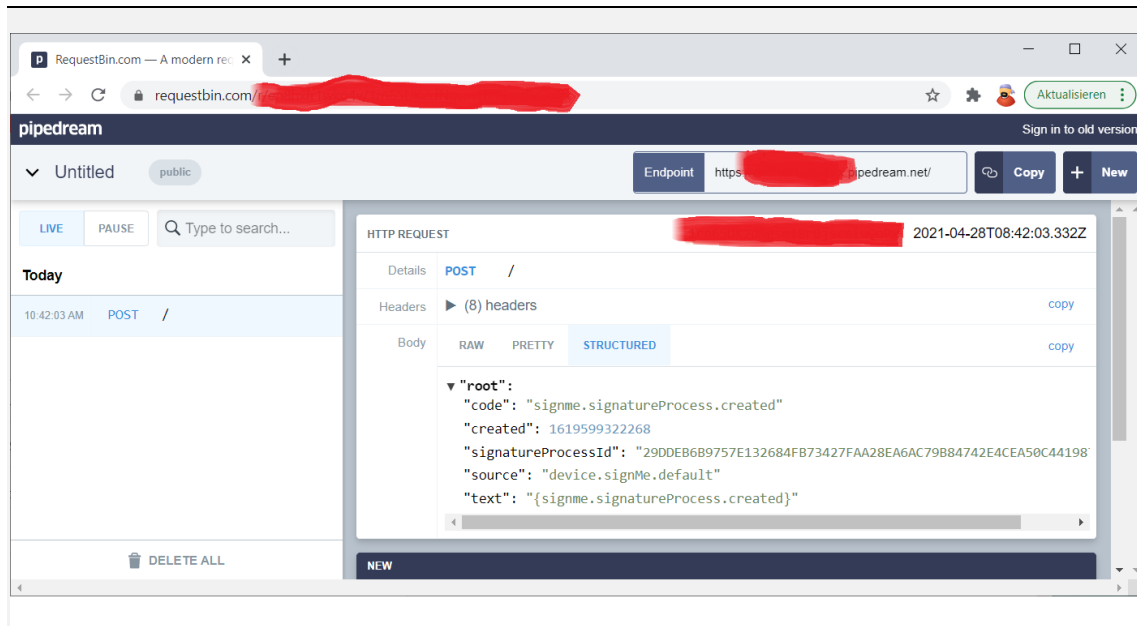
- It is here very important to differentiate between the standard spring string expansion (using \${}) and the gears dynamic expansion



(using `?{}`). If you would use `${}` here, spring would expand at system startup time, resulting in undefined or useless static content. Using `?{}` would result in a new expansion context on every use!

- **pattern** is a regular expression pattern. Be sure to correctly escape special characters.
- **filter** may be a single predicate or list of predicates.

And this is the POST finally on requestbin:



## 7.4 More on views

A view allows to select only dedicated properties, and, more importantly, to inject any other context information that is available via string expansion.

Here we present a particularly interesting scenario: we add information that was originally provided in the signature flow creation via the "variables" option.

## Spring XML fragment

```

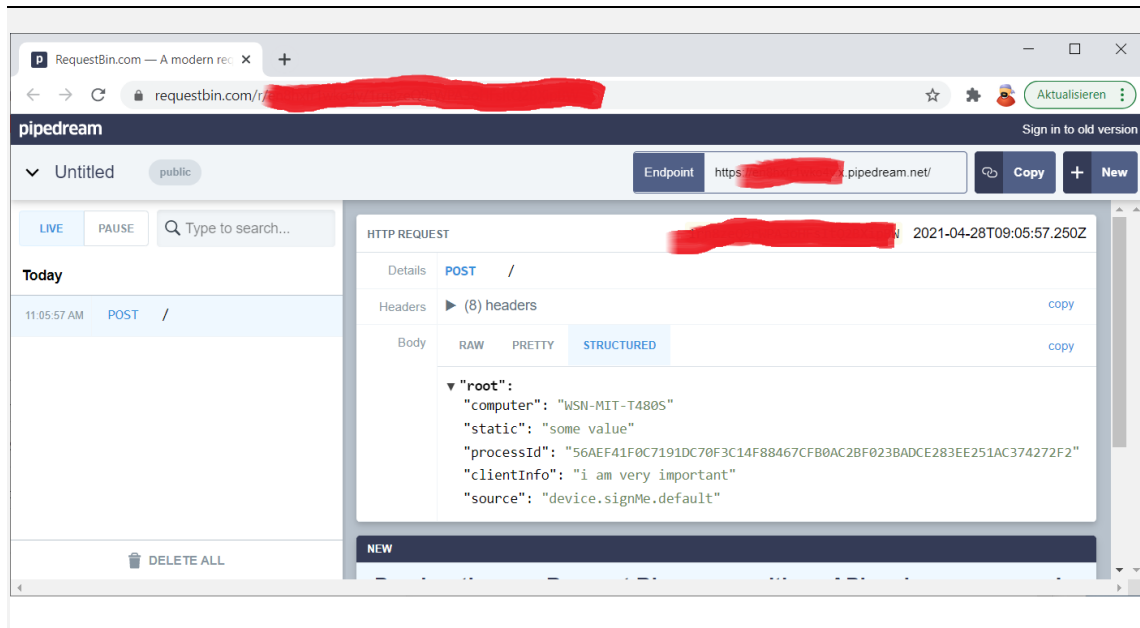
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device"/>
  <property name="filter">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AcceptPredicate">
        <property name="expr" value="{observation.code}"/>
        <property name="pattern" value="sign.signme\.signatureProcess\.created"/>
      </bean>
    </list>
  </property>
  <property name="view">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="static"/>
        <property name="value" value="some value"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="source"/>
        <property name="value" value="{observation.source}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="processId"/>
        <property name="value" value="{observation.signatureProcessId}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="clientInfo"/>
        <property name="value" value="{flow.variables.clientInfo}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="computer"/>
        <property name="value" value="{system.getenv.COMPUTERNAME}"/>
      </bean>
    </list>
  </property>
  <property name="appender">
    <bean class="de.intarsys.tools.webhook.logback.WebhookAppender">
      <property name="webhook">
        <bean class="de.intarsys.tools.webhook.impl.StandardWebhookStub">
          <property name="method" value="POST"/>
          <property name="endpointUrl" value="https://...pipedream.net"/>
        </bean>
      </property>
    </bean>
  </property>
</bean>

```

Again, some hints:

- **view** is a statement or a list of statements
- all statement results make up the view properties
- be aware that the default (all properties) is no longer active, you will have to include actively the default properties with an "IncludeStatement".
- Again, be sure to use the `{}` notation.

Here's the result:



## 7.5 More on log files

When defining observers that are backed up by plain logback file appenders, you can in theory use all logback features.

Be informed that we were not able to test all logback features in the context of observers. If you have a special problem with logback support, come back to our support team to see if we can help.

An interesting additional feature are the new pattern definition tokens "args" and "arg".

See here an example to select a dedicated property from an observation for the use in a log.

## Spring XML fragment

```

<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device"/>
  <property name="filter">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AcceptPredicate">
        <property name="expr" value="{observation.code}"/>
        <property name="pattern" value="signme\.signatureProcess\.created"/>
      </bean>
    </list>
  </property>
  <property name="view">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="static"/>
        <property name="value" value="some value"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="source"/>
        <property name="value" value="{observation.source}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="processId"/>
        <property name="value" value="{observation.signatureProcessId}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="clientInfo"/>
        <property name="value" value="{flow.variables.clientInfo}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="computer"/>
        <property name="value" value="{system.getenv.COMPUTERNAME}"/>
      </bean>
    </list>
  </property>
  <property name="appender">
    <bean class="ch.qos.logback.core.FileAppender">
      <property name="name" value="FILE"/>
      <property name="file" value="{cloudsuite.log.dir}/signme.log"/>
      <property name="append" value="false"/>
      <property name="encoder">
        <bean class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
          <property name="pattern" value="[%d{HH:mm:ss.SSS}] %msg %arg{source}
%arg{processId} %arg{clientInfo}%n"/>
        </bean>
      </property>
    </bean>
  </property>
</bean>

```

And the log file:

```

[11:25:16.332] {sign.signme.signatureProcess.created} device.signMe.default
5E3E5A17E3E1A3C17EA81A44765D955644A45DA94EA8543985034B6F5261C170 i am very important

```

## 7.6 More on webhooks

There are some best practices that come with a webhook and two of the most important are:

- Always use TLS
- Always authenticate

Well, the first one is builtin in Java if you have an appropriate SSL server certificate.

The second rule is supported by using an **authenticator** along with the webhook event.

#### Example Basic Authentication

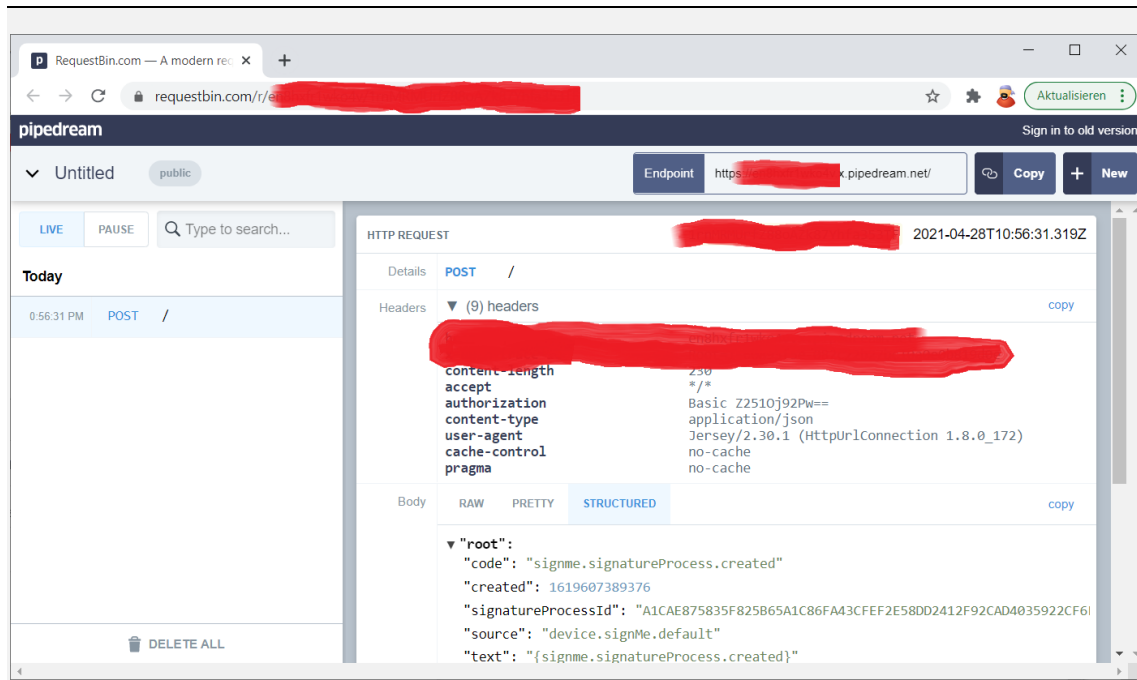
---

#### Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device"/>
  <property name="filter">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AcceptPredicate">
        <property name="expr" value="{observation.code}"/>
        <property name="pattern" value="signme.*"/>
      </bean>
    </list>
  </property>
  <property name="appender">
    <bean class="de.intarsys.tools.webhook.logback.WebhookAppender">
      <property name="webhook">
        <bean class="de.intarsys.tools.webhook.impl.StandardWebhookStub">
          <property name="method" value="POST"/>
          <property name="endpointUrl" value="https://...pipedream.net/" />
        </bean>
        <bean class="de.intarsys.tools.webhook.authenticator.BasicAuthAuthenticator">
          <property name="user" value="gnu"/>
          <property name="password" value="plain#Zm9v"/>
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

---

You can now see the additional authorization header



You can use an API token approach by leveraging `ApiTokenAuthenticator` instead

### Spring XML fragment

```

<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device"/>
  <property name="filter">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AcceptPredicate">
        <property name="expr" value="{observation.code}"/>
        <property name="pattern" value="sign.signme.*"/>
      </bean>
    </list>
  </property>
  <property name="appender">
    <bean class="de.intarsys.tools.webhook.logback.WebhookAppender">
      <property name="webhook">
        <bean class="de.intarsys.tools.webhook.impl.StandardWebhookStub">
          <property name="method" value="POST"/>
          <property name="endpointUrl" value="https://...pipedream.net/">
            <property name="authenticator">
              <bean
class="de.intarsys.tools.webhook.authenticator.ApiTokenAuthenticator">
                <property name="token" value="plain#d3Vyc3RzYWxhdA"/>
              </bean>
            </property>
          </bean>
        </property>
      </bean>
    </property>
  </bean>
</property>
</bean>

```

This will result in an authorization: Bearer <token> header.

## 7.7 Creating an audit log

An audit log can be created based on the observation implementation.

The basic observations used are the "sign.ok", "sign.failed" and "sign.canceled" observations from the device components.

Besides the basic observation properties of these events, you can leverage the wide range of string expansion namespaces to add relevant information.

Some useful properties are

- **created**  
A timestamp for the observation
- **source**  
The device that originated the observation
- **code**  
The code for the observation
- **Some token indicating the client application**  
This may be injected via the flow variables in free form
- **The user principal**  
This may be injected via flow variables or the principal flow option
- **The distinguished name for the signer certificate**  
This is available from the **observation.app**
- **Some evidence for the documents that have been signed**  
This is available from the **observation.app**, too

This definition will create a log file in JSON format that contains all of the above properties.

## Spring XML fragment

```

<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device"/>
  <property name="view">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="timestamp"/>
        <property name="value" value="{observation.created}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="source"/>
        <property name="value" value="{observation.source}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="code"/>
        <property name="value" value="{observation.code}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="app"/>
        <property name="value" value="{flow.variables.app}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="user"/>
        <property name="value" value="{principal.user.name}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="subject"/>
        <property name="value" value="{observation.app.subject.dn}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="sigEvd"/>
        <property name="value" value="{observation.app.signatureEvidence}"/>
      </bean>
    </list>
  </property>
  <property name="appender">
    <bean class="ch.qos.logback.core.FileAppender">
      <property name="name" value="FILE"/>
      <property name="file" value="{cloudsuite.log.dir}/audit.log"/>
      <property name="append" value="false"/>
      <property name="encoder">
        <bean
class="de.intarsys.tools.logging.logback.logstash.DefaultJsonEncoder">
          </bean>
        </property>
      </bean>
    </property>
  </bean>
</bean>

```

## Example output



```
{
  "source": "device.demo.default",
  "timestamp": 1621701811385,
  "code": "sign.ok",
  "app": "shrdlu",
  "user": "mit",
  "subject": "C=DE, O=intarsys GmbH, CN=cloud suite gears demo",
  "sigEvd": {
    "items": [{
      "label": "seiten1.pdf",
      "digest": {
        "algorithm": "SHA256",
        "bytes":
"R2ZaNTJPNzRDRER3KzE2V2EzTU0xODkySStmTmxwaDNkVjZ6a3MwSVZMQT0="
      }
    }
  ]
}
}{
  "source": "device.demo.tsa",
  "timestamp": 1621701819964,
  "code": "sign.ok",
  "app": "shrdlu",
  "user": "mit",
  "subject": "C=DE, O=intarsys GmbH, CN=cloud suite gears demo TSS",
  "sigEvd": null
}
```

Sending this output to an appropriate syslog device will give you additional integrity features.

## 7.8 Plain Logback Filtering

Let's assume, you want to have a smaller log with only information you are interested in. Logback filtering with the JaninoEventEvaluator [5] is an easy way to remove superfluous messages.

Copy the gears default logback configuration from cloudsuite-gears.ZIP\example\logging\config to \${cloudsuite.config.shared}.

Collect the names of the loggers to be filtered. Possibly, you have to enlarge the encoder pattern to get the full logger name (see c{80} in the example below).

### Logback XML fragment

```
<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
  ...
  <encoder>
    <charset>UTF-8</charset>
    <pattern>[%d{dd.MM.yyyy-HH:mm:ss.SSS}] [%.-1p] [%-80c{80}] [%-20.20t] [%-
10.10zone] [%X] %msg%n%rEx</pattern>
  </encoder>
</appender>
```

Add a filter to the existing file appender:

## Spring XML fragment

```
<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator class="ch.qos.logback.classic.boolex.JaninoEventEvaluator">
      <expression>
        (
          logger.startsWith(&quot;de.intarsys.security.standard.validation&quot;); ||
          logger.startsWith(&quot;de.intarsys.security.app.validation&quot;); ||
          logger.startsWith(&quot;de.intarsys.security.document.validation&quot;); ||
          logger.startsWith(&quot;de.intarsys.security.processor.validation&quot;);
        )
      </expression>
    </evaluator>
    <OnMatch>DENY</OnMatch>
    <OnMismatch>NEUTRAL</OnMismatch >
  </filter>
  ...
</appender>
```

To get the whole thing working you need to deploy the actual JANINO jars `janino-3.1.0.jar` and `commons-compiler-3.1.0.jar` from [6] to `<CATALINA_HOME>\lib`.

Result is a log without validation info.

JANINO has a lot of more expression to express every rule you need [5].

## 7.9 Graylog

Graylog is a powerful log management tool that allows you to collect, index, and analyze log data in real-time. This chapter will guide you through the setup process, including configuring Graylog support with a logback appender and deploying it using Docker Compose. For additional information please refer to the official Graylog documentation.

Graylog is supported via a logback appender. An example configuration can be found in the example folder under `/example/logging/graylog`. Add the `logback.xml` to your gears config folder or add the `GelfUdpAppender` from the example to an existing `logback.xml`.

## Logback Graylog UDP appender

```

<appender name="GELF" class="de.siegmar.logbackgelf.GelfUdpAppender">
  <graylogHost>localhost</graylogHost>
  <graylogPort>12201</graylogPort>
  <encoder class="de.siegmar.logbackgelf.GelfEncoder">
    <originHost>localhost</originHost>
    <includeLevelName>true</includeLevelName>
    <shortPatternLayout class="ch.qos.logback.classic.PatternLayout">
      <pattern>%m%nopex</pattern>
    </shortPatternLayout>
    <fullPatternLayout class="ch.qos.logback.classic.PatternLayout">
      <pattern>[%d{dd.MM.yyyy-HH:mm:ss.SSS}] [%.-1p] [%-20.20c{20}] [%-
20.20t] [%X{corr}] [%X{client}] %msg%n%rEx</pattern>
    </fullPatternLayout>
  </encoder>
</appender>

<appender name="ASYNC_GELF" class="ch.qos.logback.classic.AsyncAppender">
  <appender-ref ref="GELF" />
  <queueSize>500</queueSize>
  <discardingThreshold>0</discardingThreshold>
  <neverBlock>false</neverBlock>
</appender>

<!-- Add the GELF appender to the root logger -->
<root level="${log.level}">
  <appender-ref ref="ASYNCFILE" />
  <appender-ref ref="STDOUT" />
  <appender-ref ref="ASYNC_GELF" />
</root>

```

The example folder also includes a docker-compse.yml which can be used to get started easily using docker.

You can download Graylog from the official website or use Docker to pull the Graylog image.

To start Graylog using Docker Compose, navigate to the directory containing your docker-compose.yml file and run:

```
docker-compose up -d
```

First you should inspect the data node log and see something like the following:

```

Initial configuration is accessible at 0.0.0.0:9000, with username 'admin' and password
'gkcWeBuljW'.
Try clicking on http://admin:gkcWeBuljW@0.0.0.0:9000

```

Open the preflight ui at the link displayed in the log and follow the instructions.

Next, you can access the Graylog Web Interface. Open your browser and go to <http://localhost:9000>. Log in with the default credentials:

Username: admin

Password: admin

Configure Initial Settings: Navigate to System -> Inputs, select GELF UDP from the dropdown, and click “Launch new input”.

Now the gears log should be sent to the Graylog server and logs should be visible in the Graylog search section.

## 8. Security recipes

### 8.1 Token-based authorization on gears flow

#### 8.1.1 Application-Level Authentication with Self-Contained

##### JWS Tokens

This configuration focuses on system-level authentication using self-contained JSON Web Signature (JWS) tokens. It is designed for scenarios where the application acts on its own behalf, not representing a user, and uses JWS tokens for authentication. Upon successful authentication, it gains access to any user-related endpoints.

#### Spring XML fragment

```
<bean id="jwtDecoder" class="de.intarsys.spring.security.JwtDecoderFactoryBean">
  <property name="jwkSetUri"
value="https://keycloak.mydomain.de/auth/realms/gears/protocol/openid-connect/certs" />
</bean>

<bean id="jwtAuthenticationProvider"
class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticat
ionProvider">
  <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_USER" />
        </bean>
      </property>
    </bean>
  </property>
</bean>

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

<bean id="securityRealmFlowAuthenticationFilter"
class="org.springframework.security.oauth2.server.resource.web.BearerTokenAuthentication
Filter">
  <constructor-arg ref="securityRealmFlowAuthenticationManager" />
  <property name="bearerTokenResolver">
    <bean
class="org.springframework.security.oauth2.server.resource.web.DefaultBearerTokenResolve
r" />
  </property>
</bean>
```

```

    </property>
  </bean>

  <bean id="principalProviderUser"

class="de.intarsys.cloudsuite.gears.security.spring.SpringSecurityPrincipalProvider">
    <property name="role" value="urn:intarsys:names:principal:1.0:role:Client" />
    <property name="principalDao">
      <bean class="de.intarsys.cloudsuite.gears.security.spring.JwtPrincipalDao" />
    </property>
  </bean>

```

## 8.1.2 User-Level Authentication with Authorization Code Grant

This setup is tailored for user-level authentication using the OAuth 2.0 Authorization Code Grant flow. It is suitable for applications that need to perform actions on behalf of a user.

### Spring XML fragment

```

<bean id="jwtDecoder" class="de.intarsys.spring.security.JwtDecoderFactoryBean">
  <property name="jwkSetUri" value="https://
keycloak.mydomain.de/auth/realms/gears/protocol/openid-connect/certs" />
</bean>

<bean id="jwtAuthenticationProvider"
class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticat
ionProvider">
  <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 USER" />
        </bean>
      </property>
    </bean>
  </property>
</bean>

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

<bean id="securityRealmFlowAuthenticationFilter"
class="org.springframework.security.oauth2.server.resource.web.BearerTokenAuthentication
Filter">
  <constructor-arg ref="securityRealmFlowAuthenticationManager" />
  <property name="bearerTokenResolver">
    <bean
class="org.springframework.security.oauth2.server.resource.web.DefaultBearerTokenResolve
r" />
  </property>
</bean>

<bean id="principalProviderUser"

class="de.intarsys.cloudsuite.gears.security.spring.SpringSecurityPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="principalDao">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.JwtPrincipalDao" />
  </property>
</bean>

```

```
</property>
</bean>
```

### 8.1.3 User Authorization Verification

This configuration is aimed at verifying the authorization of users within the application. It involves checking the user's roles to determine if they are allowed to perform certain actions or access specific resources.

#### Spring XML fragment

```
<bean id="jwtDecoder" class="de.intarsys.spring.security.JwtDecoderFactoryBean">
  <property name="jwkSetUri" value="https://
keycloak.mydomain.de/auth/realms/gears/protocol/openid-connect/certs" />
</bean>

<bean id="jwtAuthenticationProvider"
class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticat
ionProvider">
  <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="realm_access.roles" />
          <property name="authorityPrefix" value="ROLE_" />
          <property name="authorityToUpperCase" value="true" />
        </bean>
      </property>
    </bean>
  </property>
</bean>
</property>
</bean>

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

<bean id="securityRealmFlowAuthenticationFilter"

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

<bean id="principalProviderUser"

class="de.intarsys.cloudsuite.gears.security.spring.SpringSecurityPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="principalDao">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.JwtPrincipalDao" />
  </property>
</bean>
```

```
</property>
</bean>
```

## 8.1.4 Enabling Introspection for Gears Core Authentication

This configuration adds token introspection capabilities to the previous configuration. It verifies the token by consulting an external authorization server e.g. Keycloak to confirm their validity.

### Spring XML fragment

```
<bean id="jwtDecoder" class="de.intarsys.spring.security.JwtDecoderFactoryBean">
  <property name="jwkSetUri" value="https://
keycloak.mydomain.de/auth/realms/gears/protocol/openid-connect/certs" />
</bean>

<bean id="jwtAuthenticationProvider"
class="org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticat
ionProvider">
  <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="realm_access.roles" />
          <property name="authorityPrefix" value="ROLE_" />
          <property name="authorityToUpperCase" value="true" />
        </bean>
      </property>
    </bean>
  </property>
</bean>

<bean id="introAuthenticationProvider"
class="org.springframework.security.oauth2.server.resource.authentication.OpaqueTokenAut
henticationProvider">
  <constructor-arg ref="keycloakTokenIntrospector" />
</bean>

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

<bean id="securityRealmFlowAuthenticationFilter"
class="org.springframework.security.oauth2.server.resource.web.BearerTokenAuthentication
Filter">
  <constructor-arg ref="securityRealmFlowAuthenticationManager" />
  <property name="bearerTokenResolver">
    <bean
class="org.springframework.security.oauth2.server.resource.web.DefaultBearerTokenResolve
r" />
  </property>
</bean>

<bean id="principalProviderUser"
class="de.intarsys.cloudsuite.gears.security.spring.SpringSecurityPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="principalDao">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.JwtPrincipalDao" />
  </property>
</bean>
```



```
</property>
</bean>

<bean id="keycloakTokenIntrospector"
class="org.springframework.security.oauth2.server.resource.introspection.SpringOpaqueTokenIntrospector">
  <constructor-arg index="0" type="java.lang.String"
value="KEYCLOACK_INTROSPECTION_ENDPOINT"/>
  <constructor-arg index="1" type="java.lang.String" value="KEYCLOACK_CLIENT_ID"/>
  <constructor-arg index="2" type="java.lang.String" value="KEYCLOACK_CLIENT_SECRET"/>
</bean>

</appender>
```

## 9. External References

---

[1] intarsys GmbH, Sign Live! Security Applications Developers Guide.

[2] intarsys GmbH, Sign Live! cloud suite gears manual.

[3] intarsys GmbH, Sign Live! cloud suite gears incubator.

[4] intarsys GmbH, Sign Live! cloud suite gears cookbook.