

# Sign Live! cloud suite gears manual

Oktober 2025

intarsys GmbH

Sign Live! cloud suite gears manual

Version 8.14

cloud suite gears architecture, design & reference

intarsys GmbH  
Sign Live! cloud suite gears manual  
Version 8.14

All rights reserved  
© 2020 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 provides both an overview of the product design and architecture and a reference for using the components and services.

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

## ■ Reviews and comments

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

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

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

## ■ Disclaimer

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

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

# Contents

---

Preface	5
▪ Author and company	5
▪ Trademarks	5
▪ Who should read this book	5
▪ Reviews and comments	5
▪ Disclaimer	6
Contents	7
Introduction	19
1. Overview	20
2. Building blocks	21
2.1 Services	21
2.1.1 Meta	21
2.1.2 Signer	21
2.1.3 Viewer	21
2.1.4 Explorer	22
2.1.5 Timestamper	22
2.2 Modules	22
2.3 Operator console	22
2.4 Admin console	22
2.5 Demo client	22
3. Architecture	23
3.1 Overview	23
3.1.1 Communication channels	23
4. Installation	26
4.1 Overview	26
4.2 System requirements	26
4.2.1 Server	26

## Content

4.2.2	Client	28
4.2.3	3 <sup>rd</sup> party	28
4.3	Documentation	29
4.4	Installation Process	29
4.4.1	Server	29
4.4.2	Client	29
4.5	Web Apps	29
4.5.1	Web App "core"	29
4.5.2	Web App "demo"	29
4.6	SDK	29
4.7	"demo"	30
4.8	Miscellaneous	30
4.8.1	CWT Rendering library	30
4.9	Version check web app	30
4.9.1	In the browser log	30
4.10	Version check server	30
4.10.1	In the WEB-INF directory	31
4.10.2	In the log	31
4.10.3	On the client	31
5.	Design details	32
5.1	Flow	32
5.2	Conversation	32
5.2.1	Reply stage	33
5.3	Document	33
5.3.1	Document Properties	34
5.3.2	Document attachments	34
5.3.3	Tags	35
5.3.4	Document specific arguments	36
5.4	Repository	38
5.5	Services	39
6.	Web UIs	41
6.1	Overview	41
6.2	Viewer	41
6.3	Explorer	42
6.4	Control Panel	43
6.4.1	Landing Page	44
6.4.2	Authentication	44
6.4.3	Control Panel Home	45
6.4.4	Device Management	45
6.4.5	License Management	53
6.4.6	Password Tool	54
6.5	Browser security	55
6.5.1	Headers	55
6.5.2	Header configuration	56
7.	Demo application	58

7.1 Overview	58
8. Crypto components	59
8.1 Basics	59
8.1.1 IByteProvider & IByteStreamProvider	59
8.1.2 IKeyDerivationFunction	59
8.1.3 ICipherFactory	59
8.1.4 Key tree	61
8.2 Cryptdec	61
8.2.1 Plain	61
8.3 Secret	62
8.3.1 Syntax	62
8.3.2 Configuration	62
8.4 Repository encryption	63
9. Configuration	64
9.1 Overview	64
9.1.1 Configuration location	64
9.1.2 Built-in configuration	65
9.1.3 Custom property definition	66
9.1.4 Custom bean definition	66
9.1.5 Using profiles	66
9.1.6 String expansion integration	67
9.2 Web application root	67
9.3 Logging	68
9.3.1 Override logging	68
9.3.2 Built-in logging	68
9.3.3 Log correlation	69
9.3.4 Log tweaks	70
9.4 Licenses	70
9.5 Data source	72
9.6 Conversation registry	72
9.7 Basic security	73
9.7.1 Master key	73
9.7.2 Master KDF	73
9.7.3 Application "cryptdec"	73
9.8 Repository	74
9.8.1 Basic settings	74
9.8.2 Encryption	74
9.9 PDF security environment	76
9.10 Principals	77
9.10.1 Overview	77
9.10.2 Principal model	77
9.10.3 Roles	80
9.10.4 Provider	80
9.10.5 Default configuration	82
9.11 Services	84

## Content

9.11.1	Signer	84
9.11.2	Viewer	84
9.11.3	Explorer	86
9.11.4	Timestamper	86
9.12	Signature environment	86
9.12.1	Timestamp creation	87
9.12.2	Signature validation	87
9.12.3	Trusted list management	87
9.13	Load Balancing	88
9.13.1	Additional gears configuration	88
9.13.2	Load balancer configuration	88
9.13.3	Bridge integration	89
9.14	String expansion	89
10.	UI definition	91
10.1	Overview	91
10.2	Widget definition	91
10.2.1	Structure	91
10.2.2	Icon	94
10.2.3	Button styling	95
10.2.4	Behavior	96
10.3	Action definition	97
10.3.1	Plain action	97
10.3.2	Complex action	98
10.3.3	Execution semantics	100
10.3.4	Simple expressions	102
10.4	String expansion	103
10.5	Overlay widgets	105
10.5.1	Annotations overlay	105
10.5.2	Fine rendering overlay	123
11.	Authentication	124
11.1	Overview	124
11.2	Opt-out	124
11.3	Concepts	124
11.4	Security realms	125
11.4.1	Overview	126
11.4.2	Common properties	126
11.4.3	Security realm "Flow"	126
11.4.4	Security realm "Control"	126
11.4.5	Security realm "Manage"	126
11.5	Logout	126
11.6	Authentication filter	127
11.6.1	Static authentication	127
11.6.2	Basic authentication	128
11.6.3	OAuth2 authentication	128
11.6.4	X.509 authentication	128

11.7 Authentication manager	128
11.7.1 In-memory repository	129
11.7.2 JDBC repository	130
11.8 Principal integration	130
11.8.1 Dao based principal lookup	130
11.8.2 JWT based principal lookup	131
11.9 Well known security beans	131
12. Authorization	133
12.1 Overview	133
12.2 Concepts	133
12.2.1 Authentication	133
12.2.2 Authority	133
12.2.3 Resource	134
12.2.4 Operation	134
12.2.5 Authorization strategy	134
12.3 Integration (observation)	134
12.4 ACL Authorization	135
12.4.1 Strategy	135
12.4.2 Configuration	136
12.4.3 Resources	137
13. Services	142
13.1 Overview	142
13.2 API	142
13.3 Protocol	142
13.3.1 Flow creation	142
13.3.2 Conversational response	142
13.3.3 "Final" stages	143
13.3.4 Multipage in-band	145
13.3.5 Redirect URI	146
13.3.6 Multipage out-of-band	147
13.3.7 Summary	147
13.4 Client helper	148
13.4.1 JavaScript Client	148
13.4.2 Java Client	148
13.5 Serialization issues	149
13.5.1 Scaling in a Java environment	149
13.5.2 Confidentiality	150
13.5.3 Property order	150
13.6 Error handling	151
13.6.1 Overview	151
13.6.2 ErrorDetail object	151
13.6.3 ResponseError object	151
13.6.4 Synchronous error handling	151
13.6.5 Conversational error handling	152
13.7 Request options	152

## Content

13.7.1	Redirect URI	152
13.7.2	Restricted identification	153
13.7.3	Principal	153
13.7.4	Language	154
14.	Devices	155
14.1	Overview	156
14.1.1	Device provider	156
14.1.2	Device	156
14.1.3	Application	157
14.1.4	Application "signer"	157
14.1.5	Policy	158
14.1.6	Policy "signer"	159
14.1.7	Monitoring	159
14.2	Demo Device	162
14.2.1	Overview	162
14.2.2	Device Configuration	163
14.2.3	Usage	163
14.2.4	Signer Configuration	164
14.2.5	Monitoring	165
14.2.6	Observations	165
14.3	Bridge Device	166
14.3.1	Overview	166
14.3.2	gears configuration	166
14.3.3	Device Configuration	166
14.3.4	Usage	168
14.3.5	Signer Configuration	169
14.3.6	Monitoring	170
14.3.7	Observations	170
14.3.8	Testing	170
14.4	Smartcard Device	172
14.4.1	Overview	172
14.4.2	Device Configuration	172
14.4.3	Usage	172
14.4.4	Monitoring	172
14.4.5	Observations	173
14.5	Pool Device	174
14.5.1	Overview	174
14.5.2	Device authentication	174
14.5.3	Device Configuration	174
14.5.4	Usage	176
14.5.5	Signer Configuration	177
14.5.6	Monitoring	177
14.5.7	Observations	179
14.6	Swisscom All-in Signing Service Device	179
14.6.1	Overview	179



14.6.2	Device Configuration	179
14.6.3	Signer Configuration	180
14.6.4	Timestampers Configuration	183
14.6.5	Usage	184
14.6.6	Monitoring	191
14.6.7	Observations	191
14.7	Bundesdruckerei sign-me Device	192
14.7.1	Overview	192
14.7.2	Authorization flow	192
14.7.3	Device Configuration	193
14.7.4	Usage	199
14.7.5	Signer Configuration	201
14.7.6	Monitoring	204
14.7.7	Observations	205
14.8	TSA Device	208
14.8.1	Overview	208
14.8.2	Device Configuration	208
14.8.3	Signer Usage	209
14.8.4	Signer Configuration integration	210
14.8.5	Timestampers Usage	210
14.8.6	Timestampers Configuration	210
14.8.7	Monitoring	211
14.8.8	Observations	211
14.9	UPReg Device	212
14.9.1	Overview	212
14.9.2	Device Configuration	212
14.9.3	Usage	213
14.9.4	Signer Configuration	214
14.9.5	Monitoring	216
14.9.6	Observations	216
14.10	Azure Device	217
14.10.1	Overview	217
14.10.2	Device Configuration	217
14.10.3	Usage	219
14.10.4	Signer Configuration	220
14.10.5	Monitoring	221
14.10.6	Observations	221
14.11	proNEXT Device	222
14.11.1	Overview	222
14.11.2	Device Configuration	222
14.11.3	Usage	224
14.11.4	Signer Configuration	225
14.11.5	Monitoring	226
14.11.6	Observations	226
14.12	CSC device	229
14.12.1	Overview	229

## Content

14.12.2	Device configuration	229
14.12.3	Usage	231
14.12.4	Monitoring	232
14.12.5	Observations	232
14.13	A-Trust device	233
14.13.1	Overview	233
14.13.2	Device configuration	233
14.13.3	Usage	233
14.13.4	Signer configuration	234
14.13.5	Monitoring	236
14.13.6	Observations	236
14.14	gears Device	237
14.14.1	Overview	237
14.14.2	Device Configuration	237
14.14.3	Usage	238
14.14.4	Signer Configuration	239
14.14.5	Monitoring	240
14.14.6	Observations	240
14.15	PKCS #11 device	241
14.15.1	Overview	241
14.15.2	Device configuration	241
14.15.3	Usage	242
14.15.4	Signer configuration	242
14.15.5	Monitoring	243
14.15.6	Observations	243
15.	Integration	244
15.1	Overview	244
15.2	Model	244
15.2.1	Observation	244
15.2.2	Observer	245
15.2.3	Filter	245
15.2.4	View	246
15.3	Implementation	246
15.4	Observer definition	246
15.5	Filter definition	247
15.5.1	Accept	248
15.5.2	Reject	248
15.6	View definition	248
15.6.1	Include	249
15.6.2	Exclude	249
15.6.3	Add property	249
15.6.4	Complete observer example	250
15.7	Appender definition	251
15.7.1	Plain logging	251
15.7.2	Pattern conversion for args	252

15.7.3	Other appenders	253
15.7.4	Logstash support	253
15.7.5	Webhook	254
15.8	Example	254
15.9	Observations	255
15.9.1	Overview	255
15.9.2	application	255
15.9.3	license	255
15.10	Webhook	256
15.10.1	Overview	256
15.10.2	Webhook stub	256
15.10.3	Execution semantics	257
15.10.4	SSL/TLS	258
15.10.5	Authentication	259
16.	Monitoring	261
16.1	Overview	261
16.2	JMX	261
16.2.1	Installation	261
16.2.2	Configuration	261
16.2.3	Client	261
16.2.4	MBean Deployment	263
16.2.5	gears MBeans	263
16.3	Spring actuator	264
16.3.1	Installation	265
16.3.2	Configuration	265
16.3.3	Endpoints	265
16.3.4	gears Health objects	267
16.4	Micrometer, Prometheus and Grafana	273
16.4.1	Components	273
16.4.2	Metric Flow Example	274
16.4.3	Metrics	274
16.4.4	Configuration	276
17.	NLS support	280
17.1	Language selection	280
17.2	Language option	280
17.3	Language static expansion	280
17.4	Language dynamic expansion	280
17.5	Additional resource path	281
17.6	String expansion within NLS files	281
17.6.1	Eliminate [] brackets	282
17.6.2	Spring string expansion in NLS files	282
18.	Reference	283
18.1	Common data models	283
18.1.1	Configuration	283

## Content

18.1.2	PluginSpec	289
18.1.3	ActionSpec	290
18.1.4	WidgetSpec	294
18.2	Principal related data models	298
18.2.1	GenericClaim	298
18.2.2	GenericPrincipal	299
18.2.3	GenericUser	300
18.2.4	JdbcPrincipalDao	301
18.2.5	JdbcUserDao	301
18.2.6	PojoPrincipalDao	302
18.2.7	PojoUserDao	303
18.2.8	ExplicitPrincipalProvider	304
18.2.9	StaticPrincipalProvider	304
18.2.10	SpringSecurityPrincipalProvider	305
18.3	Crypto related data models	305
18.3.1	ISslContextProvider	305
18.3.2	IByteProvider	306
18.3.3	ICipherFactory	308
18.3.4	IKeyDerivationFunction	309
18.4	Plugin reference	310
18.5	Action reference	310
18.5.1	Basic actions	310
18.5.2	"ui" support	314
18.5.3	"app" actions	318
18.5.4	"flow" actions	319
18.6	Widget reference	331
18.6.1	Overview	332
18.6.2	Group	332
18.6.3	Button	332
18.6.4	Toggle button	332
18.6.5	Separator	333
18.6.6	Label	333
18.6.7	Spacer	333
18.6.8	de.intarsys.ui.control.PageSelector	333
18.6.9	Toolbar	333
18.6.10	ControlOverlay	334
18.6.11	AnnotationsOverlay	334
18.6.12	SidebarContainer	334
18.6.13	ThumbnailsSidebar	335
18.6.14	PropertiesSidebar	335
18.6.15	SignaturesSidebar	335
18.7	Well known widgets	335
19.	Implementation hints	340
19.1	Chunked transfer	340
20.	String expansion	341

20.1 Disclaimer	341
20.2 Basics	341
20.2.1 Why string expansion	341
20.2.2 Terminology	341
20.2.3 Syntax	342
20.2.4 Constant text in expression	342
20.2.5 Namespaces	342
20.2.6 Spring integration	343
20.3 Namespaces	344
20.3.1 Overview	344
20.3.2 app	344
20.3.3 config	345
20.3.4 counters	345
20.3.5 digestSigner	346
20.3.6 entity	347
20.3.7 environment	348
20.3.8 flow	348
20.3.9 identifiers	349
20.3.10 nlsmg	350
20.3.11 properties	350
20.3.12 system	351
20.3.13 time	351
20.4 Formatting	352
20.4.1 Overview	352
20.4.2 String formatting	352
20.4.3 Integer formatting	353
20.4.4 Date formatting	355
20.4.5 Float formatting	356
20.4.6 File path formatting	357
20.4.7 Default value	357
20.4.8 Recursion	358
20.4.9 Conditional evaluation	359
20.5 Tutorial	359
20.5.1 Example 1: Hardcoded signature label	360
20.5.2 Example 2: Signature label via Spring string expansion	360
20.5.3 Example 3: Signature label via gears string expansion	361
20.5.4 Example 4: Signature label via global variable	361
20.5.5 Example 5: Signature label via signer request	362
20.5.6 Example 6: Signature label via NLS message	363
21. Appendices	365
21.1 Cheat sheet	365
21.1.1 Windows locations	365
21.1.2 Linux locations	365
21.1.3 Property definitions	365
21.1.4 Bean definitions	365

## Content

21.1.5	Logging	366
21.1.6	Licenses	366
21.1.7	Documentation	366
21.2	Error stage codes	366
21.3	Available icons	368
21.4	Proxies	368
21.4.1	Incoming	368
21.4.2	Outgoing	369
21.5	Spring well known beans	369
21.6	Principal roles	370
21.7	Reply stage schemes	370
21.8	EncryptionInfo schema	371
22.	External References	372

# Introduction

---

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

This book gives an overview of the architecture and design decisions for Sign Live! cloud suite gears, along with a detailed reference of how to configure and use the components and services.

# 1. Overview

---

Sign Live! cloud suite gears provides workflow enabled components, ready to be used easily in web based (and for some part, fat client based) products.

It provides a variety of features for simplifying the task of signing and validating transactions and documents that are simply plugged into your existing application.

The built-in protocols and algorithms provide best of breed support for all modern documents, protocols and standards.



## 2. Building blocks

### 2.1 Services

#### 2.1.1 Meta

Here you can collect meta information on the gears backend services.

#### 2.1.2 Signer

The **signer** service provides signatures for a collection of documents.

It is a one-stop solution for

- Different document types
  - PDF
  - XML
  - Images
  - Generic file content
- Different signature creation devices
  - Soft certificates
  - Remote signing backends
    - AIS
    - sign-me
  - Smartcards
  - HSM
  - Tokens
- Different signature formats
  - PDF intern (PAdES)
  - CMS (CAdES)
  - XML (XAdES)
  - ASiC
- Different signature "decoration"
  - Visible fields & their appearance
  - Timestamping
  - Long term validation
- Different authentication levels
  - Advanced
  - Qualified

You hand a collection of documents, along with a description of the signature process to apply. The service will coordinate all necessary participants, gather consent and reply with the signature containers created.

#### 2.1.3 Viewer

The **viewer** allows for previewing and sometimes interactive manipulation of the documents to be signed.

It is implemented as a HTML5 web application and should be usable on any device that is able to render HTML5.

The viewer is customizable to improve integration in your workflows.

### 2.1.4 Explorer

The **explorer** allows to handle a batch of documents interactively before forwarding them to other services (like the **signer**).

The explorer is customizable to improve integration in your workflows.

### 2.1.5 Timestamper

The timestamper service provides document timestamps for a collection of PDF documents. You hand a collection of documents, the service will coordinate all necessary participants and reply with the signature containers created.

## 2.2 Modules

Optional additional components for the above services are encapsulated in modules that can be attached on a per site basis.

Examples of such modules are

- AIS signature creation device (a Swisscom service)
- sign-me signature creation device (a D-Trust service)
- HSM signature creation device
- Smartcard signature creation device

## 2.3 Operator console

The operator console allows direct access to the gears application server. It provides access to runtime state and allows manipulation of certain state.

An example for this is the presentation of pool state and the ability to stop, suspend, resume and start the pool.

## 2.4 Admin console

An administrative console to manage and control information that is used in the gears runtime.

The administration console is external to the gears application, the two are coupled via external storage, e.g. a database.

You can for example configure tenant, client application and user specific settings. Use of the console is highly dependent on your individual gears installation.

## 2.5 Demo client

An important source of documentation is the complete "demo client" application that is included in the SDK.

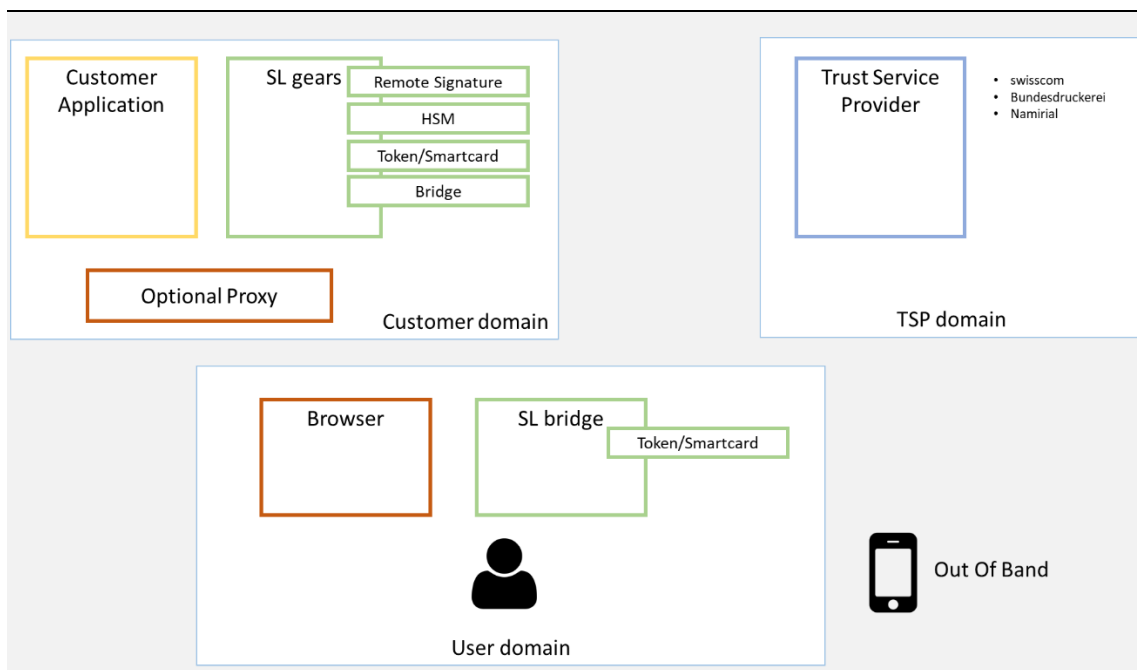
The demo client is an Angular HTML5 application, relying on JAX-RS based web services.

The demo is included both in binary and source form.

## 3. Architecture

### 3.1 Overview

This is the high-level overview of components involved in a typical gears architecture.



The green components are part of the Sign Live! cloud suite gears product family. The red ones are standard products (like the browser), the blue are third party components, optionally involved in the overall architecture (like the trust service provide TSP shown top right).

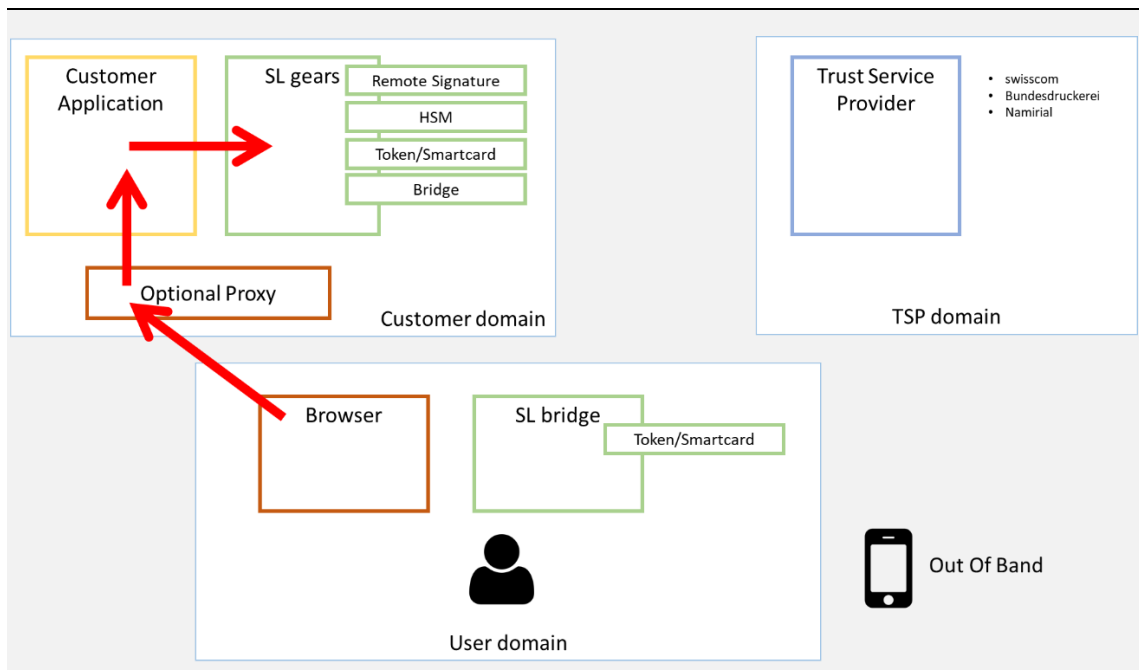
The yellow ones are the components you are developing yourself.

#### 3.1.1 Communication channels

##### 3.1.1.1 Synchronous signature

The simplest communication happens when creating a synchronous signature.

The user requests a signature in your application, your application sends a signature request and the signature result is returned (e.g. when using the unauthenticated demo device).

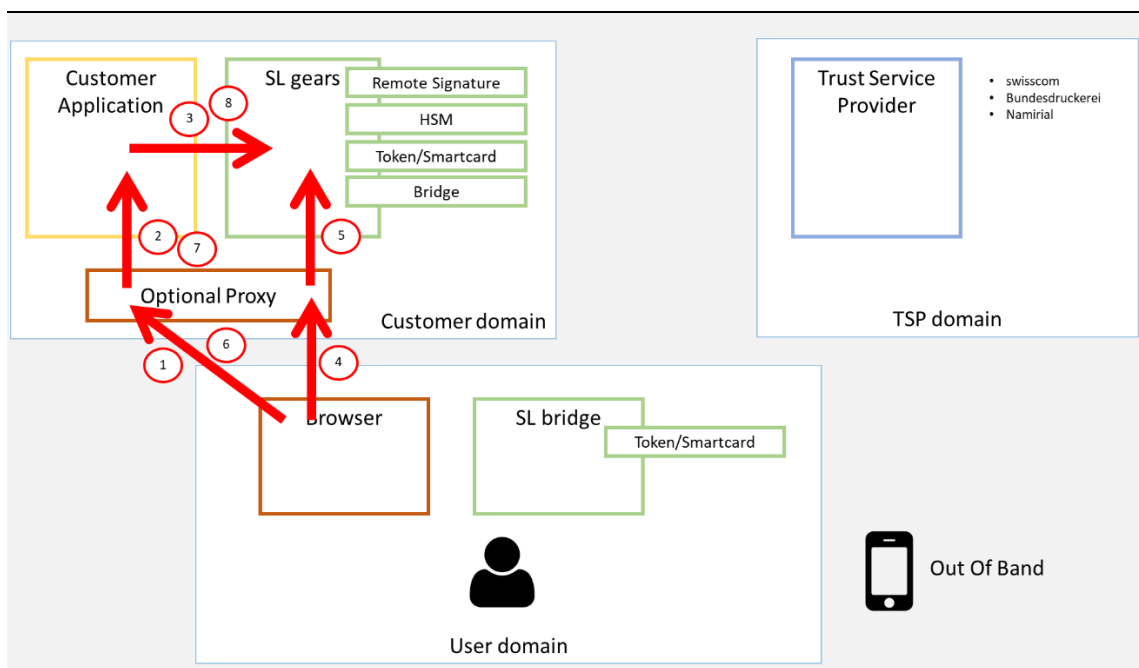


### 3.1.1.2 Remote signature

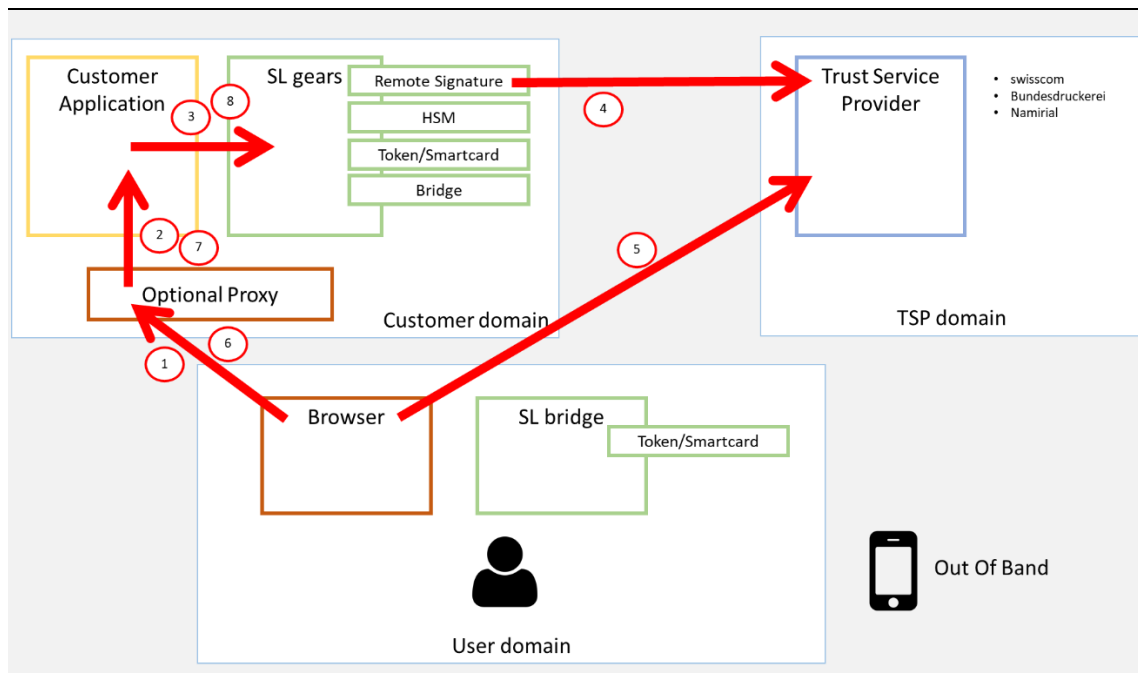
The remote signature typically requires some consent and as such is asynchronous.

After requesting the signature, your application is required to redirect to a location defined by gears.

The location may be a gears internal page (in this case the gears endpoint must be accessible from the user domain)



or a TSP page (the customer domain **and** user domain must be allowed to access the TSP domain).



Depending on the devices involved in the communication there are a number of details that have to be taken into account, like

- IP restrictions
- Client SSL

These requirements are stated with the respective device documentation chapters.

## 4. Installation

### 4.1 Overview

The product is shipped in a ZIP or TAR file, containing

- demo
- doc
- SDK
- webapps

### 4.2 System requirements

#### 4.2.1 Server

##### 4.2.1.1 Operating system

The server components are tested and released for

- Ubuntu Linux 24.04.3 LTS
- Red Hat Enterprise Linux 9.1
- Microsoft Windows 2016 / 2019 / 2022 / 2025 server

##### 4.2.1.2 Microsoft Windows requirements

To support rendering, native code bindings are required. This native code is linked dynamically to

- Visual Studio 2015 - 2022 runtime libraries

Ensure you have these libraries installed when using the viewer services.

More information on this topic is available in 4.8.1, Rendering library.

##### 4.2.1.3 Red Hat Enterprise Linux support

The server-side use of smartcards and PKCS#11 middleware is currently not supported.

##### 4.2.1.4 Containerization

The application can be operated by a container runtime (e.g. Docker) after proper container image creation. The creation of a gears container image builds on a base image which provides the required Java runtime environment and a servlet container.

Currently supported base images comprise:

- intarsys/tomcat:10-jdk17-zulu
- intarsys/tomcat:10.1.13-jdk17-rhel\_ubi9\_openjdk

Please see the cookbook [1] for recipes on obtaining or creating these base images and deriving a gears container image for operation.

#### 4.2.1.5 System software

##### 4.2.1.5.1 Java server runtime

The server components are developed in Java and require

- Java Runtime Environment 17 (64 bit)

The software is tested and released using Azul Zulu OpenJDK 17.60.17

##### 4.2.1.5.2 Servlet container Tomcat

The server components are deployed as "war" files. The server components are tested and released using

- Tomcat 10.1.44 (64 bit)

For server side rendering the Java VM is switched to headless mode automatically (-Djava.awt.headless=true).

##### 4.2.1.5.3 Reverse proxy

In complex installations you should think about installing Sign Live! cloud suite gears behind a reverse proxy (like nginx).

This may ease maintenance when it comes to multi homed servers, load balancing and SSL/TLS termination.

If using load balancing, the current version **must** use sticky connections for a client IP. The system uses special "conversations" that are not reflected in HTTP sessions. If you want to use a load balancing technique, please come back to our staff for discussing the possibilities.

#### 4.2.1.6 Hardware

##### 4.2.1.6.1 CPU

The system is tested on

- x86-64

architecture only.

For normal operation we recommend at least

- medium sized recent Core i5 or equivalent

Depending on the number of users and system setup (e.g. encrypted file repository, using server-side rendering) you may want to increase server power.

##### 4.2.1.6.2 Memory

Memory requirements depend heavily on the number of concurrent requests. We recommend at least

- 4 GB RAM

Depending on the number of users and system setup (e.g. encrypted file repository, using server-side rendering) you may want to increase available memory.

##### 4.2.1.6.3 Disk storage

Beside the software installation requirements

- ~ 1GB (Tomcat, Java, Sign Live! cloud suite gears components)

You need space for the server-side repository if documents are stored temporarily on the server. This depends on number of users, frequency and duration of requests.

- ~ 10GB

should be enough in small to medium sized installations.

#### 4.2.1.6.4 Peripherals

The system components do not require special peripherals.

You may have options (like HSM) that require additional hardware.

## 4.2.2 Client

### 4.2.2.1 Service clients

There are no special constraints on the service clients. They must provide plain JAX-RS payload (JSON over HTTP).

### 4.2.2.2 Web browser

Some components require a browser frontend. This is implemented using HTML5 and modern JavaScript features. The following browsers are tested and supported.

- Chrome > 139 (Windows 10/11, Android tablet)
- Firefox > 142 (Windows 10/11)
- Edge Chromium > 140 (Windows 10)
- Safari > 18 (MacOS, iPad)

### 4.2.2.3 Bridge

Some components access resources on the client machine. To achieve this, a local software component (bridge) has to be installed.

To access smartcards, all required drivers must be preinstalled. For the supported client environments see the software requirements defined in the "cloud suite bridge" manuals.

## 4.2.3 3<sup>rd</sup> party

Depending on the system setup there may be additional requirements to be met.

### 4.2.3.1 Database

Some system setups need a database (audit logging, profiles, ...).

The system comes with a bundled internal database (H2). This is not recommended for production.

You should provide a database installation in this case. The database can be installed on any server and is accessed using JDBC.

Our software components need DDL privileges on the associated database scheme, as by default the database is created and maintained by internal scripts.

The system is tested on

- H2
- PostgreSQL 12.15
- MariaDB 10.5

### 4.2.3.2 HSM

You can use an HSM (hardware security module) to create signatures on site.

Depending on the hardware supplier you must provide special network setup for accessing the HSM.

### 4.2.3.3 Remote signatures

You can use remote signature providers for creating signatures.



Beside the local product requirements, these providers may need special setup for the networking environment (proxies and firewall).

You must look up these individual requirements in the respective provider/module documentation.

General discussion for proxy handling can be found in the appendices.

## 4.3 Documentation

The documentation is contained in the "doc" folder of the deployment.

Additionally, the service API is documented using the swagger (OpenAPI) specification language. The documentation endpoint is

**`http://<host>/<gears context>/apidoc/index.html`**

## 4.4 Installation Process

### 4.4.1 Server

Perform the following installation steps:

1. Install Java Runtime
2. Install Tomcat
3. Deploy the "cloudsuite-gears#core.war" into the Tomcat webapps directory. For further details read chapter 4.5.
4. Open the following URL in the browser  
**`http://<host>:8080/cloudsuite-gears/core`**

### 4.4.2 Client

For the installation of the client component bridge see the corresponding documentation.

## 4.5 Web Apps

### 4.5.1 Web App "core"

You can install the core gears services by simply deploying the "cloudsuite-gears#core.war" to a standard servlet container.

The servlet container used for testing is Tomcat 9.0.x.

This will deploy the default product installation which can be used directly in combination with the "demo" from the webapps folder.

If you need further refinements, see the section "Configuration" of this manual.

The core web application needs write access to the `${cloudsuite.data.shared}` directory. This means that for a Linux installation, you must create this directory and grant access to the user running the servlet container.

### 4.5.2 Web App "demo"

You can install a demo web application that interfaces to the core gears services by simply deploying the "cloudsuite-gears#demo.war" to a standard servlet container.

## 4.6 SDK

The SDK folder contains sources and JAR files that can ease the client implementation when using the Java language.

It contains API stubs that can be directly used for development.

The resources in this directory are **not** required to write a fully functional client. They are included to ease Java based client development.

## 4.7 "demo"

This directory contains the complete sources for the "demo" web application.

This application has a backend, written based on JAX-RS in Java and an Angular frontend.

## 4.8 Miscellaneous

### 4.8.1 CWT Rendering library

The CWT rendering backend used for annotations uses the "freetype" native code to handle font programs. When rendering fails, the reason may be a missing or otherwise malformed "freetype" library.

First, we try to access a "freetype/.dll/.so/dylib" using the current process path lookup. On most Unix systems this will succeed as freetype is a de facto standard there.

If this fails, we try to extract a matching binary from the resources we deploy with the Sign Live! cloud suite gears platform code.

This can fail now for a variety of reasons, the most obvious being:

- you have an exotic platform and we have no binary packed for this one  
→ You must switch to a supported backend platform
- we try to make a dynamic deployment, but your virus scanner believes this is an attack and prevents loading  
→ You must adapt your virus scanner setup or you provide a static installation of freetype on the system binary path.
- you have a Windows platform and the required Visual Studio 2015 - 2022 runtimes for dynamic linking are not installed, thus the provided freetype.dll cannot resolve its dependencies  
→ Ensure the Visual Studio 2015 - 2022 runtime library is installed. You can do this simply by installing Sign Live CC! or you can download the Visual C++ Redistributable Package directly from Microsoft.

## 4.9 Version check web app

### 4.9.1 In the browser log

All Sign Live! cloud suite gears web apps will print a version signature in the browser log upon start up:

---

```
cloud suite gears demo v.8.0.0, Build 21, 2018-04-23T16:12:13.461Z
```

---

## 4.10 Version check server

### 4.10.1 In the WEB-INF directory

In cases you need to physically check the installation version on the deployed server application, you can look up the "<webapp>/WEB-INF/version.txt". It contains tags that describe the artifact version

```
version=8.0.0
build=321
timestamp=Mon Apr 23 14:15:54 CEST 2018
```

### 4.10.2 In the log

The log file is the most important source of information for troubleshooting an installation. You can find the version of every component contained in the deployment near the beginning of the log file

```
[23.04.2018-14:19:02.442][I][d.i.spring.tools      ][localhost-startStop-1][ version info:
[23.04.2018-14:19:02.442][I][d.i.spring.tools      ][localhost-startStop-1][ intarsys-
cloudsuite-gears-core-backend (intarsys-cloudsuite-gears-core-backend-8.0.0.jar), 8.0.0,
local, Mon Apr 23 14:15:54 CEST 2018
[23.04.2018-14:19:02.541][I][d.i.spring.tools      ][localhost-startStop-1][ +- ASM (asm-
5.0.4.jar), 5.0.4
[23.04.2018-14:19:02.546][I][d.i.spring.tools      ][localhost-startStop-1][ +- Apache
Commons Codec (commons-codec-1.10.jar), 1.10, trunk@r1637108; 2014-11-06 14:14:12+0000
[23.04.2018-14:19:02.547][I][d.i.spring.tools      ][localhost-startStop-1][ +- Apache
Commons DBCP (commons-dbcp2-2.1.1.jar), 2.1.1, tags/DBCP_2_1_1_RC1@r1693845; 2015-08-03
00:33:18+0000
```

### 4.10.3 On the client

To check correct installation or version check your installation, you can simply request

```
<proto>://<host:port>/cloudsuite-gears/core/api/v1/meta/version
```

or

```
<proto>://<host:port>/cloudsuite-gears/demo/api/v1/meta/version
```

This will return version information on the currently installed services.



## 5. Design details

### 5.1 Flow

A "flow" is our name for the interaction or "workflow component" or "use case" that is initiated by the client via the service calls.

Such a **flow** is for example a "signature creation interaction". It may be as simple as a synchronous call to a service, returning the result immediately or as complex as a redirection between multiple parties involved for preview and authentication.

Anyway, the client only sees a standardized API of modest complexity - the server manages the flow to a defined outcome - either success, cancellation or failure.

Flows are created directly by the client, using the respective "create" methods of the available services.

A flow is represented using a **conversation** internally.

### 5.2 Conversation

A **conversation** is the protocol representation of a **flow**. It represents state in a distributed workflow where a single step can have asynchronous, interactive parts.

If, for example, the client requires a signature, it sends a "signer/create" request to the Sign Live! cloud suite gears server. In a classic scenario, the server would have received the request along with the required credentials, signed it and returned the PKCS1 or CMS data structure.

Nowadays, it is quite common to make backend components more complex, partly to provide more "plug and play" features to a client, partly because regulatory requirements force logic to be situated in the backend.

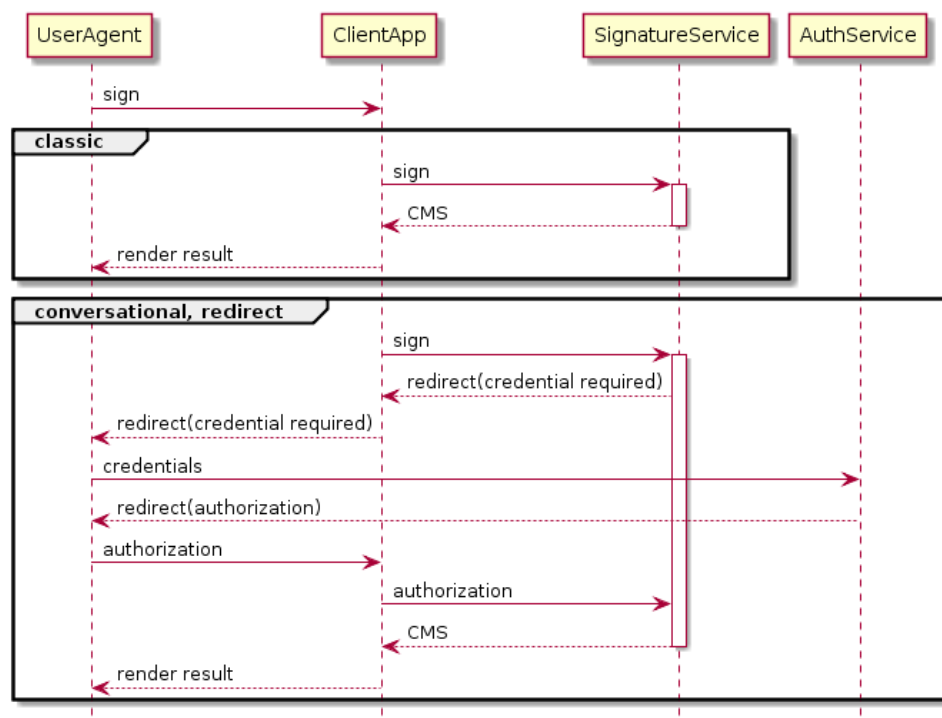
So, the server still receives the request and the hash, but now decides on its own if and how to authenticate and authorize the user. Most important, he may send a request to enter a PIN, OTP or any other kind of credential. This is a "sub interaction" to the client initiated one. The client has to take the role of a user agent, switch his application context and request the credentials from the user. The implementation and scope of this interaction may even be transparent to the client in form of a web redirect to a backend provided page.

This diagram shows these two interactions on a quite high level. One can see that the interaction is considerably more complex. Given the fact, that "redirect" is only one pattern that can arise in such a conversation (and even this is not necessarily standardized), there is a good piece of work to do.

Let's summarize some of the more common patterns that can be managed using the "conversation" paradigm:

- redirect

- OAuth2
- proprietary
- credential prompts
  - requesting PIN, OTP, mTAN and the like
  - protocols
    - HTTP authorization headers
    - proprietary
- polling
  - availability of result is signaled by a flag on the backend



The conversation implementation is built based on a generic protocol that supports plugging new interaction patterns. For each pattern we need client and server components that know how to handle the respective states in the conversation. The components drive the interaction to the next state or a final result.

Sign Live! cloud suite gears has component implementations for both client and server to handle the internals of a conversation.

A typical client will only have to include the "csconversation-<nn>.js" library and provide a redirectUri to enable the implementation to re-enter the client workflow.

### 5.2.1 Reply stage

The state of the **conversation**, as far as the directives for the next communication steps required to drive the conversation further are concerned, is represented in a "reply stage".

A simple reply stage may be a "result stage", indicating conversation end and transporting the result object.

A more sophisticated one is a "redirect stage", holding information about a web target that should be called in order to make the next step.

## 5.3 Document

A document is a central target for all services.

A document can be provided

- literal  
The document content data stream is provided at the service API. The content is copied and client and server do not share any data.
- by handle  
This special type is currently used internally only. A document can be referenced by its opaque handle to the document repository without copying any data.

The document is initially provided by the client and sent to the server via the **TransportDocument** type. On the server side, unless it was a handle to an existing document, the document is streamed to a repository location.

An example for a valid literal document

```
{
  "type": "d",
  "name": "mydoc.txt",
  "content": "<base64 content>"
}
```

Via the **repository** a **flow** can access the document data.

### 5.3.1 Document Properties

A **TransportDocument** can be decorated with generic document properties. The meaning of the document properties is defined in the context of the services only that act on the **TransportDocument** input and output. As such they are not really part of the document but "sidecar" information from or for the service implementation.

For example, the **ResultSigner** data structure uses the document properties to give a reference to the original signature target (which is important in case of detached signatures).

Example

```
{
  "type": "d",
  "name": "mydoc.pdf",
  "content": "<base64 content>",
  "properties": {
    "signature": {
      "targetName": "mydoc.pdf"
    }
  }
}
```

### 5.3.2 Document attachments

For some services, we need more than one document to completely describe the input. This is for example the case with CMS signatures, when a signature is detached from the document to be signed. If we want to provide a CMS signature along with the document, it is expected in the "attachments" of the document itself.

Example

```
{
  "type": "d",
  "name": "mydoc.txt",
  "content": "<base64 content>",
  "attachments": [
    {
      "type": "d",
      "name": "mydoc.txt.p7s",
      "content": "<base64 content"
    }
  ]
}
```

### 5.3.3 Tags

Tags denote key/value pairs that contain meta information for the document itself. They are directly attached to a document and available anywhere where a document is used.

The semantics of the document tags are defined in the operators that act on the document later on (if any).

A typical use in gears is the enrichment of operator arguments with parts that are specific to a given document. This way you can for example sign a batch of documents and define the location of the field individually per document. You can even define the field location in the document itself – for this technique please review chapter "workflow integration" in the security applications developers guide.

Document tags can stem from a variety of sources, each of them merged in the documents tag set.

#### 5.3.3.1 TransportDocument properties

Each entry in the "tags" namespace of the **TransportDocument** properties is propagated to the final document tags.

Example

```
{
  "type": "d",
  "name": "mydoc.pdf",
  "content": "<base64 content>",
  "properties": {
    "tags": {
      "foo": "bar",
      "zizz": {
        "zumb": "zork"
      }
    }
  }
}
```

This will result in the tag "foo" with value "bar" and the tag "zizz.zumb" with value "zork".

#### 5.3.3.2 Embedding

Tags may be provided by other sources, too.

The most interesting application is tag extraction from the target document itself. To activate tag detection, a **documentTagDetector** must be specified, as always in the factory/args convention. This is a direct argument to the service, on the same level as **documentSigner**.

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

```

As you see above, you add these arguments to the RequestSignerCreate. As can be derived from the name, this is suitable for PDF documents only. More details can be found in the "Security Applications Developers Guide".

An example document leveraging this technique is enclosed in the gears demo folder.

### 5.3.4 Document specific arguments

Now that we know document properties and tags, we can put them to use to define per-document arguments to a service call. 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.

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.

## 5.4 Repository

The backend processes documents provided by the client. These documents are held in a server-side **repository** for the duration of the **flow**.

The repository can be completely encrypted. The encryption may be purely static using a per-installation key derivation or even dynamic, where key derivation includes some client provided restricted identification for the flow.

The repository and the communication protocols work closely together, so that all document content is completely encrypted while receiving, processing the "flow" and transmitting the result. You will find all details for repository encryption in the section "Configuration" of the documentation.

Additionally, the repository holds the documents only for the lifetime of the flow. After flow termination or timeout, all data is erased.

You should consider that some security measures will lead to performance reductions.

## 5.5 Services

Web services are provided for the business functions of the product.

These services are designed using a procedural approach, so don't be disappointed when we do not push the "REST" buzzword.

We believe that this design approach is best suited for the product, its scalability and ease of use for the client.

Each service represents a **flow** and its manipulation. A request to the base address and the path "/create" will create and enter this **flow**.

Example

---

### service call

---

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

```
{
  "documents": [
    {
      "type": "d",
      "name": "foo.txt",
      "content": "aGVsbG8="
    }
  ]
}
```

---

The result of such a request is always a **conversation** representing the **flow** that was created. Some synchronous scenarios may even contain the result immediately (even so wrapped in the conversation protocol).

---

**service response**

---

```
200
Content-Type: application/json

{
  "snapshot": {
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Result",
      "id": "3",
      "result": {
        "@class": "de.intarsys.cloudsuite.gears.core.service.signer.api.ResultSigner",
        "value": {
          "documentNames": [ "foo.txt" ],
          "signatures": [
            {
              "type": "d",
              "name": "foo.txt.p7s",
              "content": "MIIEBAYJK...pbE"
            }
          ]
        }
      }
    },
    "conversation": "4e58daf6-b1f9-4f3d-a871-8125124b4f0c"
  }
}
```

In this case we receive a conversation snapshot for conversation '4e58daf6-b1f9-4f3d-a871-8125124b4f0c', holding a reply stage of type 'urn:intarsys:names:conversation:1.0:schemes:Result'. This indicates that the call was executed synchronously, a result was created immediately and serialized to the client.

In this case we find enclosed the **ResultSigner** holding the signature container for the requested document(s).

The content for the request and response is a Base64-encoded byte stream.

## 6. Web UIs

---

### 6.1 Overview

The product not only provides services, but standalone web UIs for integration in your workflows, too.

These components are not intended to run embedded in your web pages but are full-blown "mini" web applications on their own.

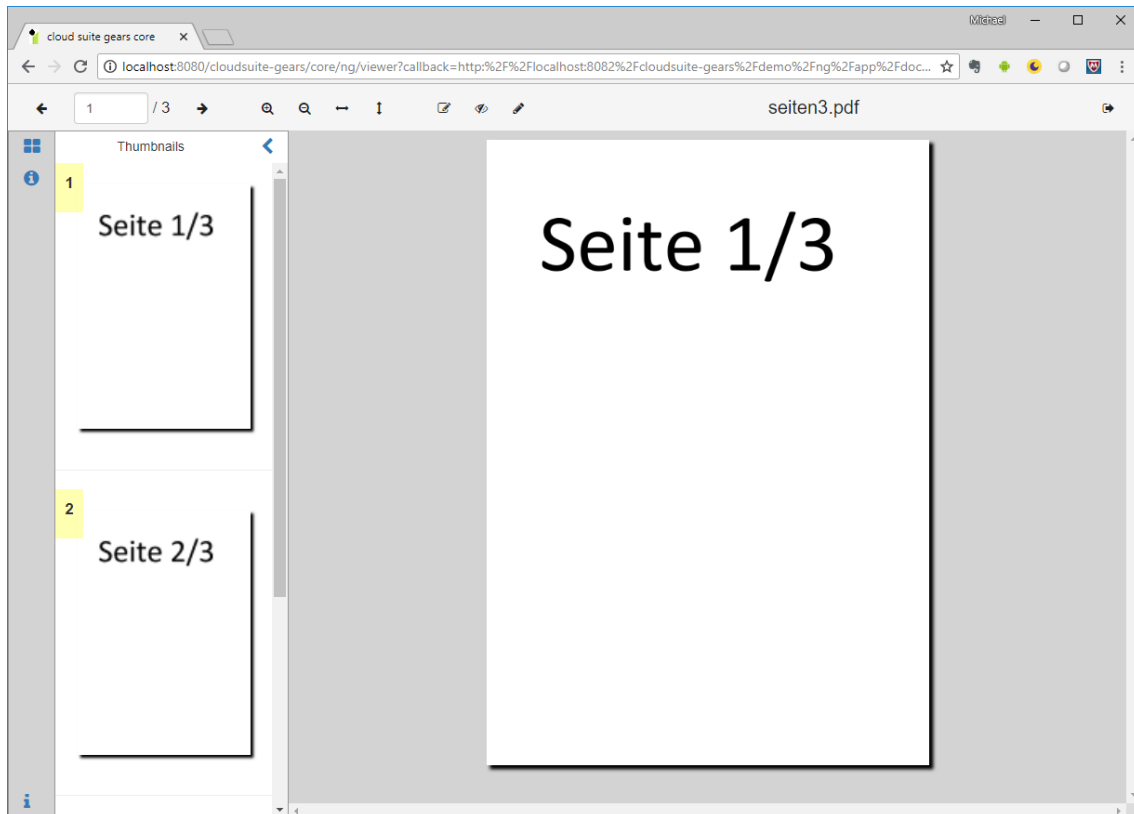
The workflow components are entered via a `HttpRedirectStage` you receive from our services and end with a call to your applications `redirectUri`.

### 6.2 Viewer

The first component provides a web-based viewing of a document. Many signature workflows require a "trusted viewer", enabling the user to review the target of signature before authorization.

The web viewer is a HTML5 application that uses server-side rendering and as such should run on most modern browsers and devices. The content is guaranteed to look the same regardless of environment.

You can enter the web viewer by starting the "viewer flow" using "viewer/create". You will be redirected to the viewer web application. Details for calling the web viewer can be found in the "Viewer" reference section.



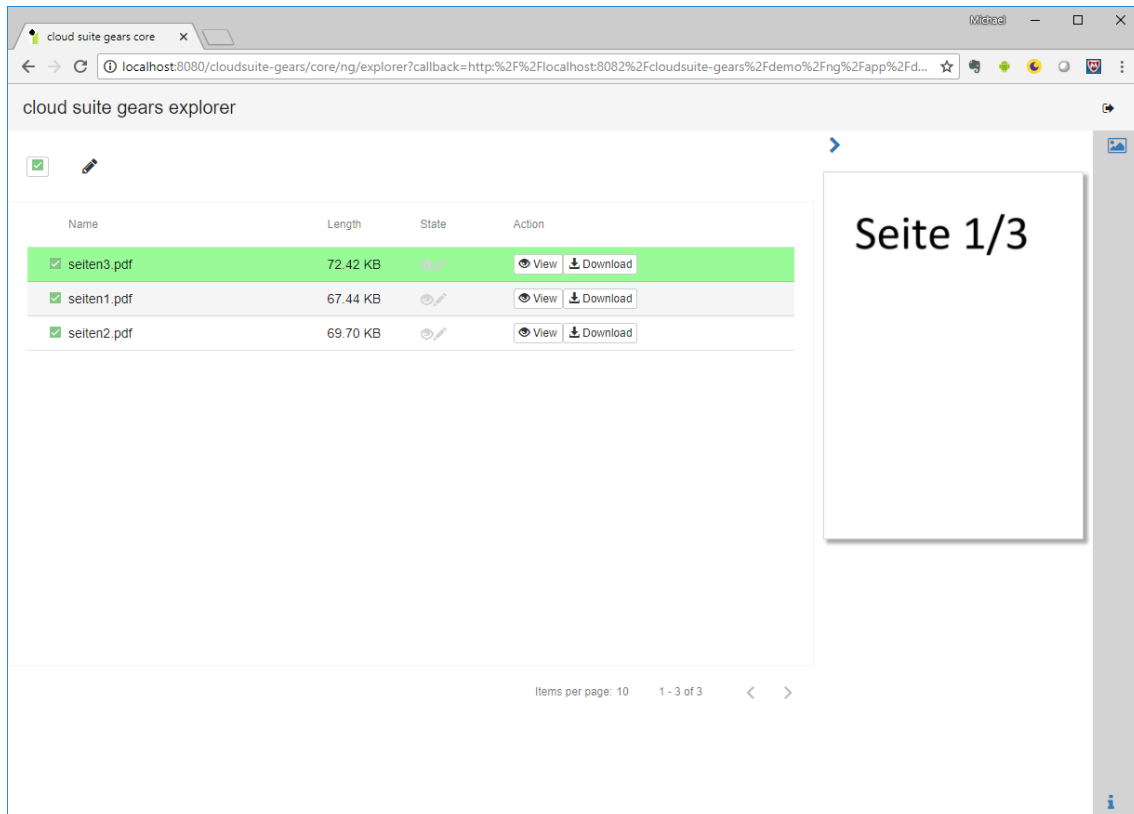
In the toolbar you can see some standard buttons to control the viewer.

The buttons in the middle are automatically created by using user defined widgets & actions. You will find information on this in the later chapters.

With the button on the right, you can return to your calling application.

## 6.3 Explorer

The explorer allows to see and control the batch of documents provided by the client for further processing.



Again, depending on your integration needs, you can define pluggable actions that are applied to the batch upon user interaction.

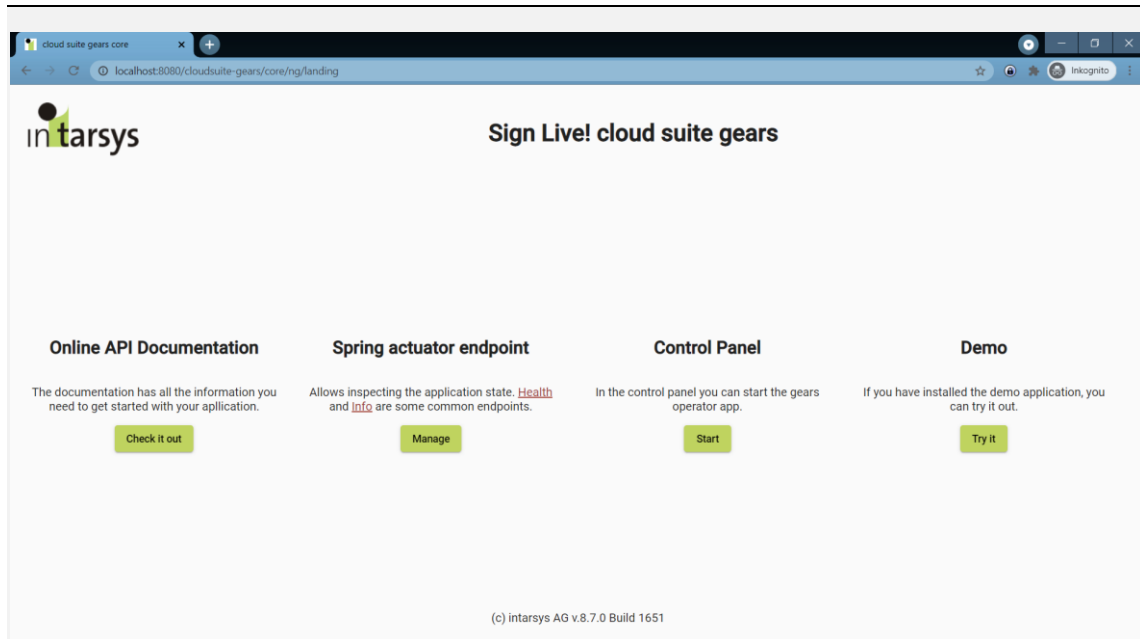
## 6.4 Control Panel

The console acts directly on the gears server state and is used by an operator that needs to control runtime behavior.

To access the console, the user needs the "operator" role. This role is by default granted without authentication. See chapter "Authentication" for detailed information about the security concept.

Currently, the frontend only supports basic authentication.

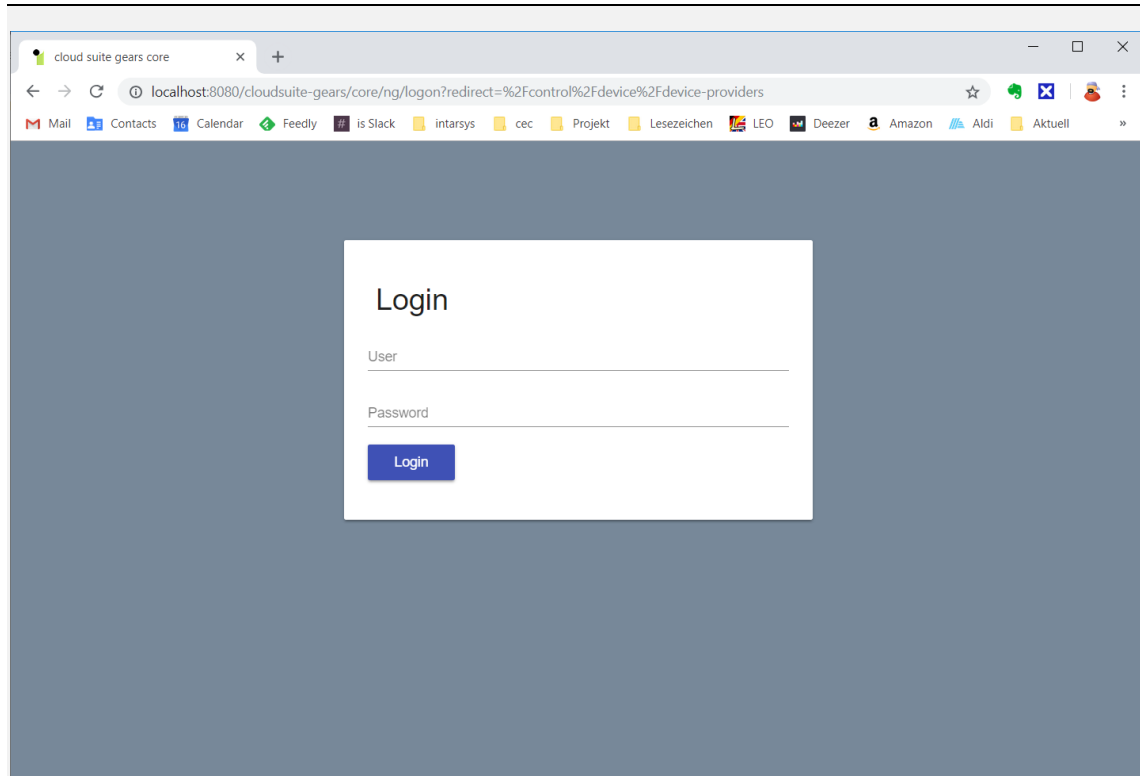
## 6.4.1 Landing Page



When you try to access gears (<https://host/cloudsuite-gears/core/>) you are redirected to the landing page in the user interface path of the application (`/ng`). From here you can access online documentation, the actuator endpoints and the control panel.

## 6.4.2 Authentication

If you switch to the control panel and basic authentication is in place, you will be prompted for your credentials.



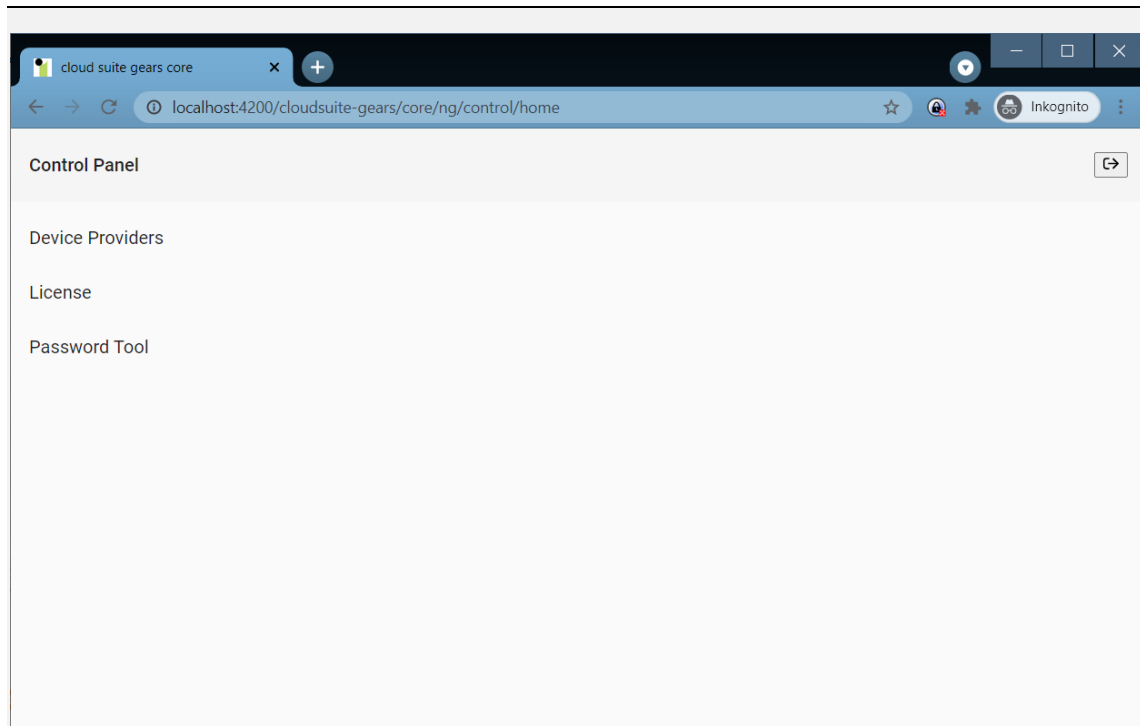
This depends on your setup, by default, access is granted.



## 6.4.3 Control Panel Home

Now you see the control panel home page. You can navigate to specific control sections from here:

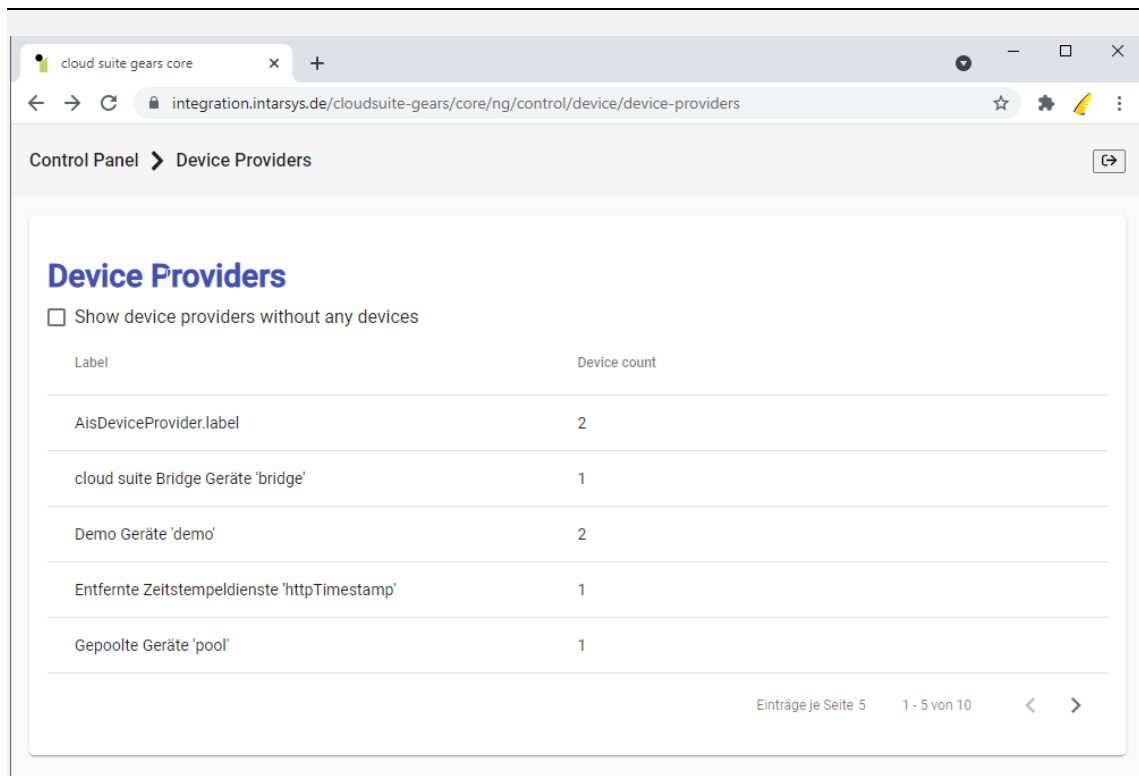
- Device Providers -> Device Management
- License -> License Management
- Password Tool -> Password Tool



## 6.4.4 Device Management

### 6.4.4.1 Device Providers

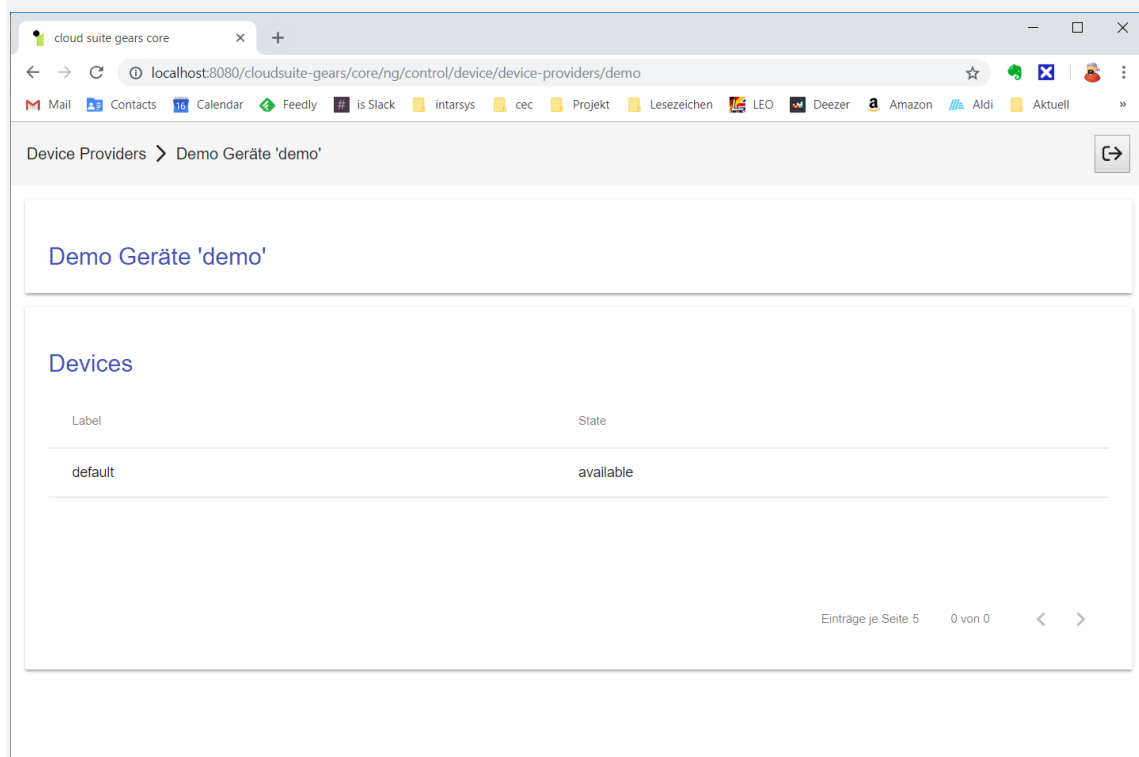
The page displays internal state information from the running gears instance. It starts with a list of all active device providers.



You see the device providers name, along with the active devices. Select "Show device providers without any devices" to get a list of all device providers, regardless of state.

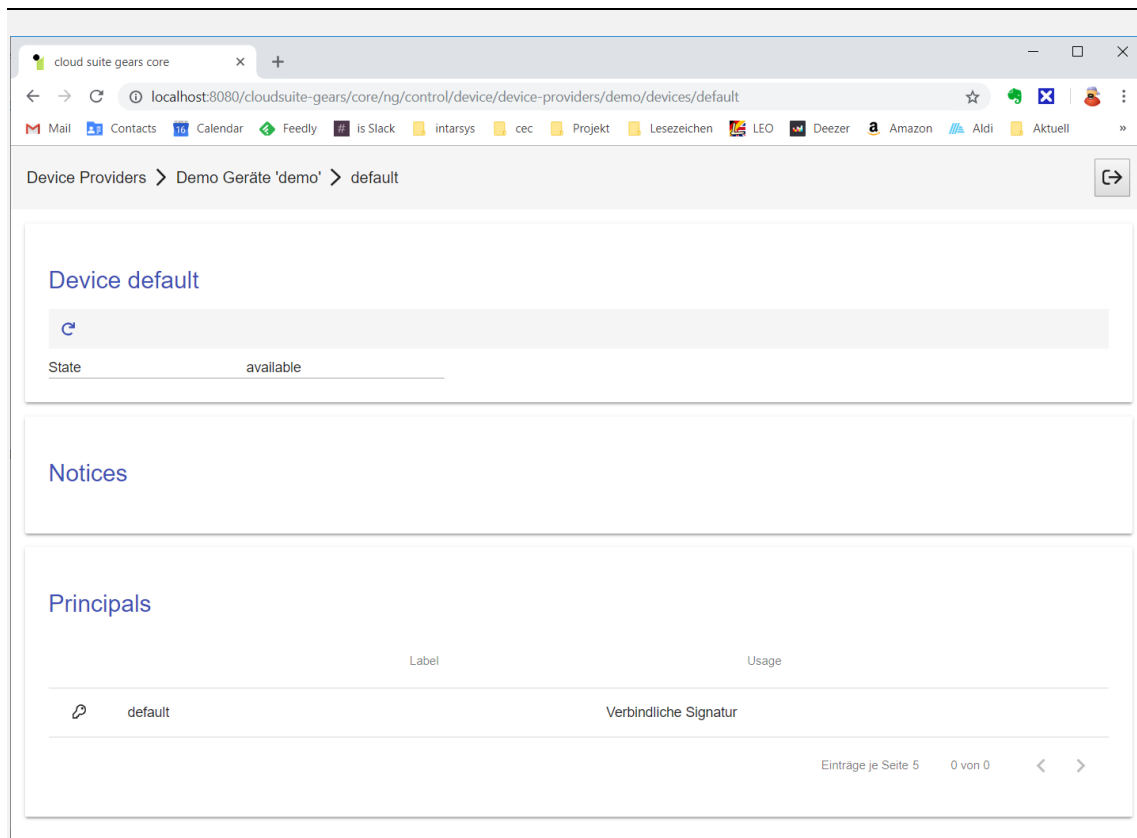
Clicking on a device provider brings you to the list of the devices of this provider, e.g. the demo devices.

#### 6.4.4.2 Devices



From here you can drill in the single device.

## 6.4.4.3 Device



This page shows you the general device attributes, depending on the device type. In the action bar of the first card you can request a "Renew", which means the equivalent of detaching and attaching the device. For a smartcard for example, this emulates a removal and insertion.

The second card, "Notices", shows active messages attached to the device.

The third card, "Principals" shows all identities available for this device. In general, this is at least one key (represented by the key icon). A principal with a key allows you to perform cryptographic operations if you have the required secret.

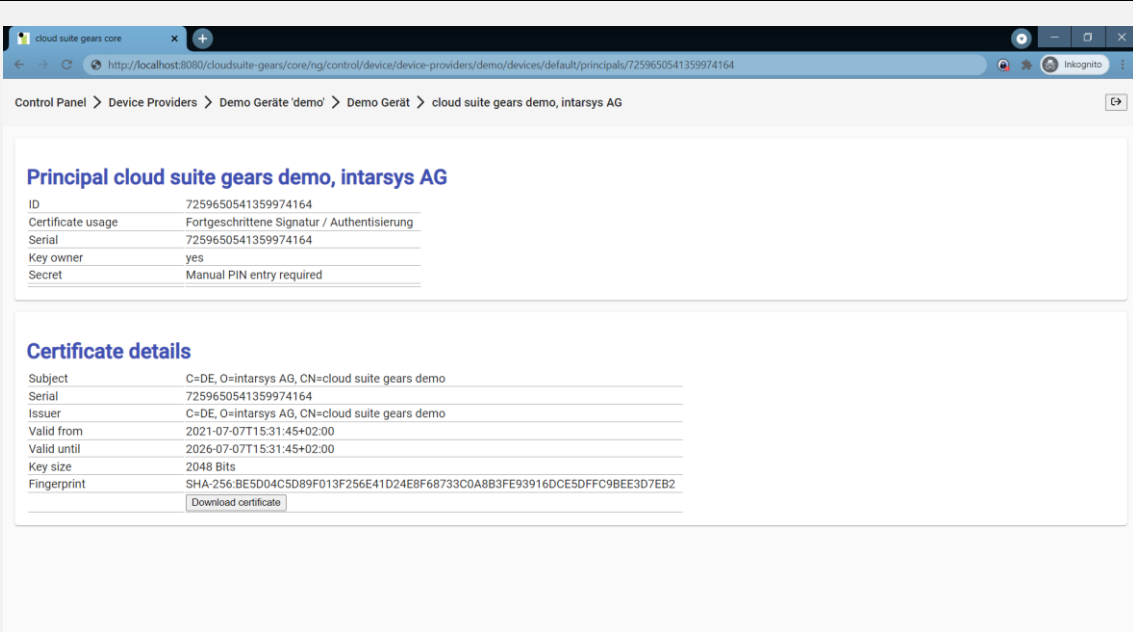
In addition, you often see other identities, marked with a certificate icon.

Principals		
	Label	Usage
	Testkarte:PN	Fortgeschrittene Signatur / Authentisierung
	DATEV Root CA 1 2014, DATEV eG	Zertifizierungsstelle
	DATEV CA 1 2014, DATEV eG	Zertifizierungsstelle

These are most often public key certificates that can be used for verifying the key holder certificate itself.

#### 6.4.4.4 Principal

Finally, you can drill down into the principal itself.



The screenshot shows a web browser window with the URL `http://localhost:8080/cloudsuite-gears/core/ng/control/device/device-providers/demo/devices/default/principals/7259650541359974164`. The breadcrumb navigation is: Control Panel > Device Providers > Demo Geräte 'demo' > Demo Gerät > cloud suite gears demo, intarsys AG. The main content area is titled "Principal cloud suite gears demo, intarsys AG" and contains two sections:

Principal cloud suite gears demo, intarsys AG	
ID	7259650541359974164
Certificate usage	Fortgeschrittene Signatur / Authentisierung
Serial	7259650541359974164
Key owner	yes
Secret	Manual PIN entry required

Certificate details	
Subject	C=DE, O=intarsys AG, CN=cloud suite gears demo
Serial	7259650541359974164
Issuer	C=DE, O=intarsys AG, CN=cloud suite gears demo
Valid from	2021-07-07T15:31:45+02:00
Valid until	2026-07-07T15:31:45+02:00
Key size	2048 Bits
Fingerprint	SHA-256:BE5D04C5D89F013F256E41D24E8F68733C0A8B3FE93916DCE5DFFC9BEE3D7EB2

Below the fingerprint, there is a button labeled "Download certificate".

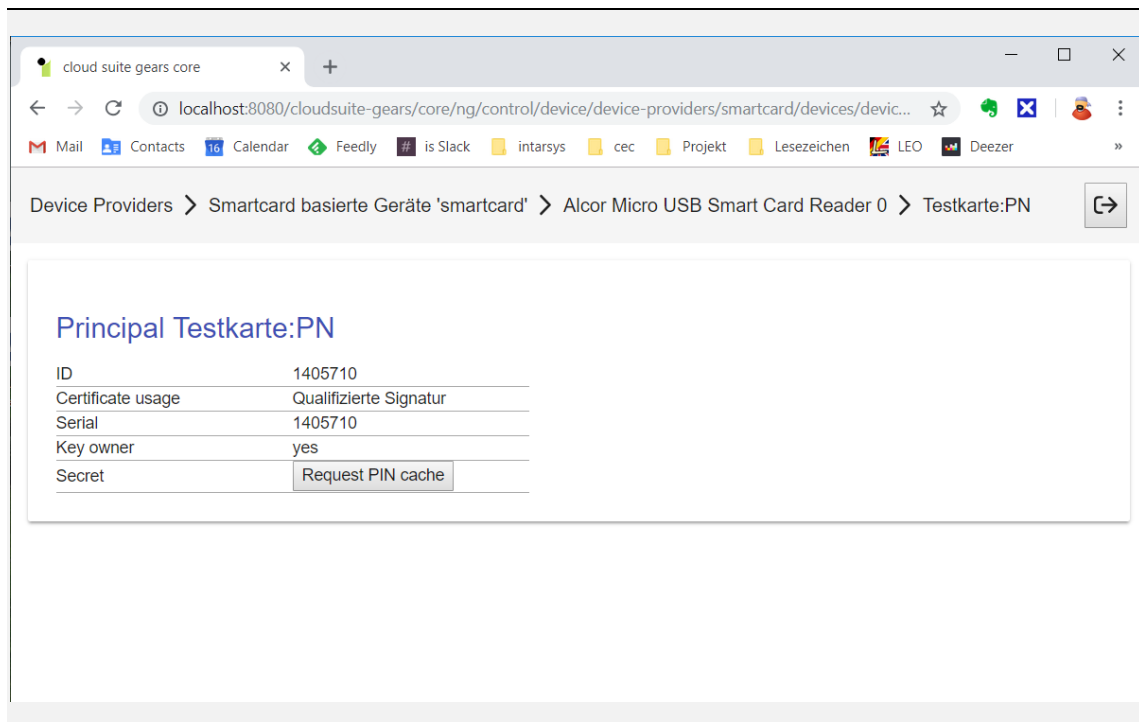
Here you see attributes of the principal and the details of the respective X.509 certificate.

You can download the DER-encoded binary certificate to a local file by clicking the button "Download certificate". In order to check the file integrity, compute the file's SHA-256 hash value (e.g. using a Windows checksum tool) and compare it to the fingerprint shown in the certificate details section.

#### 6.4.4.5 PIN Caching

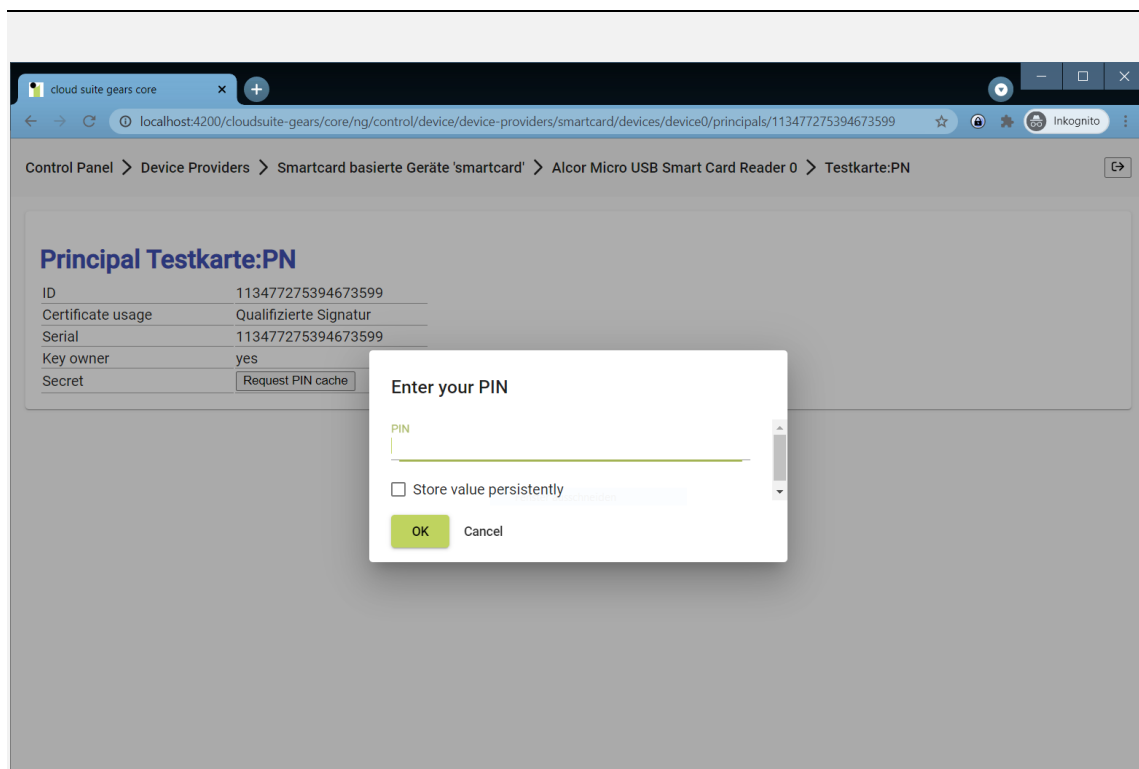
An interesting field is "Secret". Some principals on some devices support PIN caching, i.e. you can enter the PIN once in the console. Later use of the principal (the device) **do not need to be authenticated** by the key holder.

Be warned that this mode needs careful design of process and execution environment to ensure a secure signature process. Some signature creation devices (e.g. certain smartcards) **require** that the PIN is entered by the key holder to explicitly signal consent to the key usage.



This is the hardcopy of a smartcard based key holder principal, where the smartcard itself is configured to support PIN caching (see `deviceProviders.smartcard.allowSecretFromCache`).

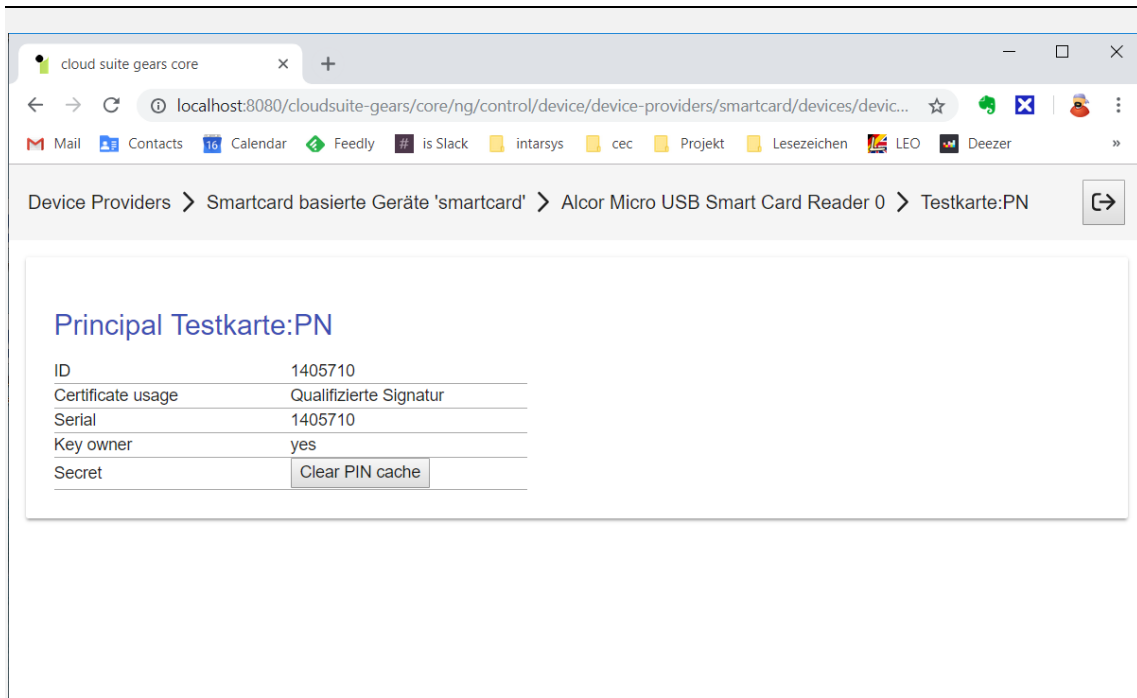
You can press "Request PIN cache" to prompt for the PIN. As the PIN is confirmed against the card this is only possible if the card is not currently active (e.g. in some pool)!



The PIN can be cached either transiently (default) or persistently. Transient mode keeps the PIN available in memory throughout the server application's up-time and will require you to reenter it upon application restart (e.g. when rebooting the server). You can achieve persistent PIN storage by explicitly selecting "Store value persistently". In this

case, the PIN will be stored encrypted in the gears runtime configuration data and thus survive application restarts until explicitly cleared.

If the PIN is already detected in the cache, you can opt to clear it:



*One note on PIN storage:*

*The storage mechanism uses the default cryptdec registered with the system for encryption. Assuming factory settings, this is a cryptdec deriving its key from the application's master key. So be aware that once you change the master key, cached PINs will become unrecoverable and thus have to be cleared and recached.*

#### 6.4.4.6 Pool Device

The pool device has some special knobs, so there's a chapter dedicated to this device.



First, we have special properties describing the pool:

#### State

started | stopped | suspended

If the pool is started, it accepts requests and delegates to an idle security application. If it is suspended, requests are accepted but queue up. If requests keep up to arrive, the container will soon run out of threads, though.

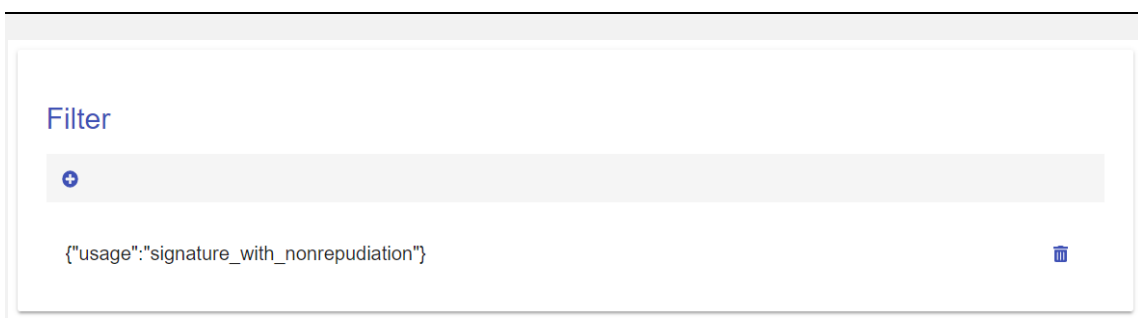
If it is stopped, requests are denied. For health indication, a stopped pool will be published as "DOWN".

Size	The number of registered security applications (e.g. smartcards)
Active	The number of currently active security applications (active requests)
Pending	The number of requests currently waiting for a security application.

The others are properties that are assigned in the pool configuration and are presented for reference purposes.

In the action bar, you can start, stop and suspend the pool, using the respective icons from the action bar.

The second feature for the pool is the management of "filters".



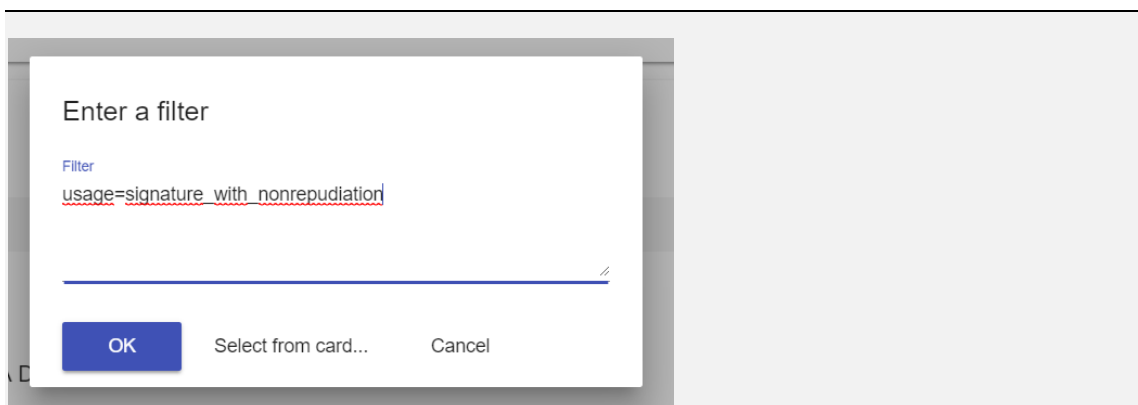
A pool monitors the availability of smartcards (respective the keys they contain). If it detects a key that is suitable for use in the pool, it tries to authenticate and register the resulting security application.

Using filters, you can restrict the smartcard it tries to accumulate.

The default filter for a pool is "usage=signature\_with\_nonrepudiation", which denotes all certificates that qualify for "qualified signatures" (pun intended). This means, this pool will by default acquire every smartcard that has a "non repudiation" key on it.

The trash icon will delete a filter (no filter means this pool will not acquire anything).

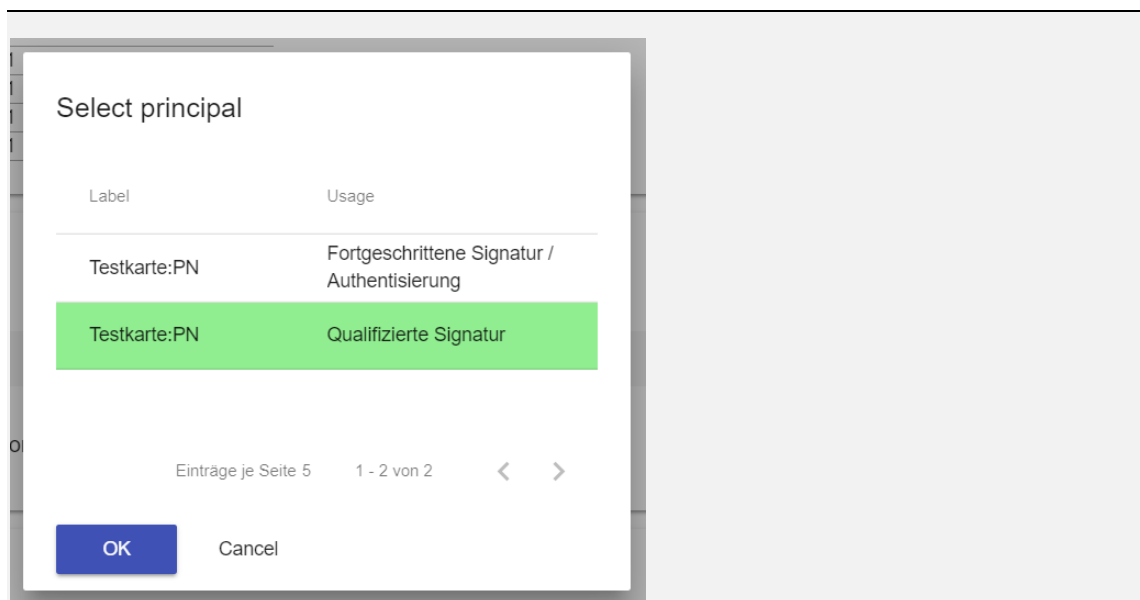
The plus icon allows you to define a filter and opens a dialog box.



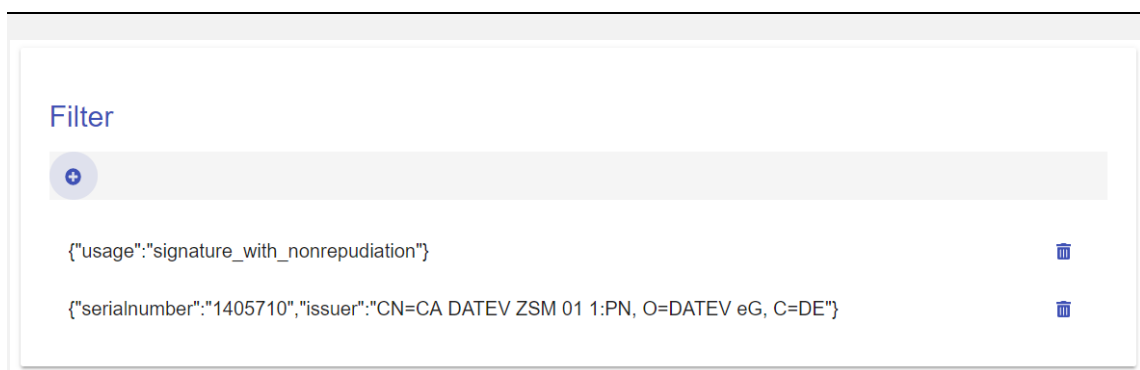
You can either directly enter a valid filter expression (this is documented in the "security application developers guide") or simply read the constraints directly from a card if you press "Select from card...".



First you are asked to select one of the connected card terminals.



Then you select one of the available principals.



The resulting filter expression is added to the list.



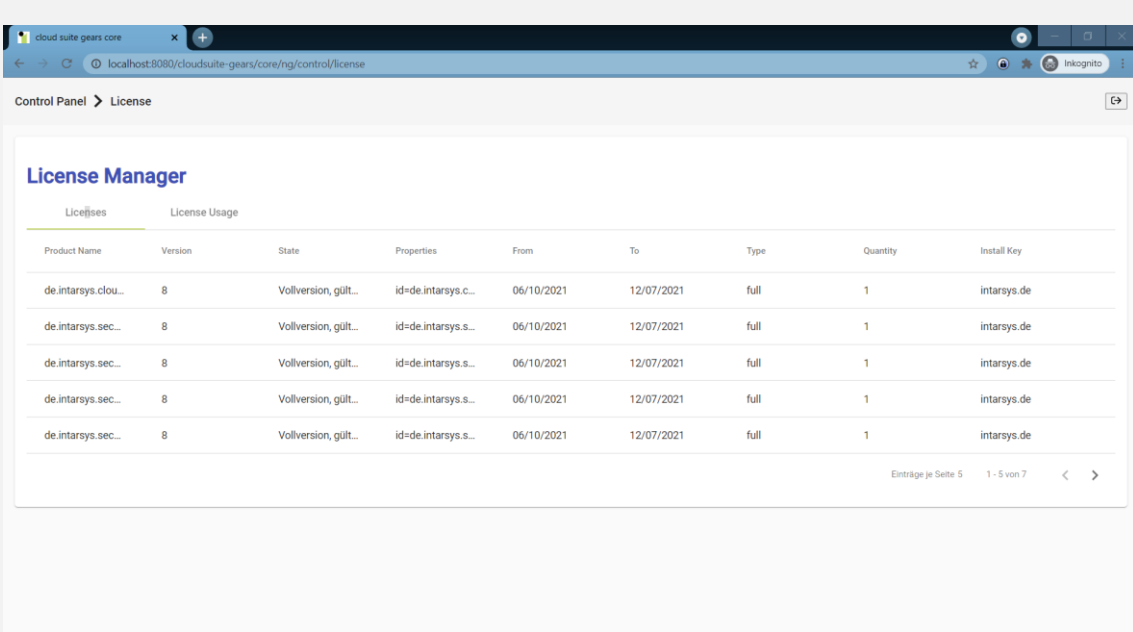
## 6.4.5 License Management

This section provides information on the following license aspects:

- installed license files and corresponding states
- license accounts and corresponding usage states

### 6.4.5.1 Licenses

The "Licenses" tab gives an overview of the licenses currently installed in the system:



The screenshot shows a web browser window with the URL `localhost:9080/cloudsuite-gears/core/ng/control/license`. The page title is "License Manager". Below the title, there are two tabs: "Licenses" (selected) and "License Usage". The "Licenses" tab displays a table with the following columns: Product Name, Version, State, Properties, From, To, Type, Quantity, and Install Key. The table contains five rows of license data, all with a quantity of 1 and an install key of "intarsys.de".

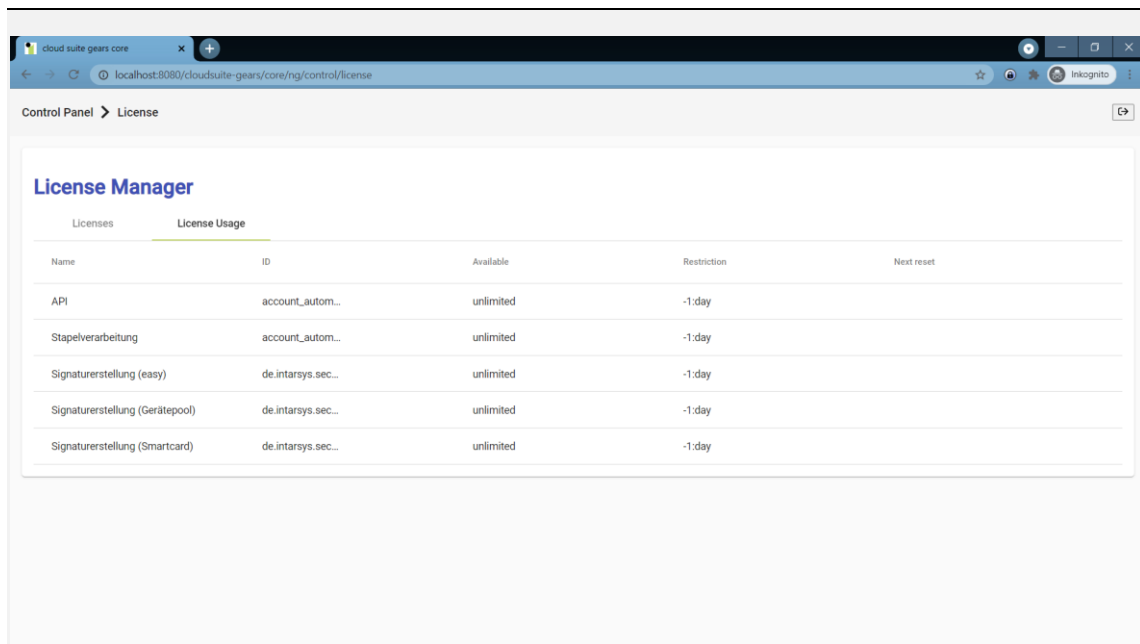
Product Name	Version	State	Properties	From	To	Type	Quantity	Install Key
de.intarsys.clou...	8	Vollversion, gült...	id=de.intarsys.c...	06/10/2021	12/07/2021	full	1	intarsys.de
de.intarsys.sec...	8	Vollversion, gült...	id=de.intarsys.s...	06/10/2021	12/07/2021	full	1	intarsys.de
de.intarsys.sec...	8	Vollversion, gült...	id=de.intarsys.s...	06/10/2021	12/07/2021	full	1	intarsys.de
de.intarsys.sec...	8	Vollversion, gült...	id=de.intarsys.s...	06/10/2021	12/07/2021	full	1	intarsys.de
de.intarsys.sec...	8	Vollversion, gült...	id=de.intarsys.s...	06/10/2021	12/07/2021	full	1	intarsys.de

You can hover over the single table cells in order to display their full value. The table columns present the following information:

Name	Description
Product Name	The product name this license has been issued for.
Version	The product version this license has been issued for.
State	The license's validity state.
Properties	The license's internal properties.
From	Begin of license validity period.
To	End of license validity period.
Type	The license type.
Quantity	The number of licensed units.
Install Key	The license's installation key.

### 6.4.5.2 License Usage

The "License Usage" tab gives an overview of the available license accounts and their current usage state:

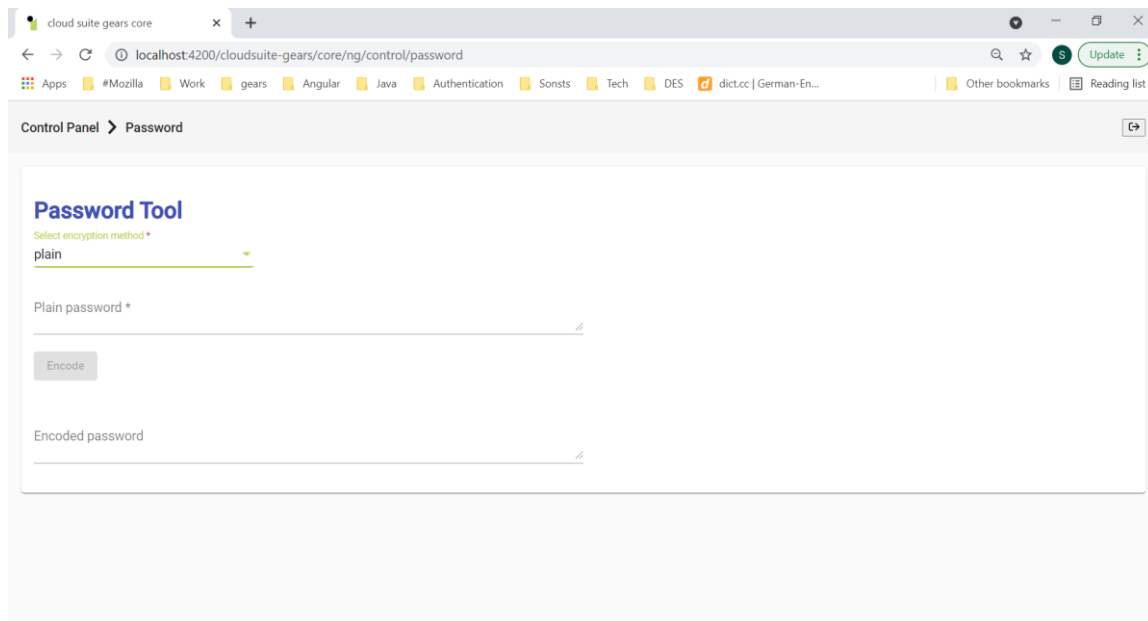


You can hover over the single table cells in order to display their full value. The table columns present the following information:

Name	Description
Name	The name of the monitored account.
ID	The ID of the monitored account.
Available	The number of units currently available on the account.
Restriction	The restriction being active on the account.
Next reset	The time when the account will be reset, providing all licensed units again.

### 6.4.6 Password Tool

This section provides an option to encrypt the plain password to an encoded password by choosing the available encryption algorithms.



This page allows you to select an encryption algorithm available in the application using the drop-down menu and a text field to provide a plain password. On providing these two mandatory values, the “Encode” button is enabled. You can press this button and obtain the encoded password in the “Encoded password” text field.

## 6.5 Browser security

### 6.5.1 Headers

All web based UIs rely on common browser security features. When requesting UI components for rendering, gears will always provide these headers.

#### 6.5.1.1 referrer-policy

The referrer policy is always set to “no-referrer”.

This is currently not configurable.

#### 6.5.1.2 hsts

when HTTPS is requested, the HSTS header is always sent.

This is currently not configurable.

#### 6.5.1.3 content-security-policy

This technique supersedes the use of many other security headers and is the main means for gears to control browser security.

The default setting is

```
report-uri ../api/v1/meta/csp/report; report-to default; default-src 'self'; style-src 'self' 'unsafe-inline'; font-src 'self'; img-src 'self' blob: data:; object-src 'none'; frame-ancestors 'none'; connect-src *;
```

The options in detail

Option	Description
report-uri ../api/v1/meta/csp/report	All csp violations are reported back to gears. You can find the entry in the log file.

	At the time of writing, the report-uri directive is deprecated and will be replaced by report-to. However, in some browsers it is the only supported way to create CSP reports (e.g. Firefox). Therefore, we use it <b>together</b> with report-uri to ensure CSP reports are captured across all browsers.
report-to default	This directive implements the new Reporting API (v1), which uses the Reporting-Endpoints header (see below). All csp violation reports are sent in groups back to gears. You can find the entry in the log file. Note: report-to only works with HTTPS enabled.
default-src 'self'	Sources may only stem from the page origin
style-src 'self' 'unsafe-inline'	Styles may stem from the page origin *and* may be set inline. This is a requirement of the Angular framework and is considered safe.
font-src 'self'	Fonts must stem from the page source
img-src 'self' blob: data:	Images must stem from the page source or from script evaluations
object-src 'none'	No objects may be embedded
frame-ancestors 'none'	The page itself may not be embedded. This is an options you may need to overwrite when using an embedded gears process.
connect-src *	The page code may connect to any server.

#### 6.5.1.4 Reporting-Endpoints

The header is set to default="/cloudsuite-gears/core/api/v1/meta/csp/report".  
This is currently not configurable.

### 6.5.2 Header configuration

For spring internal reasons you cannot overwrite the bean defining the security realm (as opposed to many other beans). As we consider this a “hotspot” regarding individual integrations, we provide special properties to overwrite the certain headers and options.

The properties have the form

security.http.<realm>.headers.<header>.\*

- for each of the UI security realms
  - securityRealmUICore  
The gears UIs themselves
  - securityRealmUIBridge  
The bridge UI code
  - securityRealmUIAuth  
Some authentication UIs
  - securityRealmUIUpreg  
The UPreg UI code
- for each of the headers
  - content-security-policy

#### 6.5.2.1 content-security-policy.directives

The value of the property is the complete header content sent to the browser.

Example

```

security.http.securityRealmUICore.headers.content-security-policy.directives= report-uri
../api/v1/meta/csp/report; report-to default; default-src 'self'; style-src 'self'
'unsafe-inline'; font-src 'self'; img-src 'self' blob: data:; object-src 'none'; frame-
ancestors 'none'; connect-src *;
security.http.securityRealmUIBridge.headers.content-security-policy.directives= report-
uri ../api/v1/meta/csp/report; report-to default; default-src 'self'; style-src 'self'
'unsafe-inline'; font-src 'self'; img-src 'self' blob: data:; object-src 'none'; frame-
ancestors 'none'; connect-src *;
security.http.securityRealmUIAuth.headers.content-security-policy.directives= report-uri
../api/v1/meta/csp/report; report-to default; default-src 'self'; style-src 'self'
'unsafe-inline'; font-src 'self'; img-src 'self' blob: data:; object-src 'none'; frame-
ancestors 'none'; connect-src *;
security.http.securityRealmUIUpreg.headers.content-security-policy.directives= report-
uri ../api/v1/meta/csp/report; report-to default; default-src 'self'; style-src 'self'
'unsafe-inline'; font-src 'self'; img-src 'self' blob: data:; object-src 'none'; frame-
ancestors 'none'; connect-src *;

```

Example, allow embedding. Note the missing “frame-ancestors”

```

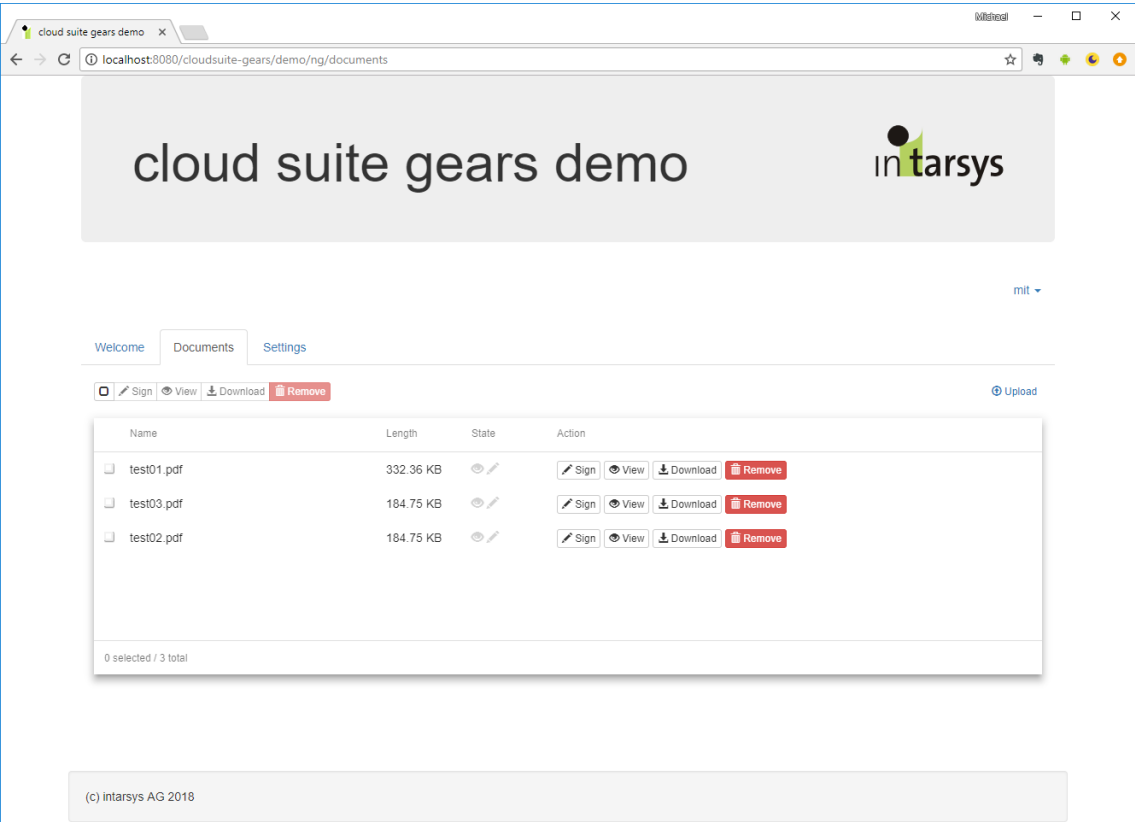
security.http.securityRealmUICore.headers.content-security-policy.directives=report-uri
../api/v1/meta/csp/report; report-to default; default-src 'self'; style-src 'self'
'unsafe-inline'; font-src 'self'; img-src 'self' blob: data:; object-src 'none';
connect-src *;

```

# 7. Demo application

## 7.1 Overview

The product is shipped including a web-based demo application.



The demo client provides interactive access to all cloud suite services.

It is distributed with complete source, both the web app and the backend.

## 8. Crypto components

### 8.1 Basics

Sign Live! cloud suite gears includes a set of cryptographic basic components that are plugged together to achieve the desired security goal.

#### 8.1.1 IByteProvider & IByteStreamProvider

At the lowest level you can inject bytes into the system. More details can be found in the reference section below.

The relevant implementations are

**de.intarsys.tools.crypto.bytes.RandomByteProvider**

providing data from a secure random data source

**de.intarsys.tools.crypto.bytes.StaticByteProvider**

providing static data from configuration or system sources.

Their most important role is the generation of the input for key derivation in the application.

**de.intarsys.tools.crypto.bytes.PbkdfByteProvider**

Derive bytes from a password (low entropy) input.

**de.intarsys.tools.crypto.bytes.DerivedByteProvider**

Derive bytes from a KDF function.

#### 8.1.2 IKeyDerivationFunction

With a key derivation function, you can create real, independent key material for every component and subcomponent of the application, using the same base input material.

The relevant implementations are

**de.intarsys.tools.crypto.kdf.HashedKeyDerivationFunction**

Hashed key derivation as defined in RFC-5869. This should define the root of your key tree.

**de.intarsys.tools.crypto.kdf.WithPrefixKeyDerivationFunction**

Takes the bytes from an IByteProvider and merges KDF input before feeding into another IKeyDerivationFunction. This is used to define subtrees of the key tree.

#### 8.1.3 ICipherFactory

Finally, you will use the key in a cipher for encrypting and decrypting data.

The ones you will work with are

**de.intarsys.tools.crypto.standard.NullCipherFactory**

A non-encrypting cipher

**de.intarsys.tools.crypto.standard.JcaCipherFactory**

A general cipher factory for using the JCA architecture.

**de.intarsys.tools.crypto.standard.DerivedKeyCipherFactory**

Takes the secret and feeds it into a KDF before forwarding it to another ICipherFactory

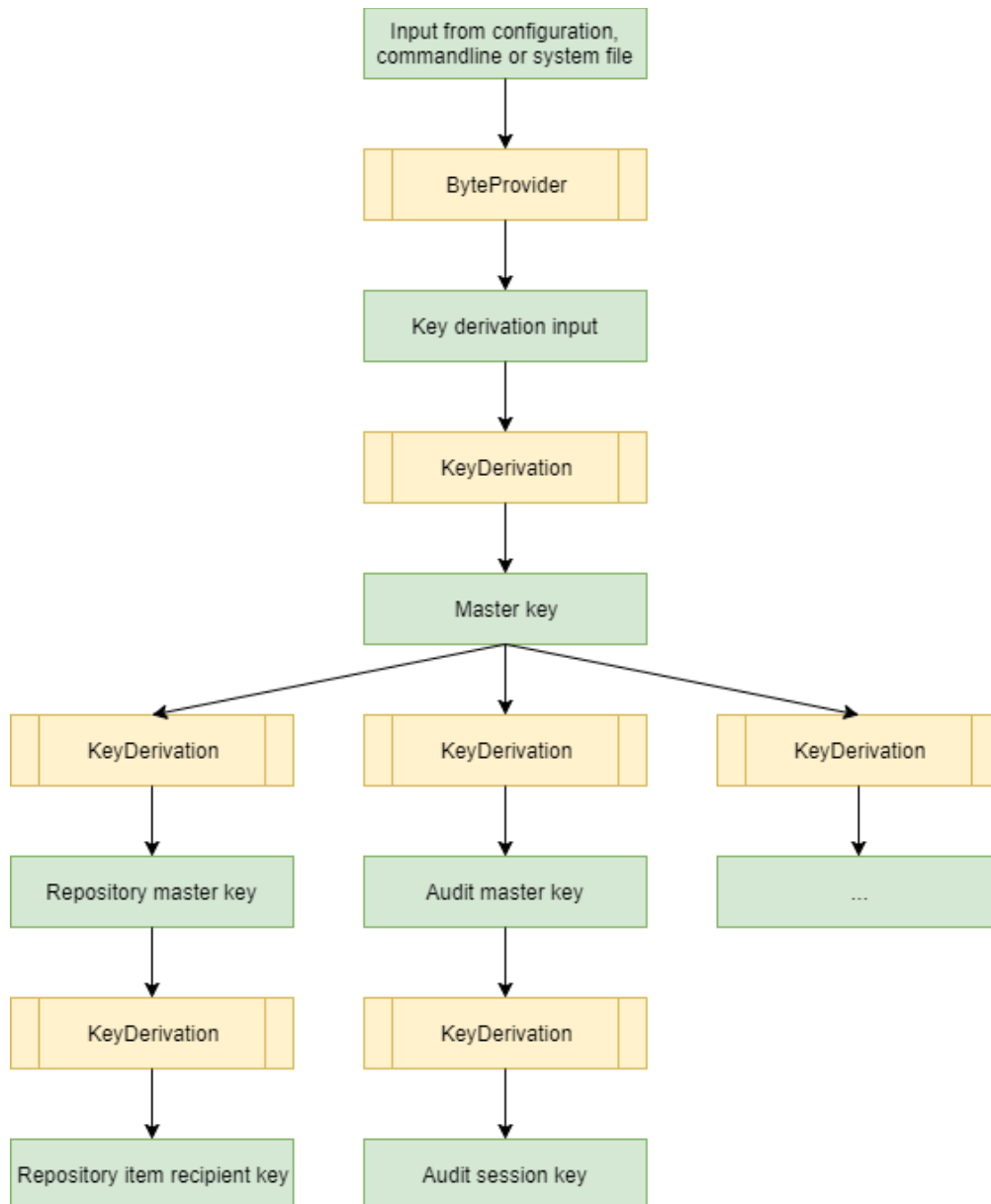
**de.intarsys.tools.crypto.standard.StaticKeyCipherFactory**

Ignore the secret and create a cipher using a static key definition.



### 8.1.4 Key tree

A typical usage of these primitives may help in understanding:



Such a hierarchy should be defined in the application configuration, depending on your needs.

## 8.2 Cryptdec

A cryptdec is a component that represents an encryption algorithm and the required key material.

### 8.2.1 Plain

The most basic cryptdec variant that is always available is the "plain" cryptdec.

It does simply nothing but BASE64 en-/decoding the data.

## Example

```
plain#Zm9v
```

represents the text string

```
foo
```

## 8.3 Secret

A "secret" is an implementation artifact that encapsulates a cipher and an encrypted string or byte sequence. It is used internally to represent, well, secrets.

The advantage of this component is that

- it can be flexibly mapped to external representations upon configuration
- it is not visible in a memory dump, as a decrypted version only exists on the application stack, not in the heap.

### 8.3.1 Syntax

A secret is made up from an id tag and the serialized, encrypted data, separated by a "#".

```
<id>#<data>
```

"id" identifies a "cryptdec" to be used for en/decryption of the data. The representation of data is private to the cryptdec that is selected by "id".

The "plain" cryptdec (see above) for example simply uses BASE64 en/decoding.

### 8.3.2 Configuration

If a secret value (like a password or PIN) is required in the application you have normally three options you can choose from:

#### 8.3.2.1 Plaintext

You can set a secret using its plaintext representation:

```
my.secret.property=foo
```

This is not recommended for production use, unless you can ensure that nobody has access to the configuration files. But it is quite useful for starting an installation and testing around.

#### 8.3.2.2 Spring string expansion

You can leverage the spring string expansion feature with the property source "secret". This will take the suffix of the expression and parse it as a secret according to the secret syntax above

```
my.secret.property=${secret.plain#Zm9v}
```

For sure, "plain" is not the best choice for production use, but you can set up your own cryptdec for this purpose.

This property is expanded by Spring when the XML file is parsed and the beans are created. It is always converted to an (intermediate) string, before gears can convert it back

to the internal secret object. This does no harm to the confidentiality of the secret – but any binary data that cannot be properly mapped to a Java String will be corrupted.

The advantage is that this notation is completely agnostic to the internal implementation. You can use a secret to obfuscate the configuration even for non-secret values – they will be mapped correctly to the internal string or secret.

### 8.3.2.3 gears string expansion

The gears expansion kicks in later, when the property value is assigned to the secret property. This allows to use any valid secret string, because no intermediate format is created.

---

```
my.secret.property=?{secret.plain#Zm9v}
```

---

Take note of the "?" instead of the "\$".

While the advantage is to support binary data, the drawback is that this syntax can only be used if expansion is explicitly supported by the target.

## 8.4 Repository encryption

The repository is by default unencrypted.

For every document a corresponding folder, containing meta data and the content stream, is created.

The folder is deleted immediately after the containing flow is terminated.

If you need additional confidentiality, you can request encryption of the content (see details in the reference section). The content is encrypted using a two-stage process (comparable to the CMS crypto standard).

Although we are using efficient random-access encryption, encrypting the content will degrade service performance, contributing up to 50% additional runtime - the services are stateless and accessing content will constantly decrypt from the repository.

## 9. Configuration

### 9.1 Overview

Sign Live! cloud suite gears is highly configurable and comes with a bunch of possibilities where to tweak the installation. Here's an overview of the configuration mechanics.

The backend is based on Spring infrastructure and as such you have all well-known Spring customization tools at hand.

Whenever we reference an XML based configuration file, we use the term "bean definition", when we talk about Spring properties (key/value pair definitions) we use the term "property definition".

These terms are augmented with "built-in" when we mean hardcoded, pre-deployed definitions and "custom" when we talk of individual definitions invented by your configuration.

#### 9.1.1 Configuration location

The configuration tries to harmonize Windows and Linux based installations. We only use "abstract" location names by default. Here's the list of locations supported.

Variable	Description
cloudsuite.config.name	Name of configuration file to be used. Defaults to "gears".
cloudsuite.config.user	User individual configuration data. Here you can store e.g. individual property files. This location has highest precedence.
cloudsuite.config.shared	System wide configuration data. Here you can store e.g. system wide property files. This location overrides the built-in configuration and is overridden by the user individual configuration.
cloudsuite.data.user	User individual state data. Here is where user specific databases, caches etc. will reside.
cloudsuite.data.shared	System wide state data. Here is where system specific databases, caches etc. will reside.
cloudsuite.temp.dir	Location for temporary data, defaults to java.io.tmpdir.
cloudsuite.log.dir	Location for writing log files.

These variables are mapped differently on different platforms to adhere to platform specific conventions.

All of these basic variables can be overridden the standard Spring way:

- Use a command line parameter to the Java VM  
-Dcloudsuite.data.shared=/srv/data/cloudsuite
- Use an environment variable  
CLOUDSUITE\_DATA\_SHARED=/srv/data/cloudsuite
- Use a key/value pair entry in a properties definition file like "gears.properties".  
It's understood that you cannot set the "cloudsuite.config.\*" properties in a property file (it would have only strange effects).

If a configured directory does not exist, the web application will try to create it. For this it will need write access to the parent directory. Alternatively, you can create the directories yourself and make sure the web application has permission to write to them.

#### 9.1.1.1 Windows

Variable	Description
cloudsuite.config.name	gears
cloudsuite.config.user	%USERPROFILE%/cloudsuite/config
cloudsuite.config.shared	%ProgramData%/cloudsuite/config
cloudsuite.data.user	%USERPROFILE%/cloudsuite/data
cloudsuite.data.shared	%ProgramData%/cloudsuite/data
cloudsuite.temp.dir	%AppData%/local/temp
cloudsuite.log.dir	%ProgramData%/cloudsuite/log

#### 9.1.1.2 Linux

Variable	Description
cloudsuite.config.name	gears
cloudsuite.config.user	<user home>/cloudsuite/config
cloudsuite.config.shared	/etc/cloudsuite
cloudsuite.data.user	<user home>/cloudsuite/data
cloudsuite.data.shared	/var/lib/cloudsuite
cloudsuite.temp.dir	/tmp
cloudsuite.log.dir	/var/log/cloudsuite

If you are using this default, you must create the directories and grant access to the user running the servlet container.

If you should deviate from this default, keep in mind that the values of "cloudsuite.config.user" and "cloudsuite.config.shared" must not point to the same target directory. The same applies for "cloudsuite.data.user" and "cloudsuite.data.shared".

Example: Linux hosting Tomcat servlet container

#### Unix shell commands

```
sudo mkdir /var/lib/cloudsuite
sudo chgrp tomcat /var/lib/cloudsuite
sudo chmod 770 /var/lib/cloudsuite

sudo mkdir /var/log/cloudsuite
sudo chgrp tomcat /var/log/cloudsuite
sudo chmod 770 /var/log/cloudsuite
```

### 9.1.2 Built-in configuration

Sign Live! cloud suite gears is bootstrapped with built-in bean definitions that are contained in the WAR deployment.

There are two options to tweak this standard configuration.

First, some of the bean definitions (like the "dataSource", see below) are instrumented with Spring property definitions. This way you can fine-tune the bean using custom property definitions.

Second, bean definitions that are eligible for "overwriting" are always defined using a well-defined role name. You can create a bean definition with this role name in your custom bean definition to overwrite the system standard.

### 9.1.3 Custom property definition

By default, properties are defined with increasing priority from

- Built-in  
classpath:\${cloudsuite.config.name}.properties
- System level definition  
\${cloudsuite.config.shared}/\${cloudsuite.config.name}.properties
- User level definition  
\${cloudsuite.config.user}/\${cloudsuite.config.name}.properties

### 9.1.4 Custom bean definition

Custom bean definitions are read from the following locations, again with increasing priority:

- Built in  
classpath:spring-gears-core.xml  
classpath:modules/spring-gears-\*.xml
- System level  
\${cloudsuite.config.shared}/spring-gears-core.xml  
\${cloudsuite.config.shared}/modules/spring-gears-\*.xml
- User level  
\${cloudsuite.config.user}/spring-gears-core.xml  
\${cloudsuite.config.user}/modules/spring-gears-\*.xml

### 9.1.5 Using profiles

You can use Spring profiles to parameterize your configuration. A profile allows to bundle beans and properties and give them a meaningful name.

When starting the server, you can activate any of these profiles by defining

---

```
spring.profiles.active=<comma separated profile names>
```

---

This definition can be made in any of the well-known ways, either

- as environment variable
- as system property
- in your \${cloudsuite.config.name}.properties

To restrict a bean definition to a specific profile, you can add a section in your configuration like this

## Spring XML fragment

```
...
<beans profile="profilename">
...
</beans>
...
```

Any definition contained in the "<beans>" element will be used only if the corresponding profile is activated.

In addition, the application will read specific property files in addition to "\${cloudsuite.config.name}.properties" whose name match "\${cloudsuite.config.name}-<profilename>.properties".

The priority (increasing to the bottom) is

- System level definition
  - \${cloudsuite.config.shared}/\${cloudsuite.config.name}.properties
  - \${cloudsuite.config.shared}/\${cloudsuite.config.name}-<profilename>.properties, In the order of profile definition, first one highest
- User level definition
  - \${cloudsuite.config.user}/\${cloudsuite.config.name}.properties
  - \${cloudsuite.config.user}/\${cloudsuite.config.name}-<profilename>.properties, In the order of profile definition, first one highest

### 9.1.6 String expansion integration

The gears string expansion is integrated with the spring property definition to ease using custom properties at runtime.

You can use the prefix "config." for a spring property to make it visible in the "config" string expansion namespace.

For more information see the chapter "String expansion".

## 9.2 Web application root

In many production environments the web application container itself cannot know what is the fully qualified external URL for the services. This is most often caused by a reverse proxy.

While most of the time relative addresses are fine, sometimes the server needs to construct fully qualified URL, for example for redirect addresses.

While the server does it's best to derive what may be the URL from an external point of view by examining de-facto standard HTTP headers, there may be times this is not working.

In such special cases you can override the automatic detection by setting the root URL explicitly using the property

```
de.intarsys.application.rootUrl
```

These are the locations where we look up the property from lowest to highest precedence.

You can set the property in the web.xml

**servlet container web.xml**

```
<context-param>
  <param-name>de.intarsys.application.rootUrl</param-name>
  <param-value>https://my.server.com</param-value>
</context-param>
```

You can use an environment variable

```
DE_INTARSYS_APPLICATION_ROOTURL=https://my.server.com
```

You can set the property using a Java system property

```
-Dde.intarsys.application.rootUrl=https://my.server.com
```

## 9.3 Logging

With regard to logging cloud suite tries to come up with a default setup that can be used out of the box.

Internally, the Logback logging framework is used. The features and syntax of Logback are beyond the scope of this documentation. There are many resources available on the internet.

### 9.3.1 Override logging

When starting up, Logback is configured using the default "logback.xml", situated **anywhere** in a root package on the class path (or known from the Logback command line options). All standard Logback functionality is available. If you are happy with this and provide a configuration, you can skip the rest. As soon as cloud suite is aware of this "external" Logback configuration, it skips all further activities. You just have to be **sure** that you do not have any of the cloud suite context variables at your hand at this moment in time.

### 9.3.2 Built-in logging

If not overridden, at the earliest moment in the application lifecycle (in a Spring listener), we perform a relaunch of the Logback environment - this time with the well-known cloud suite variables established.

We then try to load a "\${cloudsuite.config.user}/logback.xml" and "\${cloudsuite.config.shared}/logback.xml", if this fails, we fall back to an internal default Logback configuration.

This configuration has two appenders, STDOUT and ASYNCFILE. ASYNCFILE will write all its output to the "\${cloudsuite.log.dir}" directory - a platform dependent location as stated above, using the log level "INFO". The encoding for the file is UTF-8.

The following Logback variables are available for your use within your logback.xml at this stage.

Logback variable	Definition
cloudsuite.config.shared	\${cloudsuite.config.shared}
cloudsuite.config.user	\${cloudsuite.config.user}
cloudsuite.data.shared	\${cloudsuite.data.shared}
cloudsuite.log.dir	\${cloudsuite.log.dir}
cloudsuite.log.level	\${cloudsuite.log.level}:INFO
config.shared	\${cloudsuite.config.shared}
config.user	\${cloudsuite.config.user}



data.shared	\${cloudsuite.data.shared}
log.dir	\${cloudsuite.log.dir}
log.level	\${cloudsuite.log.level}:INFO

### 9.3.3 Log correlation

gears provides some mechanisms to correlate log entries. Be aware that forwarding the correlation value in internal interactions (viewer, authentication pages,...) is not supported.

#### 9.3.3.1 Builtin correlation id

Builtin support is active for correlation of conversation related requests. The active conversation ids are pushed to a MDC named "corr". You can add this information to your log pattern using

```
%X{corr}
```

#### 9.3.3.2 3rd party integration

If you want to integrate 3<sup>rd</sup> party correlation information, you can configure a special "CorrelationFilter" that extracts information from standard HTTP requests.

CorrelationFilter properties

source	
string	One of <ul style="list-style-type: none"> <li>• header</li> <li>• parameter</li> <li>• cookie</li> </ul> to identify the source for the correlation value Default "parameter"
sourceKey	
string	The name of the property in the source to retrieve the correlation value Default "corr"
corrKey	
string	The name of the property in the MDC Default "corr"

#### Example

```
<bean id="customCorrelationFilter"
class="de.intarsys.tools.correlation.CorrelationFilter">
  <!--
  this is one of "parameter" | "header" | "cookie"
  -->
  <property name="source" value="parameter" />
  <property name="sourceKey" value="custom-corr" />
  <property name="corrKey" value="corr" />
</bean>

<bean class="org.springframework.boot.web.servlet.FilterRegistrationBean">
  <property name="name" value="customCorrelationFilter" />
  <property name="filter" ref="customCorrelationFilter" />
  <property name="order" value="1" />
</bean>
```

#### 9.3.3.3 gears to gears calls

For architectures where gears is directing its requests to another gears instance, forwarding the correlation id is provided by default.

This means that the originating gears instance creates its builtin correlation id (see above). When it comes to forwarding the request, the id is included via the "corr" URL parameter

and collected by the receiving instance using the above mentioned preconfigured filter. The local builtin correlation id of the receiving instance is concatenated with the calling one.

### 9.3.4 Log tweaks

If you don't want to provide a complete logback.xml configuration yourself, you can use one of the built-in configurations:

- `<default>`
- `console`

In order to select a configuration, you can set a property like this:

```
cloudsuite.log.config=console
```

#### 9.3.4.1 Default built-in configuration

The default configuration registers a file and a console appender. To activate it, just omit the corresponding property. You can try to use the following configuration hot-spots that are built-in into it.

#### 9.3.4.2 Directory

If you simply want to set another target directory, you can just set a property like this. This will be forwarded to the built-in log definition.

```
cloudsuite.log.directory=/srv/logs
```

#### 9.3.4.3 Level

If you simply want to set another logging level, you can just set a property like this. This will be forwarded to the built-in log definition.

```
cloudsuite.log.level=DEBUG
```

#### 9.3.4.4 Built-in configuration 'console'

The built-in configuration 'console' only enables console output for messages. This is particularly suitable for environments where file access is not available and logs are aggregated from the console output, e.g. on Cloud Foundry. You can activate it by setting the property like this:

```
cloudsuite.log.config=console
```

## 9.4 Licenses

The product requires valid licenses for execution.

Licenses are obtained from intarsys, a limited demo license is included with the product for basic usage.

You can provide new licenses by copying the license files either to

```
${cloudsuite.config.shared}/licenses
```

or

```
${cloudsuite.config.user}/licenses
```

Upon startup, all licenses that have been picked up are logged to the log file.

```
[04.05.2018-10:36:18.622][I][d.i.tools.license ][localhost-startStop-1][ ] loading
licenses from 'C:\ProgramData\cloudsuite\config'
[04.05.2018-10:36:18.637][D][d.i.s.d.pool.device ][rEnvironment service][ ] signature
pool launcher started
[04.05.2018-10:36:18.653][I][d.i.tools.license ][localhost-startStop-1][ ] license
'de.intarsys.cloudsuite.product.gears; 8; intarsys.de; 12/01/2017; 06/30/2018;
id=de.intarsys.cloudsuite.product.gears; account_automation_cli=-1:day;
account_automation_batch=-1:day; bundle=professional; batchSize=-1;
account_automation_api=-1:day; ' loaded
[04.05.2018-10:36:18.653][I][d.i.tools.license ][localhost-startStop-1][ ] license
'de.intarsys.security.device.common; 8; intarsys.de; 12/01/2017; 06/30/2018;
id=de.intarsys.security.device.common; de.intarsys.security.app.sign.account=-1:day;
de.intarsys.security.app.decrypt.account=-1:day; ' loaded
[04.05.2018-10:36:18.653][I][d.i.tools.license ][localhost-startStop-1][ ] license
'de.intarsys.security.device.bridge; 8; intarsys.de; 12/01/2017; 06/30/2018;
id=de.intarsys.security.device.bridge; de.intarsys.security.app.sign.account=-1:day;
de.intarsys.security.app.decrypt.account=-1:day; ' loaded
```

## 9.5 Data source

There is a single data source that is shared by the standard product components (like auditing).

The default configuration uses an apache data pool on a JDBC data source.

You can override the JDBC settings in your custom property definition (see example below)

### Spring properties

```
jdbc.driverClassName=org.h2.Driver
jdbc.url=jdbc:h2:${cloudsuite.data.shared}/db/gears;AUTO_SERVER=TRUE
jdbc.username=user
jdbc.password=password
```

or you can override the complete bean by providing a "dataSource" bean in your custom bean definition (see example below)

### Spring XML fragment

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
  <property name="initialSize" value="3" />
  <property name="defaultAutoCommit" value="false" />
  <property name="maxTotal" value="10" />
  <property name="poolPreparedStatements" value="true" />
</bean>
```

## 9.6 Conversation registry

The conversation registry is the internal "session manager" for the asynchronous flows. It is defined as a Spring bean and can be overridden in your custom Spring configuration.

With the registry, you can configure the maximum idle lifetime of a conversation and the conversation cleanup interval.

Example

## Spring XML fragment

```
<bean
  id="conversationRegistry"
  class="de.intarsys.tools.conversation.impl.StandardConversationRegistry">
  <!-- cleanup every minute -->
  <property name="cleanupInterval" value="60000"/>
  <!-- 10 minute idle time -->
  <property name="expireTimeout" value="600000"/>
</bean>
```

The **expireTimeout** will define how long a conversation will remain valid when no interaction is detected. You can compare this to the classic "session timeout" with a web application. If for example the user opens a viewer, in the above configuration after 10 minutes of inactivity the conversation will be discarded. The default is 30 minutes.

## 9.7 Basic security

The default configuration provides the application with a bunch of basic security tools.

To fully understand the encryption settings, be sure to read chapter "Crypto components" in advance.

### 9.7.1 Master key

The system requires a "master key" (an `IByteProvider`) bean "cryptoKeyMaster".

By default, this is a byte sequence derived from some password input using the PBKDF2WithHmacSHA1 algorithm. The password input is expected via Java system properties (\*not\* Spring properties), either

- `cloudsuite.crypto.key.password.hex`  
The password input in hex notation
- `cloudsuite.crypto.key.password.text`  
The password input as plain text. The text is converted to bytes using UTF-8 encoding.
- `cloudsuite.crypto.key.password.file`  
The password is contained in the specified file (in binary format)

As a default, a hard-coded password is contained in the definition. Do not retain this for production use.

You can redefine the "cryptoKeyMaster" byte provider to collect the master key from wherever you want.

### 9.7.2 Master KDF

You should never use the master key directly. Instead, for each component you should derive another key. To achieve this, by default a key derivation function "cryptoKdfMaster" is defined, providing key derivation based on RFC-5869.

You can redefine this bean with your own KDF implementation.

### 9.7.3 Application "cryptdec"

For persisting certain artifacts in the application (like cached secrets), an internal encryption is set up, defined in the "cryptoCryptdec" bean. This defines a dynamic, AES 128 based de/encryption, where the key is derived from the "cryptoKdfMaster".

## 9.8 Repository

The repository is defined as a Spring bean and can be overridden in your custom Spring configuration.

### 9.8.1 Basic settings

For the repository you need to configure the "repository" bean. By default, this is the **StandardRepository** implementation. That delegates much of its implementation to a "DAO".

---

#### Spring XML fragment

```
<bean id="repository"
  class="de.intarsys.cloudsuite.gears.repository.standard.StandardRepository">
  <property name="repositoryDao" ref="repositoryDao" />
</bean>
```

---

The DAO for example determines where the repository resides on the file system, the encryption mechanics and so on.

---

#### Spring XML fragment

```
<bean id="repositoryDao"
  class="de.intarsys.cloudsuite.gears.repository.fs.FSRepositoryDao">
  <property name="baseDir" value="${cloudsuite.data.shared}" />
</bean>
```

---

The **baseDir** property determines where to put the "repo" folder, containing all of the repository state.

The "repo" folder will be completely deleted when the application starts.

### 9.8.2 Encryption

With the basic security properly set up, you can configure enhanced repository security.

A complete example for this setup is included in the "demo/repository-encryption" folder of your installation. The definitions are guarded by the Spring profile "encrypt", so be sure to add this to your active profiles if you want to use it literally.

To prepare for repository encryption, it would be best practice to start with a separate "key subtree" derived from the master key, so we will derive from the **"cryptoKdfMaster"**:

---

#### Spring XML fragment

```
<bean id="cryptoKdfRepo"
  class="de.intarsys.tools.crypto.kdf.WithPrefixKeyDerivationFunction">
  <property name="kdf" ref="cryptoKdfMaster" />
  <property name="prefixProvider">
    <bean class="de.intarsys.tools.crypto.bytes.StaticByteProvider">
      <property name="text" value="repository.encryption" />
    </bean>
  </property>
</bean>
```

---

We start a new key subtree for "repository.encryption" here.

The standard repository encryption uses a two-stage approach - first the content encryption using completely random key material, then key-wrapping using keys derived according to the configuration.

To enable encryption, the property "contentCipherFactory" of the DAO must be instrumented with an "ICipherFactory". Concretely there is a generic factory for accessing JCA supported algorithms.

---

#### Spring XML fragment

---

```
...
<property name="contentCipherFactory">
  <bean class="de.intarsys.tools.crypto.standard.JcaCipherFactory">
    <property
      name="encryptionAlgorithmTransformation"
      value="AES/CTR/NoPadding"/>
    </bean>
  </property>
</property>
...
```

---

This component creates ciphers for content encryption. The supported transformations for the content are

- AES/CTR/NoPadding

For each document, a new cipher with random IV and random key is created by this DAO implementation (the content encryption key, "cek", is random and is not configurable).

This key is then encrypted using the "keyWrapperFactory" of the DAO. Key wrapping can be configured with another ICipherFactory. The supported transformations for key wrapping are:

- AESWrap

Most useful scenarios for the repository will be:

- Use a static key for key encryption, derived from the system master key.

In this case we simply base on the "**cryptoKdfRepo**" key subtree defined in the beginning, deriving for the key "static" and inject the resulting key in a StaticKeyCipherFactory.

---

#### Spring XML fragment

---

```
<property name="keyWrapperFactory">
  <bean class="de.intarsys.tools.crypto.standard.StaticKeyCipherFactory">
    <property name="cipherFactory">
      <bean class="de.intarsys.tools.crypto.standard.JcaCipherFactory">
        <property name="encryptionAlgorithmTransformation" value="AESWrap" />
      </bean>
    </property>
  </bean>
  <property name="keyProvider">
    <bean class="de.intarsys.tools.crypto.bytes.DerivedByteProvider">
      <property name="kdf" ref="cryptoKdfRepo" />
      <property name="inputProvider">
        <bean class="de.intarsys.tools.crypto.bytes.StaticByteProvider">
          <property name="text" value="static" />
        </bean>
      </property>
    </bean>
  </property>
</bean>
</property>
</property>
</property>
```

---

- Use a dynamic, "per user" key for key encryption, derived from the system master key and some identification input from the client.

We start from the "kdfRepo", too, but we need no additional configuration as the repository provides the "per user" key by default.

**Spring XML fragment**

```
<property name="keyWrapperFactory">
  <bean class="de.intarsys.tools.crypto.standard.DerivedKeyCipherFactory">
    <property name="cipherFactory" ref="repositoryCipherKey" />
    <property name="kdf" ref="cryptoKdfRepo" />
  </bean>
</property>
```

In each case, to recover the key, the DAO stores an "EncryptionInfo" object along with the document meta information, the structure is comparable to CMS.

## 9.9 PDF security environment

Creating PDF signatures or timestamps requires an intermediate, transient document representation.

You can configure how this representation is handled. It is defined as a Spring bean and can be overridden in your custom Spring configuration.

The "PlainTransientLocatorFactory" will create an in-memory buffer up to a specified size. If the document gets larger, a plain document is created in the temp directory.

**Spring XML fragment**

```
<bean
  class="de.intarsys.security.document.type.pdf.common.PlainTransientLocatorFactory">
  <property name="maxBufferSize" value="1000000" />
</bean>
```

The "EncryptedTransientLocatorFactory" will create an in-memory buffer up to a specified size. If the document gets larger, an AES128 encrypted document is created in the temp directory. Be aware that this will reduce the performance for large files.

**Spring XML fragment**

```
<bean
  class="de.intarsys.cloudsuite.gears.document.type.pdf.\
EncryptedTransientLocatorFactory">
  <property name="maxBufferSize" value="1000000" />
</bean>
```



## 9.10 Principals

### 9.10.1 Overview

Sign Live! cloud suite gears has a generic concept of "principals" - entities that in some way provide context to or control the service execution.

Typical influence of a principal on the service execution or outcome:

- The user principal provides claims that are used in string expansion
- The user principal defines a profile whose properties enhance service arguments
- The client principal is billed for the service execution

**You don't have to deal with principals – if you ask yourself what may be the use of it, then you may safely ignore it.**

By default, gears deals with three roles for contextual principals.

- A "tenant" represents a customer or organizational unit  
(urn:intarsys:names:principal:1.0:role:Tenant)
- A "client" represents an application  
(urn:intarsys:names:principal:1.0:role:Client)
- A "user" represents a single entity within the client world, eventually holding a private key  
(urn:intarsys:names:principal:1.0:role:User)

There is no hardcoded use of these roles, a principal can be configured so that it is under your control

- which principals do exist
- how principals are derived
- where principals are used

### 9.10.2 Principal model

#### 9.10.2.1 Overview

Principals are "model" objects and as such have an internal representation and a data access strategy.

A generic internal implementation is available.

The data access strategy (or short DAO for data access object) determines the way principals are derived and looked up when given an id. Examples are in-memory maps of principals or a database lookup.

#### 9.10.2.2 Structure

These implementation objects can be used when creating principal objects literally in the configuration (see upcoming examples)

```
de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal
de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim
```

#### 9.10.2.3 In-Memory DAO

The in-memory representation allows you to literally define principals and claims in the configuration.

This can be used conveniently for small scale applications with only a few tenants, clients or users.

The in-memory representation is implemented using the

---

```
de.intarsys.cloudsuite.gears.model.entity.principal.PojoPrincipalDao
```

---

You can use this class to configure all available principals. The bean id used internally for this object is "modelPrincipalDao".

Spring example for in-memory configuration

---

### Spring XML fragment

---

```
<bean id="modelPrincipalDao"
class="de.intarsys.cloudsuite.gears.model.entity.principal.PojoPrincipalDao">
  <property name="items">
    <list>
      <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
        <property name="name" value="foo" />
        <property name="claims">
          <list>
            <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
              <property name="key" value="claim1" />
              <property name="value" value="value1" />
            </bean>
            <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
              <property name="key" value="claim2" />
              <property name="value" value="value2" />
            </bean>
          </list>
        </property>
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
        <property name="name" value="E=support@intarsys.de,CN=gears demo client
ssl,O=intarsys GmbH" />
        <property name="claims">
          <list>
            <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
              <property name="key" value="claimX" />
              <property name="value" value="valueX" />
            </bean>
            <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
              <property name="key" value="claimY" />
              <property name="value" value="valueY" />
            </bean>
          </list>
        </property>
      </bean>
    </list>
  </property>
</bean>
```

---

#### 9.10.2.4 Literal DAO

You can use this DAO if your principal data is transmitted literally. In combination with the `ExplicitPrincipalProvider` you can send a principal in the request options like here:

Request example

---

### service call

---

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

{
  "options": {
    "principal": "foo"
  },
  ...
}
```

---

Request example

## service call

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

```
{
  "options": {
    "principal": {
      "name": "foo",
      "claims": {
        "gnu": "gnat"
      }
    }
  },
  ...
}
```

The fully qualified name of the DAO implementation is

```
de.intarsys.cloudsuite.gears.model.entity.principal.LiteralPrincipalDao
```

This DAO simply takes its lookup argument and creates a principal with that name if it is a string. If it receives a complete object it tries to deserialize "name" and "claims" from that object.

Spring configuration example

## Spring XML fragment

```
<bean
  id="modelPrincipalDao"
  class="de.intarsys.cloudsuite.gears.model.entity.principal.LiteralPrincipalDao"
>
</bean>
```

This is the **default** configuration for principal lookup in gears.

## 9.10.2.5 Jdbc DAO

Use a JDBC DAO if the principal assets are stored in a database already.

We provide a generic implementation for accessing a database with

```
de.intarsys.cloudsuite.gears.model.entity.principal.JdbcPrincipalDao
```

The DAO needs two bean properties, the "dataSource", which is a JDBC DataSource object and "sql", a standard JDBC compatible SQL expression. The sql expression must evaluate to columns

- name (the principal name)
- key (a claim key)
- value (a claim value)

Spring configuration example

## Spring XML fragment

```
<bean id="modelPrincipalDao"
class="de.intarsys.cloudsuite.gears.model.entity.principal.JdbcPrincipalDao">
  <property name="dataSource" ref="dataSource" />
  <property name="sql" value="select Principal.name, Claim.key, Claim.value from
Principal inner join Claim on Principal.name = Claim.name where Principal.name=?" />
</bean>
```

This will read from tables "Principal" and "Claim" the required name, key and value fields.

#### 9.10.2.6 Other DAO's

If you provide the principal information from other sources, other DAO implementations can be created with little overhead.

Examples of such sources may be

- LDAP
- Web Services
- Local files

### 9.10.3 Roles

#### 9.10.3.1 Tenant principal

The tenant principal is a "container", grouping clients and/or users. It can be used with different modules for tasks like authentication, billing, providing templates etc.

The well-known role associated with the tenant principal is

```
urn:intarsys:names:principal:1.0:role:Tenant
```

#### 9.10.3.2 Client principal

The client principal is a "container", contained in a tenant, grouping possible users. You can use this to fine tune tenant specific context information.

The well-known role associated with the client principal is

```
urn:intarsys:names:principal:1.0:role:Client
```

#### 9.10.3.3 User principal

The user principal identifies the user that executes (or on whose behalf is executed) the current request.

The well-known role associated with the user principal is

```
urn:intarsys:names:principal:1.0:role:User
```

#### 9.10.3.4 Other roles

You can define additional principal roles in your configuration, if required.

### 9.10.4 Provider

The "principal provider" implements the way that principals are provided at runtime. For each role required, a provider is defined.

This chapter describes the available options for creating principals in your installation.

#### 9.10.4.1 Static provider

The static provider simply defines all principal characteristics in the configuration itself. To do so, it uses a generic principal model implementation, `GenericPrincipal` and `GenericClaim` (see section "Reference").

This is typically used for the "tenant" or "client" role.

Spring example for a statically injected principal:

#### Spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.model.entity.principal.StaticPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:Tenant" />
  <property name="principal">
    <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
      <property name="name" value="tenant" />
    </bean>
  </property>
</bean>
```

#### 9.10.4.2 Explicit provider

The explicit provider associates the principal from a concrete service request with the given role. This allows an (unauthenticated) explicit principal definition from the client. The principal name (or the complete principal structure) is provided by the client in the request using the service request option "principal".

Request example

#### service call

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

{
  "options": {
    "principal": {
      "name": "foo",
      "claims": {
        "gnu": "gnat"
      }
    }
  },
  ...
}
```

The principal lookup is delegated to a principal DAO (see above).

This is typically used for the role "user". You can opt to look up the principal claims in a database or send it completely in the request.

Spring example for an explicit injected principal:

---

**Spring XML fragment**

---

```
<bean class=
    "de.intarsys.cloudsuite.gears.model.entity.principal.ExplicitPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="principalDao" ref="modelPrincipalDao"/>
</bean>
```

---

#### 9.10.4.3 Spring security provider

The spring security provider associates a principal that was previously authenticated with a given role. This allows for an (authenticated) per-request contextual principal.

This is typically used for the "client" or "user" role.

The authentication mechanism itself is completely orthogonal and described in chapter "Authentication".

Spring example for "authenticated principal injection":

---

**Spring XML fragment**

---

```
<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="keyConverter">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.\
AuthenticationToStringConverter"/>
  </property>
  <property name="principalDao" ref="modelPrincipalDao" />
</bean>
```

---

#### 9.10.5 Default configuration

The default configuration holds three principal providers, one for "tenant", "client" and "user".

The respective definitions are

---

**Spring XML fragment**

---

```
<bean
  id="principalProviderTenant"
  class="de.intarsys.cloudsuite.gears.model.entity.principal.StaticPrincipalProvider">

  <property name="role" value="urn:intarsys:names:principal:1.0:role:Tenant" />
  <property name="principal">
    <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
      <property name="name" value="tenant" />
    </bean>
  </property>
</bean>

<bean
  id="principalProviderClient"
  class="de.intarsys.cloudsuite.gears.model.entity.principal.StaticPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:Client" />
  <property name="principal">
    <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
      <property name="name" value="client" />
    </bean>
  </property>
</bean>

<bean id="principalProviderUser"
  class="de.intarsys.cloudsuite.gears.model.entity.principal.ExplicitPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="principalDao" ref="modelPrincipalDao" />
</bean>

<bean id="modelPrincipalDao"
class="de.intarsys.cloudsuite.gears.model.entity.principal.LiteralPrincipalDao">
</bean>
```

To overwrite these definitions, you include a bean definition with the respective name "principalProviderTenant", "principalProviderClient" or "principalProviderUser" in your own configuration.

## 9.11 Services

All service configuration is done via Spring beans and can be overridden in your custom Spring configuration.

### 9.11.1 Signer

#### 9.11.1.1 Static configuration

"static configuration" here denotes beans and properties that are supplied by an administrator for the gears application and apply to all runtime features.

There is currently no static configuration for the signer required.

#### 9.11.1.2 Flow Configurations

A **signer** is using a "FlowSignerConfiguration" to define the context for the service to be processed. For details on a Configuration structure see chapter Configuration. Note that not all configuration properties are supported or make sense for a signer (e.g. properties describing UI elements).

The concrete configuration is passed in the "configuration" parameter when creating a signer. This can be a reference, a literal configuration or an array thereof.

The most useful property for defining a signer configuration is the "definitions" property. This allows to define a template for a complete signer call. This way you can avoid to scatter signature related details throughout your application.

### 9.11.2 Viewer

#### 9.11.2.1 Flow configuration

You can provide multiple configurations for a viewer for use in different scenarios.

The concrete configuration is passed in the "configuration" parameter when creating a signer. This can be a reference, a literal configuration or an array thereof.

For example, the gears demo will provide a "plain" configuration, where only the basic document viewing features are present and a "demoSignature" configuration, where you have a selection of three buttons for different signature types.

You can define your own configuration where you can provide only buttons appropriate for a dedicated use case. You do this in your custom bean definition or literally in the call to viewer/create.

Be aware that this definition is a quite advanced and complex topic, involving interpretation both in the client and the server, both at startup and execution runtime. The complete documentation is in the section "Reference".



## Spring XML fragment

```

<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="demoCustom" />
  <property name="widgets">
    <list>
      <w:widget parent="de.intarsys.widget.toolbar.additions"
        label="{nlsmsg.de.intarsys.gears.core.demo.messages#btnSignFixed.label}"
        icon="pencil">
        <w:on event="select" do="Sign">
          <entry key="requireField" value="false" />
          <entry key="requireSignature" value="false" />
          <entry key="field" value="x=355;y=165;width=60;height=50" />
          <entry key="pageRange" value="last" />
          <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:on>
      </w:widget>
    </list>
  </property>
</bean>

```

This is a template for creating your own signature button.

We need a bean of type "FlowViewerConfiguration" with an "id" property. This bean is registered in the server and can be requested in the viewer creation service.

To add a custom button we add a new widget as a child of the "de.intarsys.widget.toolbar.additions" widget. You should add a "label" attribute and an "icon" property.

The widget is then associated with an action, here triggered by the "select" event of the button. The action is a predefined "SignerAction".

gears will execute the action according to the action properties (e.g. "requireField" and "requireSignature") and allows you even to merge the arguments to the "signerCreate" flow with arguments you have provided with the viewer call.

For details of the available action definitions see the respective reference chapters.

### 9.11.2.2 PDF vs PDF/A

The PDF syntax and semantics is an ISO standard (ISO 32000-x), defining encoding and rendering of electronic documents. Being a de-facto "silver bullet" in nearly every business document workflow and exchange process, its biggest drawback is the extremely wide range of possibilities and its dependence on external font resources.

The PDF/A set of standards comes to the rescue to restrict plain PDF to a subset that is self-contained and guaranteed to be long-term usable. This is the reason you should always use PDF documents compliant to one of the PDF/A conformance levels.

For this product, only PDF/A compliant documents are officially supported.

That said, most of non-PDF/A documents should render without problems out of the box. Here, we present some details how to proceed if a document cannot be rendered properly.

For the viewer component there are two main areas of interest regarding rendering non-PDF/A documents.

### 9.11.2.3 Dynamic content

First, as a "trusted viewer", we do not render any (well, most) of the dynamic content that may be contained in a PDF. In theory, you may include JavaScript that modifies the appearance of the document - something you do not really want to happen when signing

a contract. There's nothing you can do in the built-in rendering to change this rendering queue. The best idea would be to preprocess the document and "flatten" it, means creating a new PDF document where the dynamic rendering is executed once and a new static PDF is created. Using a PDF/A converter will do exactly this.

#### 9.11.2.4 Font handling

Second, rendering documents without embedded fonts is something that is near to black arts. The outcome is dependent on the rendering platform context, the "correct" resolution of a font is only a heuristic.

The way font resolution works is as follows:

1. Lookup an embedded font in the document
2. If lookup fails, register all user defined fonts
3. Lookup in font registry
4. If lookup fails, register all system fonts
5. Lookup in font registry
6. If lookup fails, resume with 1, looking for a "regular" version of the font
7. If lookup fails, resume with 1. looking for a version of the font implemented in another font program (Type1 vs TrueType)
8. If lookup fails, try to replace with "Arial" if a TrueType font was searched
9. If lookup fails, try to replace with "Helvetica" if a Type1 font was searched.

We see, the easiest way to provide a font that is not embedded is importing it into the platform specific font directory.

#### 9.11.3 Explorer

You can provide multiple configurations for an explorer for use in different scenarios.

The concrete configuration is passed in the "configuration" parameter when creating a signer. This can be a reference, a literal configuration or an array thereof.

For example, the default implementation will provide a "demoSignature" configuration, where you have an additional button for batch signing the documents.

You can define your own configuration where you can provide only buttons appropriate for a dedicated use case. You do this in your custom bean definition.

The definition is similar to the viewer configuration above.

#### 9.11.4 Timestamp

##### 9.11.4.1 Flow Configurations

Similar to the signer, a **timestamp** is using a "FlowTimestampConfiguration" to define the context for the service to be processed. For details on a Configuration structure see chapter Configuration. Note that not all configuration properties are supported or make sense for a timestamp (e.g. properties describing UI elements).

The concrete configuration is passed in the "configuration" parameter when creating a timestamp. This can be a reference, a literal configuration or an array thereof.

### 9.12 Signature environment

Some services leverage environmental configuration to execute signature-related processes. This configuration is given by beans and properties that are supplied by an administrator for the gears application and apply to all runtime features.

### 9.12.1 Timestamp creation

Processes like the creation of longterm-validatable signatures require the addition of timestamps. While the timestamp service to use can be determined per request, it is possible to configure a default timestamp service for fallback use.

The following properties are supported:

signatureEnvironment.defaultTimestampDevice	
string	The name of the timestamp device to use by default.  Default: default@tsa

### 9.12.2 Signature validation

Some processes and utilities may require the execution of signature validation processes. Examples are the creation of LT-level signatures or displaying the signature integrity within the viewer's SignaturesSidebar component.

Be aware that signature validation requires HTTP and HTTPS access to external resources. In exceptional cases, outgoing LDAP connectivity may be necessary in order to obtain certificate revocation lists.

The following properties are supported:

digsig.validation.validationContext.qualificationSeverity	
string	Identifier determining how non-qualified signatures are being handled. Must be one of: error   warning   info  The meaning is as follows: <ul style="list-style-type: none"> <li>error: Reject the signature if it is not qualified.</li> <li>warning: Show a warning if the signature is not qualified.</li> <li>info: Accept an otherwise valid signature, even if it is not qualified.</li> </ul> Default: error

### 9.12.3 Trusted list management

The signature validation algorithm heavily relies on trust anchors defined in trusted list. While a current set of trusted lists is shipped with the application, certificates being registered and used are ever-changing, so that these lists need to be updated from time to time. To ensure you're always up-to-date, you can schedule an automatic trusted list update, using a cron expression which triggers the update at a time which best suits your environment's constraints.

Be aware that updating trusted lists requires HTTP and HTTPS access to external resources.

The following properties are supported:

trustedLists.update.cron	
string	A Spring cron expression defining the time trigger for automatic trusted list update. Details on how to define such expressions are given in <a href="https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/support/CronExpression.html">https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/support/CronExpression.html</a> Example: <pre># update trusted list every Sunday at midnight trustedLists.update.cron=0 0 0 * * SUN</pre> If omitted, no automatic update is performed.

## 9.13 Load Balancing

As described in [2], gears has built-in support for load balancing.

This enables seamless, sticky addressing of the correct gears instance through all involved conversation participants.

Load balancing can be simply enabled by adding the “loadbalance” profile to the gears.properties.

Example

```
spring.profiles.active=demo,loadbalance
```

This will enable the following

- On every outgoing HTTP response a cookie “sticky\_id” is added that uniquely identifies the gears instance. This cookie is for load balancer use only. If you rely on the cookie in your application/configuration you may lose the ability to have more than a single conversation active.
- On every URL that is emitted and that finally involves gears a query parameter “sticky\_id” is automatically added (e.g. the viewer redirect)
- The “\_links” section in the non final reply stages is filled with fully instrumented endpoint URLs. With correct load balancer setup these will address the correct gears instance.

The only (but important) thing your client application must be aware of is the decoration of every conversation related request with the “sticky\_id” query parameter. If you choose to use the indirection provided in the “\_links” section, you are ready. This is the recommended approach as it is “future proof” when more bells and whistles are added to the gears protocol.

### 9.13.1 Additional gears configuration

If the default behavior is not matching your needs there are some properties you can tweak.

loadbalance.cookieName	
string	<p>The name of the cookie / query parameter that is added to the protocol elements.</p> <p>Do not change this without need as not all protocol participants play nicely with generic field names</p>
loadbalance.cookieValue	
string	<p>The value of the sticky_id property that is published.</p> <p>By default this is a random UUID for each server.</p>

### 9.13.2 Load balancer configuration

A typical load balancer configuration will implement the following behavior

- On request, lookup the sticky\_id query parameter and select the corresponding server from the map
- If no sticky\_id is present, take a server that matches your load balancing strategy
- On response, extract the “sticky\_id” cookie and store for later retrieval of the corresponding server

That’s it.

This process is chosen because it does not need any payload inspection or deep protocol knowledge. The “hard work” is done by gears by decorating the URLs with the correct `sticky_id`.

A simple example configuration for the well-known HAProxy shows the concept:

```
global

    # The global section defines the global logging to standard out using the default
    # format
    log stdout format raw local0
    # Uncomment the statement below to turn on verbose logging
    # debug

defaults

    # haproxy works in http mode (layer 7)
    mode http
    # redirects all logging to the global log definition and activates http based log
    log global
    option httplog
    # The default timeout is infinite, so set up some useful constraints.
    timeout connect 10s
    timeout client 1m
    timeout server 1m
    # If sending a request to one server fails, try to send it to another, 3 times
    # before aborting the request
    retries 3

frontend http-in

    # make gears aware that it is behind a proxy
    option forwardfor
    # bind to port 80 on any interface
    bind *:80
    # send all requests to this backend
    default_backend gears_core

backend gears_core

    # use all servers in turn
    balance roundrobin

    # we need a stick table that can handle the gears sticky_id
    stick-table type string len 64 size 5M expire 30m
    # matching sticky_id in query
    stick match url param(sticky id)
    # we must fill the table with the cookie returned by gears
    stick store-response res.cook(sticky_id)

    # the servers are hosted on the container machine, check is made for availability
    server server0 host.docker.internal:8090 check
    server server1 host.docker.internal:8091 check
    server server2 host.docker.internal:8092 check
```

### 9.13.3 Bridge integration

Gears load balancing includes the bridge agent, even so it runs in a completely different process space.

To take advantage of the integration, you must use at least bridge version 7.1.11.

## 9.14 String expansion

Some namespaces supported by the string expansion grant access to sensitive data like secrets or the service’s environment variables. By default, expressions in these sensitive namespaces are only resolved for trusted data. The following two configuration properties can be used to selectively lift this restriction so that such expressions are also resolved in untrusted data like user input or request arguments:

- `gears.core.expressions.untrusted.allow`
- `gears.core.expressions.untrusted.block`

The values of these properties are Java-flavored regular expressions. Sensitive expressions that match the allow-expression, but not the block-expression are also resolved for untrusted data.

For example, the following properties allow the resolution of the environment variable `PORT` and all myservice-related configuration properties except `properties.myservice.password`:

```
gears.core.expressions.untrusted.allow=environment\\.PORT|properties\\.myservice\\.\\..*
gears.core.expressions.untrusted.block=properties\\.myservice\\.\\.password
```

**NOTE** In a regular expression, the dot “.” is a wildcard that matches any character. If you want to match a literal dot, you should precede it with a backslash to ignore its special meaning. Unfortunately, in property files, the backslash itself has a special meaning and therefore has to be escaped as “\\”. In sum, a literal dot in a regular expression in a property file is represented by “\\.”.

**NOTE** The configuration properties above should be used carefully to prevent any leak of sensitive information!

## 10. UI definition

---

### 10.1 Overview

Some application components have a customizable user interface.

The building blocks for customization are the "widget" and "action" definitions. A "widget" defines the UI structure, an "action" defines the application behavior. You can find a detailed reference for this in section "Reference". Here we go through the topic in an informal manner.

An example may be the "Sign" action. Its behavior upon execution on the server side is to call the gears endpoint "signer/create" and create a signer flow. It provides properties that define the interaction with the user, e.g., whether the interaction is allowed to create a visible field and properties that directly act as input to the signer flow.

This "Sign" action can be plugged into a "Viewer" flow and mapped to a toolbar button using a widget.

This is the basic pattern that is used throughout gears to allow for customization of user interface layout and behavior.

### 10.2 Widget definition

#### 10.2.1 Structure

The widget defines the UI control element with which the user interacts to initiate actions in the application.

A widget can be anything from a button to a complete viewer that itself is a container for other widgets. Widgets are arranged hierarchically, though.

The widget definitions are requested from the UI component upon start, the component configures its layout then accordingly.

For a complete reference, see chapter WidgetSpec.

Let's start with a simple example

## Spring XML fragment

```
<w:widget parent="de.intarsys.widget.toolbar.additions" id="sign" label="Signiere">
  <w:on event="select" do="Sign">
    <entry key="requireField" value="false" />
    <entry key="requireSignature" value="false" />
    <entry key="signerCreate.configuration" value="myConfiguration" />
  </w:on>
</w:widget>
```

## JSON fragment

```
{
  "id": "sign",
  "label": "Signiere",
  "parent": "de.intarsys.widget.toolbar.additions",
  "callbacks": {
    "select": {
      "factory": "Sign",
      "args": {
        "requireField": false,
        "requireSignature": false,
        "signerCreate.configuration": "myConfiguration"
      }
    }
  }
}
```

The widget **type** defines the type of control element that is created – but this is derived from the context very often, e.g. a widget in a toolbar is by default a button (like in the example above).

The widget **id** defines its identity and allows direct access for different use cases, e.g. defining a hierarchy, switching widget properties or programmatically triggering events.

The widget **parent** allows to define the hierarchy and the role of the widget in the UI. The predefined widget extension points are documented in the reference section.

The **label**, **tip** and **icon** properties define aspects of the visual appearance of the widget (if supported by the type).

Besides these properties that are available as XML attributes, a widget can possess any set of additional properties. These can be defined using the "w:property" element:

## Spring XML fragment

```
<w:widget id="de.intarsys.widget.toolbar">
  <w:property name="visible" value="false" />
</w:widget>
```

## JSON fragment

```
{
  "id": "de.intarsys.widget.toolbar",
  "properties": {
    "visible": false
  }
}
```

The **properties** supported depend on the widget type.

Another important feature of widgets is the hierarchy definition. You can either nest widgets simply – this is not very common and so this example is rather advanced:



## Spring XML fragment

```

<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotations"
type="AnnotationsOverlay">
  <w:widget id="canvas" type="switch">
    <w:widget type="case">
      <w:property name="type" value="Widget" />
      <w:property name="subType" value="Sig" />
      <w:property name="signed" value="false" />
      <w:on event="select" do="Sign" />
    </w:widget>
  </w:widget>
<w:widget id="popup">
  <w:widget label="Sign..">
    <w:property name="type" value="Widget" />
    <w:property name="subType" value="Sig" />
    <w:property name="signed" value="false" />
    <w:on event="select do="Sign" />
  </w:widget>
  ...
...

```

## JSON fragment

```

{
  "parent": "de.intarsys.widget.renderer.overlays",
  "id": "annotations",
  "type": "AnnotationsOverlay",
  "children": [
    {
      "id": "canvas",
      "type": "switch",
      "children": [
        {
          "type": "case",
          "properties": {
            "type": "Widget",
            "subType": "Sig",
            "signed": false
          },
          "callbacks": {
            "select": {
              "factory": "Sign"
            }
          }
        }
      ]
    },
    {
      "id": "popup",
      "children": [
        {
          "label": "Sign...",
          "properties": {
            "type": "Widget",
            "subType": "Sig",
            "signed": false
          },
          "callbacks": {
            "select": {
              "factory": "Sign"
            }
          }
        }
      ]
    }
  ]
}

```

A hierarchy definition that is used more often operates by referencing the parent in the **parent** attribute like in the first example.

## 10.2.2 Icon

The **icon** property is particularly interesting because of the mechanics that are supported behind the scene.

The use of icons in the application is based on fontawesome, a well-known icon library. With the advent of plugins (see [3]) you have access to all of over 7000 available icons (see <https://fontawesome.com/icons?d=gallery>).

To address the icons properly you must provide the icon name in the form

```
prefix:name
```

with the prefix one of the fontawesome icon set prefixes

- far
- fas
- fal
- fab
- fad

e.g.

### Spring XML fragment

```
<w:widget parent="de.intarsys.widget.toolbar.additions" icon="fas:angry">
  <w:on event="select" do="Alert">
    <entry key="message" value="hi" />
  </w:on>
</w:widget>
```

### JSON fragment

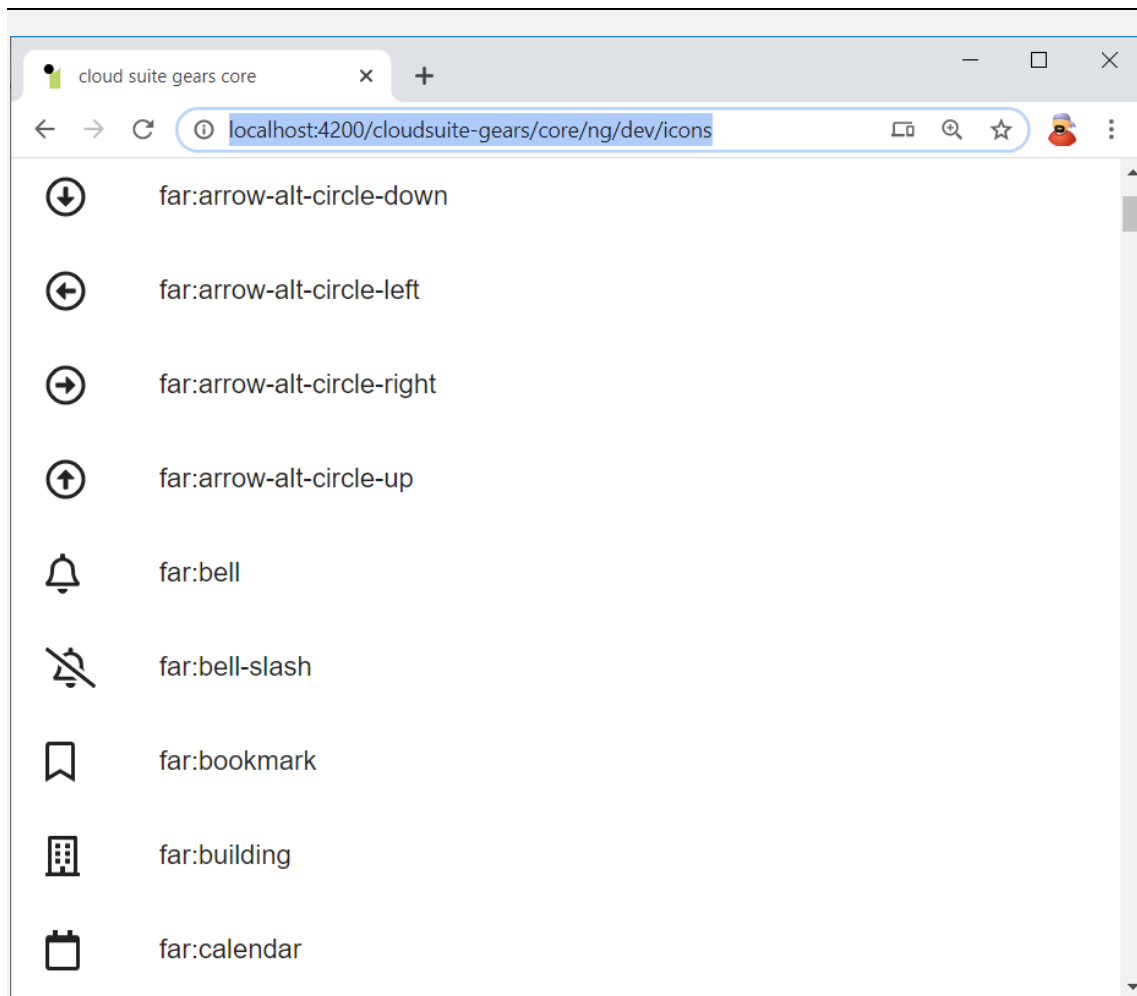
```
{
  "parent": "de.intarsys.widget.toolbar.additions",
  "icon": "fas:angry",
  "callbacks":
  {
    "select":
    {
      "factory": "Alert",
      "args":
      {
        "message": "hi"
      }
    }
  }
}
```

If no prefix is supplied, "far" is used.

You can see a list of all icons currently loaded into the application and available for use in a widget using the link

```
<host>/cloudsuite-gears/core/ng/dev/icons
```

This is what you will see:



### 10.2.3 Button styling

If no type is specified, an icon button is created. The label—if present— is only used as a tooltip. Widgets with an explicit declaration “Button” can have both an icon and a label which in turn support additional styling. Instead of specifying these as a simple string, you can specify a nested element with name **icon** or **label**.

The icon’s property **descriptor** receives the prefixed icon name and the property **css** receives styling instructions in CSS format.

The label’s property **message** receives the label string and the property **css** receives styling instructions in CSS format.

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.toolbar.additions" type="Button">
  <w:icon descriptor="far:smile" css="color: red;" />
  <w:label message="Say hi" css="font-size: larger;" />
  <w:on event="select" do="Alert">
    <entry key="message" value="hi" />
  </w:on>
</w:widget>
```

---

---

**JSON fragment**

---

```
{
  "parent": "de.intarsys.widget.toolbar.additions",
  "type": "Button",
  "icon": {
    "descriptor": "far:smile",
    "css": "color:red;"
  },
  "label": {
    "message": "Say hi",
    "css": "font-size:larger;"
  },
  "callbacks": {
    "select": {
      "factory": "Alert",
      "args": {
        "message": "hi"
      }
    }
  }
}
```

---

## 10.2.4 Behavior

A more interesting part is the fact that a widget can trigger actions and as such control the application behavior. This is done in the **w:on** child element.

## Spring XML fragment

```
<w:widget parent="de.intarsys.widget.toolbar.additions" id="sign" label="Signiere">
  <w:on event="select">
    <w:action factory="Sign">
      <entry key="requireField" value="false"/>
      <entry key="requireSignature" value="false"/>
      <entry key="signerCreate.configuration" value="myConfiguration"/>
    </w:action>
  </w:on>
</w:widget>
```

## JSON fragment

```
{
  "id": "sign",
  "label": "Signiere",
  "parent": "de.intarsys.widget.toolbar.additions",
  "callbacks": {
    "select": {
      "factory": "Sign",
      "args": {
        "requireField": false,
        "requireSignature": false,
        "signerCreate.configuration": "myConfiguration"
      }
    }
  }
}
```

Our initial example defines that a "select" event from the widget (the button press for a toolbar item) is bound to an action (see chapter below).

There are many predefined actions to support application customization.

The combination of **w:on** with a single action is so frequent that we support a shortcut. You add the **action.factory** as the **on.do** attribute and inline the **entry** elements in the **w:on** element. The example from above will look like this:

## Spring XML fragment

```
<w:widget parent="de.intarsys.widget.toolbar.additions" id="sign" label="Signiere">
  <w:on event="select" do="Sign">
    <entry key="requireField" value="false" />
    <entry key="requireSignature" value="false" />
    <entry key="signerCreate.configuration" value="myConfiguration" />
  </w:on>
</w:widget>
```

In our documentation we use the short XML form where possible.

## 10.3 Action definition

### 10.3.1 Plain action

Actions define the behavioral part of the application. Typically, an action is attached to a widget's event callback via the **w:on** element. Let's refresh our memory with the simple example:

## Spring XML fragment

```
<w:widget parent="de.intarsys.widget.toolbar.additions" id="sign" label="Signiere">
  <w:on event="select" do="Sign">
    <entry key="requireField" value="false" />
    <entry key="requireSignature" value="false" />
    <entry key="signerCreate.configuration" value="myConfiguration" />
  </w:on>
</w:widget>
```

## JSON fragment

```
{
  "id": "sign",
  "label": "Signiere",
  "parent": "de.intarsys.widget.toolbar.additions",
  "callbacks": {
    "select": {
      "factory": "Sign",
      "args": {
        "requireField": false,
        "requireSignature": false,
        "signerCreate.configuration": "myConfiguration"
      }
    }
  }
}
```

This definition binds a signer action to a toolbar widget.

An action definition requires a **factory**, which is one of the well-known predefined action types (see section "Action reference"). Most often the action can be parameterized with additional arguments to refine the execution. In the example above, we tell the "Sign" action that no local field definition or graphical signature is collected before signature creation.

### 10.3.2 Complex action

You can define a more complex action block with these elements

- w:action
- w:catch
- w:finally

Example

## Spring XML fragment

```

<w:widget parent="de.intarsys.widget.toolbar.additions" label="MySign">
  <w:on event="select">
    <w:action factory="Greet">
      <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="Greet">
      <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>

```

## JSON fragment

```

{
  "try": [{
    "factory": "Alert",
    "args": {
      "message": "step 1"
    }
  }, {
    "factory": "Confirm",
    "args": {
      "message": "step 2"
    }
  }
],
  "catch": {
    "factory": "Alert",
    "args": {
      "message": "catch"
    }
  },
  "finally": {
    "factory": "Alert",
    "args": {
      "message": "finally"
    }
  }
}

```

This complex example shows a bunch of features:

First, you can add multiple actions that 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.

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 "**w: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.

You can nest complex action blocks if required, for example when creating a list of complex actions in the viewer configuration.

This example defines a list of two actions, each with a complete block.

### Spring XML fragment

```
<w:action>
  <w:action factory="Alert">
    <entry key="message" value="I will sign now"/>
  </w:action>
  <w:catch factory="Alert">
    <entry key="message" value="I failed"/>
  </w:catch>
  <w:finally factory="Alert">
    <entry key="message" value="I finished"/>
  </w:finally>
</w:action>
<w:action>
  <w:action factory="Alert">
    <entry key="message" value="I will sign now"/>
  </w:action>
  <w:catch factory="Alert">
    <entry key="message" value="I failed"/>
  </w:catch>
  <w:finally factory="Alert">
    <entry key="message" value="I finished"/>
  </w:finally>
</w:action>
```

## 10.3.3 Execution semantics

### 10.3.3.1 Arguments

An action has arguments that are specific to the action implementation. E.g., the "Alert" action supports the "message" argument.

These arguments are looked up via (in the order of precedence)

- spec args  
The arguments that you statically provide with the action definition
- execution args  
The arguments that are available in the execution context at runtime, either via the triggering event or some intermediate action execution (see below)

The action tries to find an argument using the well-known name as defined in the reference. To support more complex scenarios, you can access an argument using the argument binding syntax with a "?" suffix after the argument you want to assign.



```

{
  try : [{
    factory: "Prompt",
    args: {
      "message": "Enter:"
      "=": "message1"
    }
  }, {
    factory: "Alert",
    args: {
      "message?": "message1"
    }
  }
]
}

```

### 10.3.3.2 Result

An action can produce a result of any type.

The result of the last action execution is returned to the caller.

In addition, the result of any action can be pushed to the execution context, where it can be looked up as an argument. You can enforce this transfer to the execution context by defining the special argument "=".

```

{
  try : [{
    factory: "Prompt",
    args: {
      "message": "Enter"
      "=": "message1"
    }
  }, {
    factory: "Alert",
    args: {
      "message?": "message1"
    }
  }
]
}

```

Some actions have predefined result roles (like "CollectField" and others), so the result object is automatically pushed to the execution context using a default name. You can still overwrite this name at any time using the syntax above.

### 10.3.3.3 Expression evaluation

Argument values can be derived using expressions – you have seen examples using expressions in the chapters above.

An argument value is computed as an expression (instead of using the plain literal value) by appending a "?" to the argument name:

```

{
  factory: "Alert",
  args: {
    "message?": "foo"
  }
}

```

The example above will evaluate the expression "foo" and show the result as the alert message.

```
{
  factory: "Alert",
  args: {
    "message": "foo"
  }
}
```

This example will simply show the string "foo".

The expression engine can be plugged in – by default there's a simple engine that allows named access to the application state.

### 10.3.4 Simple expressions

The simple expression engine allows reading and writing of application state.

The expression syntax is simply

```
[$container.]property
```

an optional container name and the property. If the container is omitted the default container is used – in this case the active execution context. After the action execution, the values are lost.

The predefined containers are

- execution  
The current action execution. The container lives from the start of action execution until the end.
- page  
The current browser page. The container lives as long as the application page is loaded (a page change to an external application like sign-me etc. will clear the page container)
- session  
The current browser session.
- local  
The browser local storage

Examples:

```
{
  "factory": "SetValue",
  "args": {
    "name": "myvar",
    "value": "foobar"
  }
}
```

```
{
  "factory": "SetValue",
  "args": {
    "name": "$execution.myvar",
    "value": "foobar"
  }
}
```

These snippets will both set the property "myvar" to the literal "foobar".

```
{
  "factory": "SetValue",
  "args": {
    "name": "$page.myvar",
    "value": "foobar"
  }
}
```

This snippet will set the property "myvar" in the page container to the literal "foobar". The value is accessible (e.g. via GetValue) as long as the application is not reloaded.

An expression can be evaluated by marking an action argument assignment with the suffix "?".

```
{
  "factory": "Alert",
  "args": {
    "message?": "myvar"
  }
}
```

```
{
  "factory": "Alert",
  "args": {
    "message?": "$execution.myvar"
  }
}
```

These snippets will both read the property "myvar" from the execution context.

```
{
  "factory": "Alert",
  "args": {
    "message?": "$page.myvar"
  }
}
```

This snippet will read the property "myvar" from the page container.

## 10.4 String expansion

You have seen that widget and action definitions are configured statically. While you can always define multiple configurations for a component like the viewer, it may be tedious or impossible to enumerate all possibilities.

This is why the widget and action definitions can access variables by using string expansion.

Let's review the initial example

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.toolbar.additions" id="sign" label="Signiere">
  <w:on event="select" do="Sign">
    <entry key="requireField" value="{flow.variables.myvar.a}" />
    <entry key="signerCreate.configuration" value="{flow.variables.myvar.b}" />
  </w:on>
</w:widget>
```

---

**JSON fragment**

---

```
{
  "id": "sign",
  "label": "Signiere",
  "parent": "de.intarsys.widget.toolbar.additions",
  "callbacks": {
    "select": {
      "factory": "Sign",
      "args": {
        "requireField": false,
        "requireSignature": false,
        "signerCreate.configuration": "${flow.variables.myvar.b}"
      }
    }
  }
}
```

---

Now the action argument "requireField" and "signerCreate.configuration" are replaced with a string expansion expression.

The string expansion should be performed at runtime, not at system startup time. If you use the "\${}" notation here in a Spring XML file, Spring will kick in and try to resolve when parsing. You either need to revert to Spring expression language notation (which is quite unreadable) or use the "?" notation within Spring configuration files.

Note that you do not need this replacement in JSON.

String expansion is performed immediately before the configuration is serialized and streamed to the client component.

Within the expression you can use all available namespaces (see reference section). A namespace that is particularly interesting is "flow.variables". All information you sent previously with the "variables" property of your flow creation request or within a referenced configuration can be used here.

Example request excerpt

**service call**

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

```
{
  "options": {
    ...
  },
  "variables": {
    "myvar": {
      "a": false,
      "b": "testConfiguration",
    },
    ...
  },
  ...
}
```

This will set the **requireField** argument to "false" and the **signerCreate.configuration** argument to "testConfiguration".

String expansion is for example supported for

- widget.icon
- widget.label
- widget.tip
- all widget property values
- all action argument values

## 10.5 Overlay widgets

Overlays are widgets that implement the different layers of information that together represent the information in the viewer.

For example, we have an overlay that renders the background, one that renders the PDF page itself and one that renders annotations on the page.

Most overlays that are active in your viewer are configurable, while the background and page overlay are added by default.

### 10.5.1 Annotations overlay

This overlay shows the annotations available on a page.

If you want to see the overlay in your viewer, you add the corresponding widget in your viewer configuration:

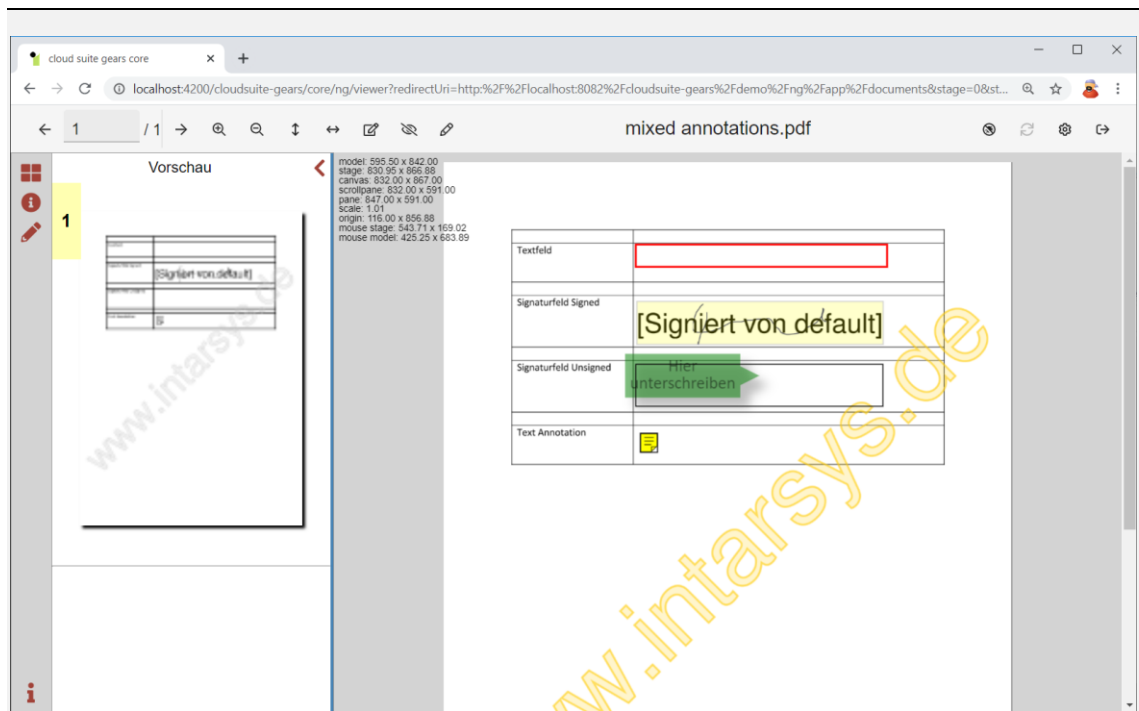
**Spring XML fragment**

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

**JSON fragment**

```
{
  "parent": "de.intarsys.widget.renderer.overlays",
  "id": "annotations",
  "type": "AnnotationsOverlay"
}
```

Here you see the overlay at work



### 10.5.1.1 Annotations

The overlay will collect and render annotations from different sources.

#### 10.5.1.1.1 PDF annotations

First, all PDF annotations (widgets and markups) are extracted from the document.

#### 10.5.1.1.2 Tag based annotations

The second collector allows you to "create" annotations leveraging simple word processing tools, without diving deep in PDF libraries. If you are already familiar with the "document tags" concept you will find it very easy:

Just embed a certain text in the document and get the annotation created by gears. In this case the text must follow the form:

```
annot.<name>={ "key": "value", ...}
```

First, we indicate that this tag will be collected by our collector by starting it with "annot.". Then we can attach a unique name for the annotation. After the "=" sign we follow with a plain JSON string, holding key/value pairs for all other parameters that we require.

Take care of the correct syntax for the JSON string – especially to use simple quotes, not typographic ones!

You can create multiple tags with the same name, indicating that an operation on this tag should use all occurrences.

These are the properties that are actually usable along with the annotation tags.

annotationType	
string	Force this annotation to be of a specific type (either "markup" or "widget", "auto" if the software decides).
	Default "auto"
name	

string	By default the name is derived from the suffix of the tag. This property allows to overwrite the name for the field.
llx	
integer	Lower left X coordinate of the field.  The default is derived from the position where the tag is located.
lly	
integer	Lower left Y coordinate of the field.  The default is derived from the position where the tag is located.
width	
integer	Width of the field. In certain circumstances it may be easier to give a fix width than to derive from the tag.  The default is derived from the position where the tag is located.
height	
integer	Height of the field. In certain circumstances it may be easier to give a fix height than to derive from the tag.  The default is derived from the position where the tag is located.
pageRange	
a page range description	Page(s) of the field.  The default is derived from the page where the tag is located.
type	
string	Type of the field. This is the PDF specific annotation type name.  Default "Widget"
subType	
string	Subtype of the field, if required. This is the PDF specific acro form field subtype name.  Default "Sig"

## Full example

```
annot.sig1={ "width": 200, "height": 50 }
```

If using this technique be sure to activate the PDF tag detection in the viewer arguments.

## Example

```
{
  "documentTagDetector": {
    "factory": "de.intarsys.document.app.pdf.tags.PDFContentTagDetectorFactory",
    "args": {
      "syntax": "separated"
    }
  }
}
```

## 10.5.1.1.3 "annotations" widget

There is a widget container "annotations" in the overlay that allows to define "virtual" annotations that are not yet contained on the page or within tags and as such cannot be derived from the document.

This allows to add and interact with "virtual" annotations via viewer configurations.

For every annotation you want to be rendered in the overlay, you add a child widget to "annotations".

The annotation is then derived from these widget properties:

name	
String	<p>The name for the annotation.</p> <p>This property is required and must be unique for all annotation names already used in the document.</p>
type	
String	<p>The annotation type represented by this widget. This is derived from the available annotation types in the PDF specification.</p> <p>Currently supported:</p> <ul style="list-style-type: none"> <li>Widget This is a PDF form field annotation</li> </ul>
subtype	
String	<p>The annotation subtype represented by this widget. This is derived from the annotation subtypes (if available) in the PDF specification.</p> <p>Currently supported:</p> <ul style="list-style-type: none"> <li>Sig This is a signature form field</li> </ul>
position	
String	<p>A definition for the position of the annotation in user space (see [4]) as number 'x' number (you can use either 'x', '*' or '@' as a separator).</p> <p>Example</p> <ul style="list-style-type: none"> <li>100x100</li> </ul>
size	
String	<p>A definition for the size of the annotation in user space (see [4]) as number 'x' number (you can use either 'x', '*' or '@' as a separator).</p> <p>Example</p> <ul style="list-style-type: none"> <li>200x100</li> </ul>
pageRange	
String	<p>A definition for the pages where the annotation is rendered (see [4]).</p> <ul style="list-style-type: none"> <li>number a single page</li> <li>number-number an interval</li> <li>element;element an enumeration of the above</li> </ul> <p>Example</p> <ul style="list-style-type: none"> <li>1;5-8;10 Select page 1, or any page from 5 to 8 or page 10</li> </ul>
hAlign	
String	<p>A definition for the horizontal alignment of the annotation rectangle (see [4]).</p> <p>One of <b>left</b>   <b>right</b>   <b>center</b></p>
vAlign	
String	<p>A definition for the vertical alignment of the annotation rectangle (see [4]).</p>



One of <b>bottom</b>   <b>top</b>   <b>center</b>
---

### Example

### Spring XML fragment

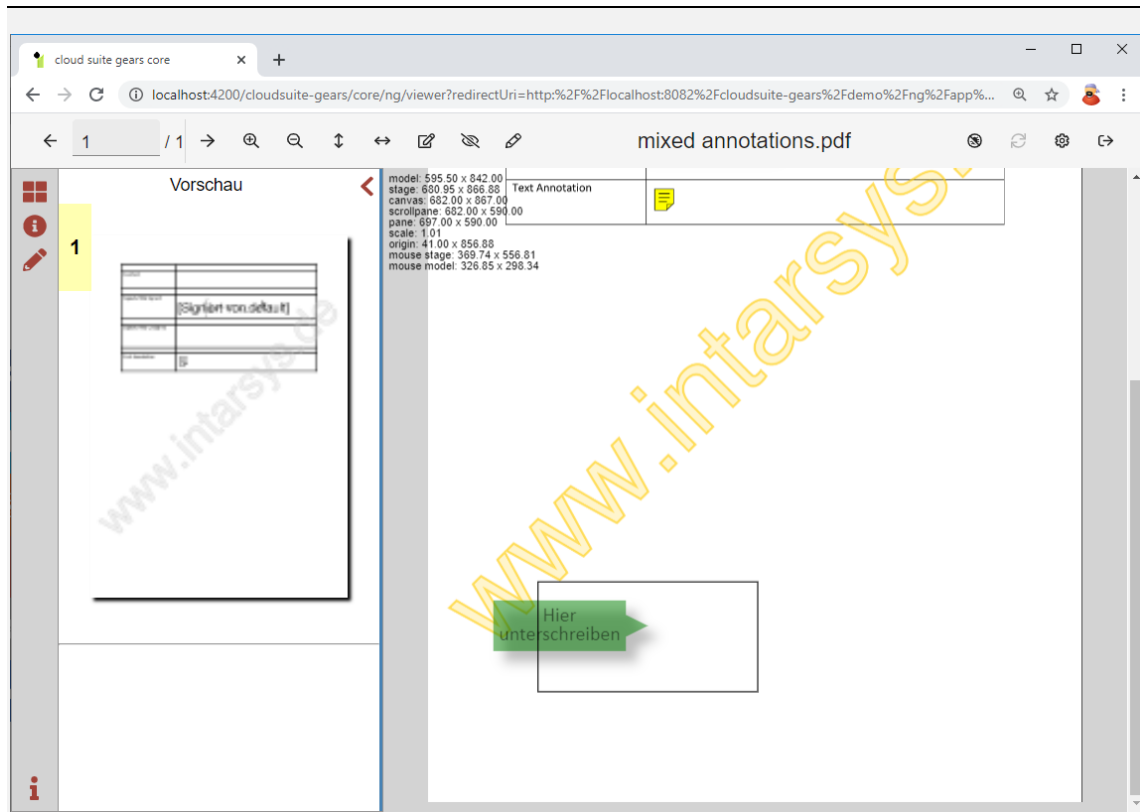
```
<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotations"
type="AnnotationsOverlay">
  <w:widget id="annotations">
    <w:widget>
      <w:property name="name" value="MySignatureField"/>
      <w:property name="type" value="Widget"/>
      <w:property name="subType" value="Sig"/>
      <w:property name="position" value="100x100"/>
      <w:property name="size" value="200x100"/>
      <w:property name="pageRange" value="all"/>
      <w:on event="select" do="Alert">
        <entry key="message" value="Hi there" />
      </w:on>
    </w:widget>
  </w:widget>
</w:widget>
```

### JSON fragment

```
{
  "parent": "de.intarsys.widget.renderer.overlays",
  "id": "annotations",
  "type": "AnnotationsOverlay",
  "children": [
    {
      "id": "annotations",
      "children": [
        {
          "properties": {
            "name": "MySignatureField",
            "type": "Widget",
            "subType": "Sig",
            "position": "100x100",
            "size": "200x100",
            "pageRange": "all"
          }
        }
      ]
    }
  ]
}
```

This will add a widget in the document lower left that can be interacted with the same way as if it had been defined in the document.

The widget can contain callback definitions that can be called by the "canvas" forward technique.



#### 10.5.1.1.4 Callbacks

You can attach callbacks (event/action pairs) directly to an annotation for later execution in the UI (e.g. via the "canvas" or the "popup").

The simplest way to define a callback is directly in the widget within the "annotations" widget.

A tag based annotation can come along with "on-<event>" or even "callbacks" properties.

If no explicit callbacks are available this way, a default mechanism is applied: All actions in the action registry that match

```
annot.<name>.<event>
```

where <name> is the annotation name and <event> is the event name, for example

```
annot.sig1.select
```

is attached to the annotation target.

#### 10.5.1.2 Styles

The rendering of components in the overlay is customized using styles. A style object defines the appearance of a graphical shape.

Style properties

stroke	
string	The color to be used for the border
strokeWidth	
number	The border width
fill	
string	The color used to fill the rectangle

opacity	
number	The opacity of the rectangle fill color (from 0 to 1)

### 10.5.1.3 Rectangle styles

A shape (e.g. an annotation rectangle) can have different styles for different "interaction states". The well-known states are

- normal
- hover
- active

These styles can be configured with the "styles" property of the widget, followed by the style name and then the style property name.

Example

#### Spring XML fragment

```
<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotations"
type="AnnotationsOverlay">
  <w:property name="styles.normal.stroke" value="green" />
  <w:property name="styles.normal.strokeWidth" value="5" />
  <w:property name="styles.active.stroke" value="" />
  <w:property name="styles.active.fill" value="blue" />
  <w:property name="styles.active.opacity" value="0.5" />
</w:widget>
```

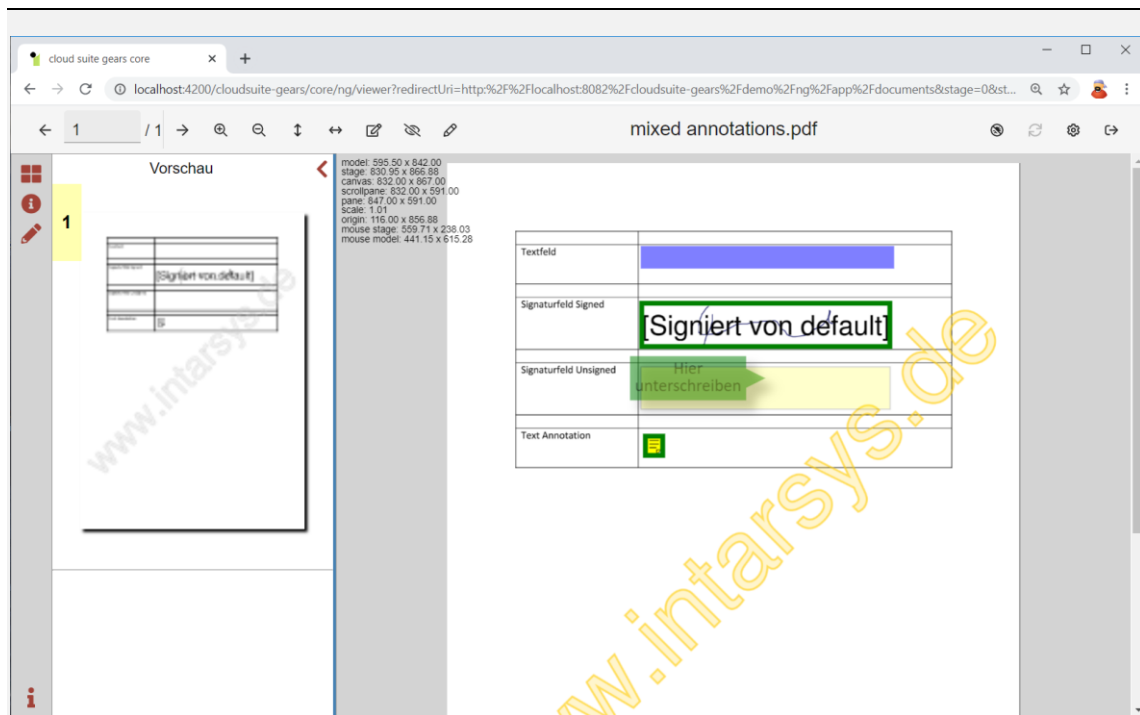
#### JSON fragment

```
{
  "parent": "de.intarsys.widget.renderer.overlays",
  "id": "annotations",
  "type": "AnnotationsOverlay",
  "properties": {
    {
      "styles.normal.stroke": "green",
      "styles.normal.strokeWidth": 5,
      "styles.active.stroke": "",
      "styles.active.fill": "blue",
      "styles.active.opacity": 0.5
    }
  }
}
```

The default settings are

- each annotation is marked as a rectangle with black border ("normal" style)
- hovering an annotation will show a yellow glow in the rectangle ("hover" style)
- selecting the annotation will draw a red border ("active" style)

After applying the example the appearance will change – be aware that we need to explicitly switch off "styles.active.stroke", the default red border would have been shown up otherwise.



#### 10.5.1.4 Callout style

A signature field may have a "callout", indicating to the user that this field is active and should be selected to start a signature operation.

The appearance of this callout can be customized using widget properties, too.

First, you select the target of the customization. In this case this is

```
target.WidgetSigUnsigned.callout
```

##### callout properties

visible	
Boolean	Flag if the callout should be rendered at all
tag	
object	Properties for the tag attached to the annotation.
text	
object	Properties for the text attached to the annotation.

##### tag properties

stroke	
string	The color to be used for the border
strokeWidth	
number	The border width
fill	
string	The color used to fill the rectangle
opacity	
number	The opacity of the rectangle fill color (from 0 to 1)
shadowColor	
string	The color to be used for the shadow region
shadowBlur	
number	The level of the blurred shadow region
shadowOffsetX	

number	The offset of the shadow in x direction
shadowOffsetY	
number	The offset of the shadow in y direction
shadowOpacity	
number	The opacity of the shadow

## text properties

fill	
string	The color used to draw the text
opacity	
number	The opacity of the text color (from 0 to 1)
fontFamily	
string	The font family Default Arial
fontSize	
number	
fontStyle	
string	normal   bold   italic
text	
string	

## Example

## Spring XML fragment

```

...
<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotations"
type="AnnotationsOverlay">
  <w:property name="target.WidgetSigUnsigned.callout.visible" value="true" />
  <w:property name="target.WidgetSigUnsigned.callout.tag.fill" value="#ea0a8e" />
  <w:property name="target.WidgetSigUnsigned.callout.tag.opacity" value="0.5" />
  <w:property name="target.WidgetSigUnsigned.callout.tag.shadowColor" value="blue" />
  <w:property name="target.WidgetSigUnsigned.callout.tag.shadowBlur" value="10" />
  <w:property name="target.WidgetSigUnsigned.callout.tag.shadowOffsetX" value="50" />
  <w:property name="target.WidgetSigUnsigned.callout.tag.shadowOffsetY" value="50" />
  <w:property name="target.WidgetSigUnsigned.callout.tag.shadowOpacity" value="0.5" />
  <w:property name="target.WidgetSigUnsigned.callout.text.fill" value="white" />
  <w:property name="target.WidgetSigUnsigned.callout.text.opacity" value="1" />
  <w:property name="target.WidgetSigUnsigned.callout.text.fontSize" value="100" />
  <w:property name="target.WidgetSigUnsigned.callout.text.text" value="Tu es..." />
</w:widget>

```

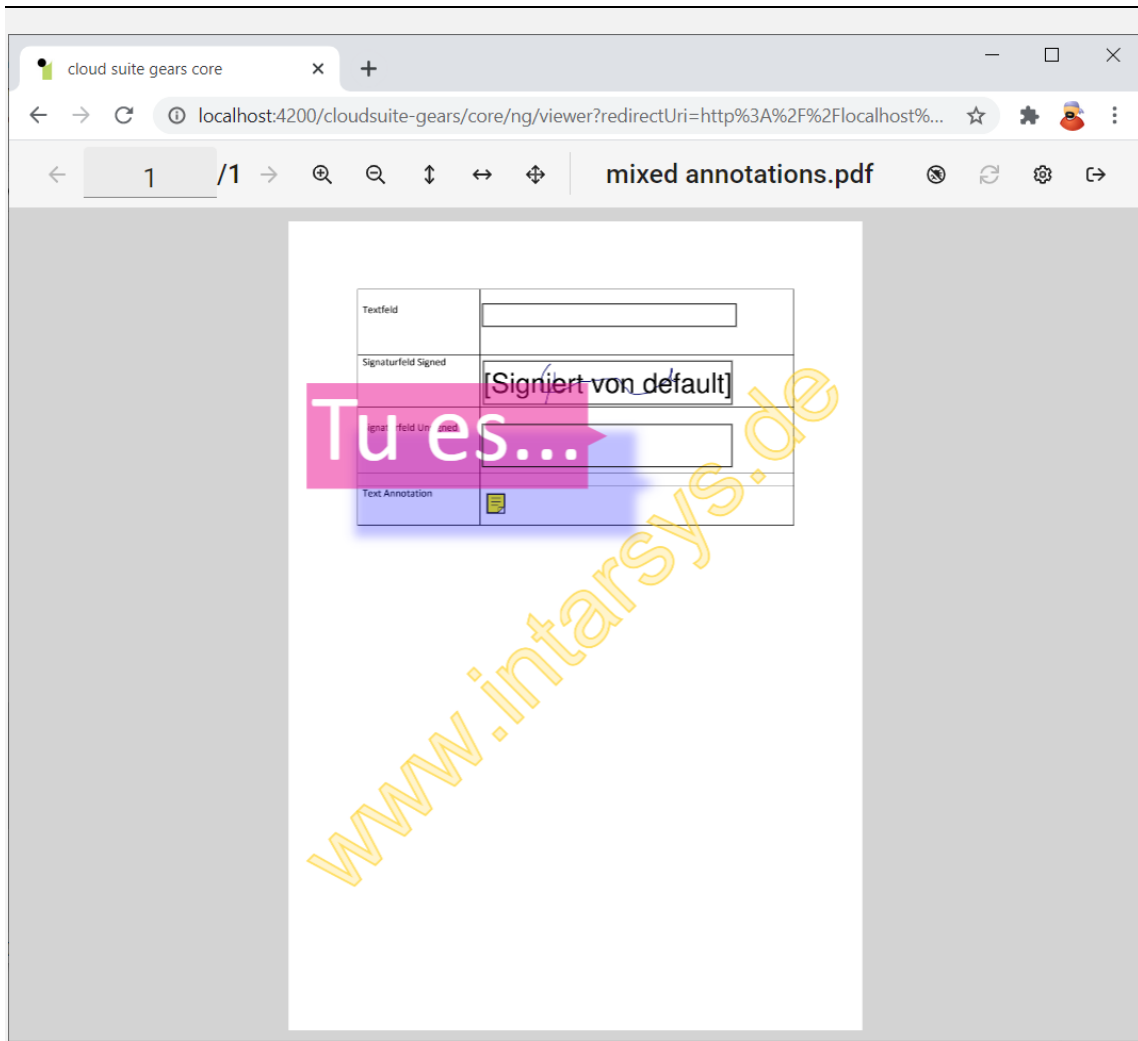
## JSON fragment

```

{
  "parent": "de.intarsys.widget.renderer.overlays",
  "id": "annotations",
  "type": "AnnotationsOverlay",
  "properties":
  {
    "target.WidgetSigUnsigned.callout.visible": true,
    "target.WidgetSigUnsigned.callout.tag.fill": "#ea0a8e",
    "target.WidgetSigUnsigned.callout.tag.opacity": 0.5,
    "target.WidgetSigUnsigned.callout.tag.shadowColor": "blue",
    "target.WidgetSigUnsigned.callout.tag.shadowBlur": 10,
    "target.WidgetSigUnsigned.callout.tag.shadowOffsetX": 50,
    "target.WidgetSigUnsigned.callout.tag.shadowOffsetY": 50,
    "target.WidgetSigUnsigned.callout.tag.shadowOpacity": 0.5,
    "target.WidgetSigUnsigned.callout.text.fill": "white",
    "target.WidgetSigUnsigned.callout.text.opacity": 1,
    "target.WidgetSigUnsigned.callout.text.fontSize": 100,
    "target.WidgetSigUnsigned.callout.text.text": "Tu es..."
  }
}

```

This will result in the callout rendering below:



### 10.5.1.5 Filter

Another important feature is the filtering of annotations, for example you may want to see signature annotations only. A filter can be defined with the "filter.match.\*" widget properties.

If a filter property is defined, only annotations are taken into account that match the regular expression value given. The filter properties are combined with "and", so all predicates must hold true.

To simplify selection, you can prefix the regular expression with "!" to express an inverted condition (all annotations that do **not** match the regular expression value).

Filter properties

type	
string	One of the supported PDF annotation types: <ul style="list-style-type: none"> <li>• Widget</li> <li>• Square</li> <li>• Circle</li> <li>• (... remaining PDF annotation classes)</li> </ul>
subType	
string	If the annotation type is further sub structured (like the "Widget" type), this is the sub-type specification <ul style="list-style-type: none"> <li>• Sig (for signature widgets)</li> <li>• Btn (for button widgets)</li> </ul>

- Tx (for text widgets)

## Example

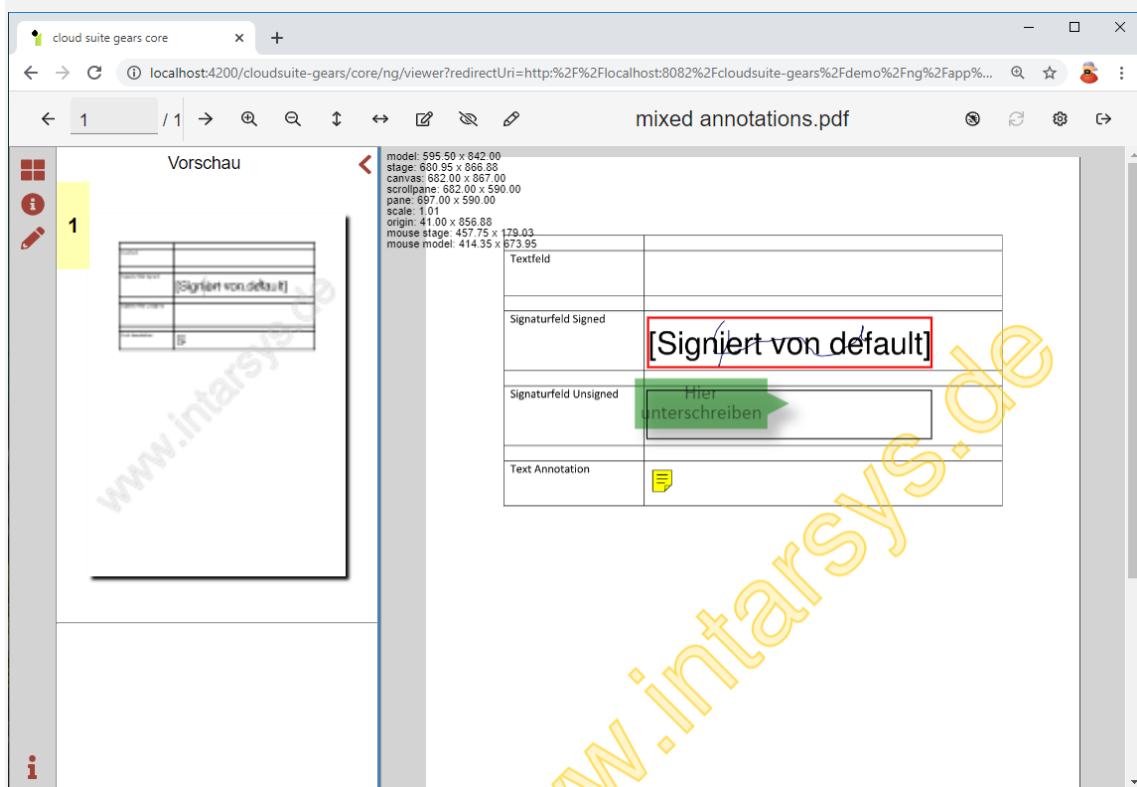
## Spring XML fragment

```
<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotations"
type="AnnotationsOverlay">
  <w:property name="filter.match.type" value="Widget" />
  <w:property name="filter.match.subType" value="Sig" />
</w:widget>
```

## JSON fragment

```
{
  "parent": "de.intarsys.widget.renderer.overlays",
  "id": "annotations",
  "type": "AnnotationsOverlay",
  "properties": {
    "filter.match.type": "Widget",
    "filter.match.subType": "Sig"
  }
}
```

Now only signature fields will get rendered.



### 10.5.1.6 Nested widgets

If it was only for rendering, the annotation overlay would not be very impressive. What would be lacking is the ability to interact...

This is why the annotation overlay supports a bunch of widget definitions.



### 10.5.1.7 "canvas" widget

The "canvas" widget represents the drawing area as a whole. Its purpose is to define what is going to happen when a mouse button is pressed within the rectangle of an annotation.

Example

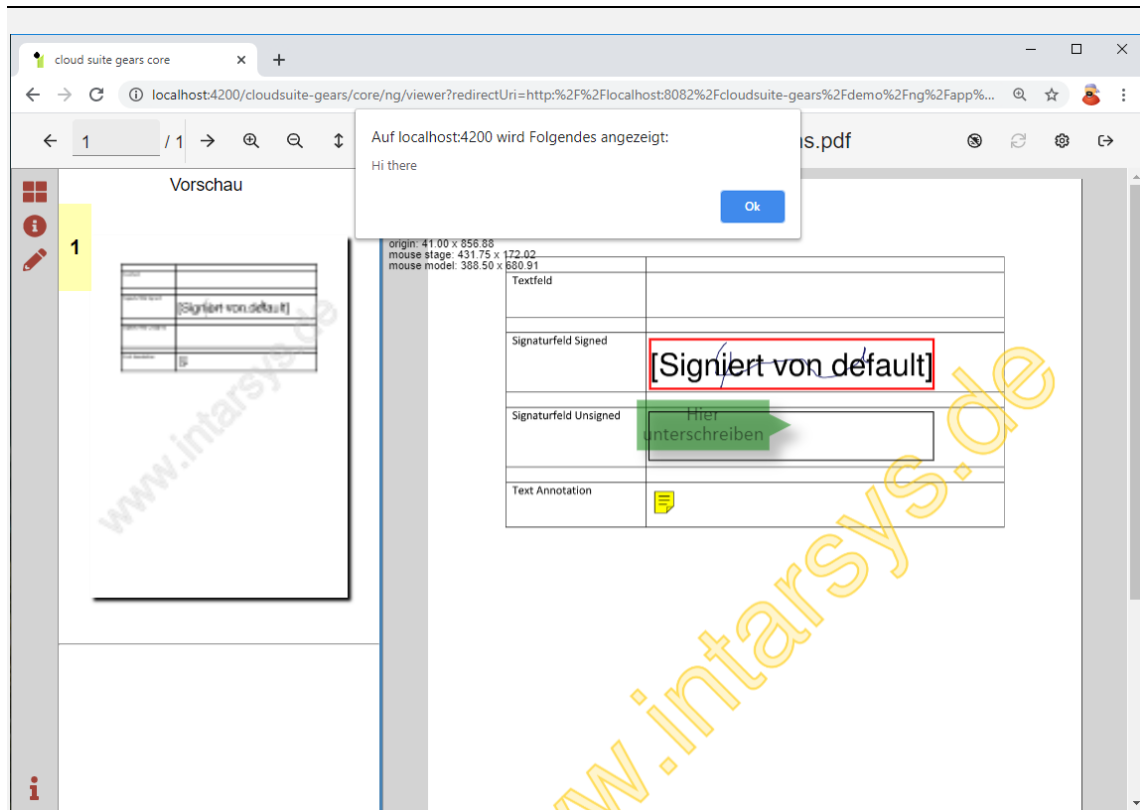
#### Spring XML fragment

```
<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotations"
type="AnnotationsOverlay">
  ...
  <w:widget id="canvas">
    <w:on event="select" do="Alert">
      <entry key="message" value="Hi there" />
    </w:on>
  </w:widget>
</w:widget>
```

#### JSON fragment

```
{
  "parent": "de.intarsys.widget.renderer.overlays",
  "id": "annotations",
  "type": "AnnotationsOverlay",
  "children": [
    {
      "id": "canvas",
      "callbacks": {
        "select": {
          "factory": "Alert",
          "args": {
            "message": "Hi, there"
          }
        }
      }
    }
  ]
}
```

This will simply show an alert box when clicking on any annotation currently rendered.



For sure you can establish a more subtle interaction. Actually, there are two main techniques for adding functionality here.

#### 10.5.1.7.1 "forward" widget

The forward widget allows to delegate callback execution to the target annotation. For defining a callback with an annotation see chapter "Callbacks" above.

The "forward" mechanics is active by default when no "canvas" is defined at all.

Example: "forward" active by default

#### Spring XML fragment

```
<w:widget
  parent="de.intarsys.widget.renderer.overlays"
  id="annotOverlay"
  type="AnnotationsOverlay">
  <!--no child definition -->
</w:widget>
```

If you have a canvas, you can include the "forward" as a direct child or within a "switch/case" (see below).

Example: explicit "forward" child

---

**Spring XML fragment**


---

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

---

"forward" can be contained in a "switch" widget, too. In this combination the "case" can be selected only if the target comes along with a matching callback.

Example: "forward" in a case

---

**Spring XML fragment**


---

```
<w:widget parent="de.intarsys.widget.renderer.overlays" id="annotOverlay"
type="AnnotationsOverlay">
  <w:widget id="canvas">
    <w:widget type="switch">
      <w:widget type="case">
        <w:widget type="forward">
        </w:widget>
      </w:widget>
      <w:widget type="default">
        <w:widget>
          <w:on event="select">
            <w:action factory="Notice">
              <entry key="message" value="no callback found"/>
            </w:action>
          </w:on>
        </w:widget>
      </w:widget>
    </w:widget>
  </w:widget>
</w:widget>
```

---

A callback can actually be defined by a widget in the "annotations" container or with tag-based annotations.

With a widget the technique is well known:

---

**JSON fragment**


---

```
{
  "parent": "de.intarsys.widget.renderer.overlays/annotOverlay/annotations",
  "properties": {
    "name": "foobar",
    "x": 10,
    "y": 10,
    "width": 100,
    "height": 100
  },
  "callbacks": {
    "select": {
      "factory": "Alert",
      "args": {
        "message": "foobar"
      }
    }
  }
}
```

---

When the forward widget is in place, the "Alert" will be triggered by a click.

When you are working with tag-based annotations, you can inline the callback definition directly in the tag:

Using an action reference:

---

#### PDF document content

---

```
@@annot.sig1={ "width": 200, "height":50 , "on-select": "signRef1"}@@
```

---

Take notice that a special short form for the callbacks is used that allows a more concise definition here. **"callbacks" cannot be used here.**

```
"on-select": "signRef1"
```

---

is equivalent to

```
"callbacks": {
  "select": {
    "ref": "signRef1"
  }
}
```

---

Using an action definition:

---

#### PDF document content

---

```
@@annot.sig2={ "width": 200, "height":50 , "on-select": { "factory": "Alert", "args":
{"message": "embedded action"}} }@@
```

---

For better readability you should stick with the reference syntax in example 1. You can reference any action that was defined in your configuration.

### 10.5.1.7.2 "switch" widget

If your annotations are "dumb", the "canvas" widget can be defined to be of type "switch". This allows for any number of child widgets of type "case" and a single "default".

If a user interaction takes place, each "case" child widget is checked in turn if it matches the target object (here the annotation). To express the predicate, we resort to the well-known annotation filter properties, "type" and "subType". In addition, we can differentiate the annotation state of a signature with the "signed" property.

Remember, that you can combine this technique with the "forward" widget!

The following definition will ensure that only signature fields are active. If you click on a field that is not yet signed, the "Signer" action is automatically launched in the context of the annotation.

**Spring XML fragment**

```

<w:widget id="canvas" type="switch">
  <w:widget type="case">
    <w:property name="type" value="Widget" />
    <w:property name="subType" value="Sig" />
    <w:property name="signed" value="false" />
    <w:on event="select" do="Sign">
      <entry key="signerCreate" value="{flow.variables.signerCreate}" />
    </w:on>
  </w:widget>
</w:widget>

```

**JSON fragment**

```

{
  "id": "canvas",
  "type": "switch",
  "children": [
    {
      "type": "case",
      "properties": {
        "type": "Widget",
        "subType": "Sig",
        "signed": false
      },
      "callbacks": {
        "select": {
          "factory": "Sign",
          "args": {
            "signerCreate": "${flow.variables.signerCreate}"
          }
        }
      }
    }
  ]
}

```

**10.5.1.8 "popup" widget**

The popup widget allows to do define entries for a popup menu in the context of an annotation.

As with the "switch" widget you can use both "forward" and predicate based filtering.

You can use the same predicates as in the "canvas" widget above. All matching entries are added to the popup menu.

## Spring XML fragment

```

<w:widget id="popup">
  <w:widget label="Sign">
    <w:property name="type" value="Widget" />
    <w:property name="subType" value="Sig" />
    <w:property name="signed" value="false" />
    <w:on event="select" do="Sign">
      <entry key="signerCreate" value="{flow.variables.signerCreate}" />
    </w:on>
  </w:widget>
  <w:widget label="Clear">
    <w:property name="type" value="Widget" />
    <w:property name="subType" value="Sig" />
    <w:property name="signed" value="true" />
    <w:on event="select" do="AnnotationEdit">
      <entry key="confirm" value="Really?" />
      <entry key="action" value="clear" />
    </w:on>
  </w:widget>
</w:widget>

```

## JSON fragment

```

{
  "id": "popup",
  "children": [
    {
      "label": "Sign...",
      "properties": {
        "type": "Widget",
        "subType": "Sig",
        "signed": false
      },
      "callbacks": {
        "select": {
          "factory": "Sign",
          "args": {
            "signerCreate": "${flow.variables.signerCreate}"
          }
        }
      }
    },
    {
      "label": "Clear",
      "properties": {
        "type": "Widget",
        "subType": "Sig",
        "signed": true
      },
      "callbacks": {
        "select": {
          "factory": "AnnotationEdit",
          "args": {
            "confirm": "Really?",
            "action": "clear"
          }
        }
      }
    }
  ]
}

```

This example adds a popup menu with a single entry to an unsigned signature field, allowing to launch a signature action via context menu. If the field is signed, an action is added that will clear the signature content.

## 10.5.2 Fine rendering overlay

This overlay renders a PDF page in view resolution improving the quality of the display.

This means that whenever you zoom in or out, a new image of the page is loaded from the server. To save memory, only the part of the page you can see is loaded. If you scroll to a different part of the page, another new image will be loaded.

This overlay is not active by default; however, it is part of the demo viewer configuration “demoSignature”.

If you want to see the overlay in your viewer, you add the corresponding widget in your viewer configuration:

---

### Spring XML fragment

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

---

### JSON fragment

```
{
  "parent": "de.intarsys.widget.renderer.overlays",
  "id": "viewerFine",
  "type": "ViewerFineOverlay"
}
```

### 10.5.2.1 Activate overlay in gears demo application

Using the gears demo application this overlay can also be activated via the viewer configuration:

Go to <http://<host>/cloudsuite-gears/demo/ng/app/settings>  
At “Viewer configuration” add the following

---

### gears demo viewer configuration

```
[
  { "widgets": [
    {
      "parent": "de.intarsys.widget.renderer.overlays",
      "id": "viewerFine",
      "type": "ViewerFineOverlay"
    }
  ] },
  "myFlowViewerConfiguration"
]
```

This assumes that a FlowViewerConfiguration with id " myFlowViewerConfiguration" is already present.

# 11. Authentication

---

## 11.1 Overview

Starting with version 8.2.0, the security mechanics are completely handled by Spring security. This provides a broad range of features and protocols for authentication and application security, all implemented by an industry proven framework.

If you want to dive into configuring the security on your own, the Spring documentation (<https://docs.spring.io/spring-security/site/docs/5.3.2.RELEASE/reference/html5/>) is considered required prerequisite knowledge!

## 11.2 Opt-out

All chapters below describe in detail the security default configuration and some customizing options if you are OK with the overall design.

All decisions of the default security configuration are “opinionated” and provided as we believe this is a sound basic production configuration. For your reference, the default security configuration (“spring-gears-security.xml”) is provided in “example/spring-beans/gears security default”.

If this design does not fit your needs, you can either customize (by reconfiguring spring beans) or completely opt out of the gears default security by adding the profile

---

`customSecurity`

---

After this, gears backs off and spring security is off – spring boot auto configuration is switched off, too.

Now you can (and should) construct a spring security configuration of your own.

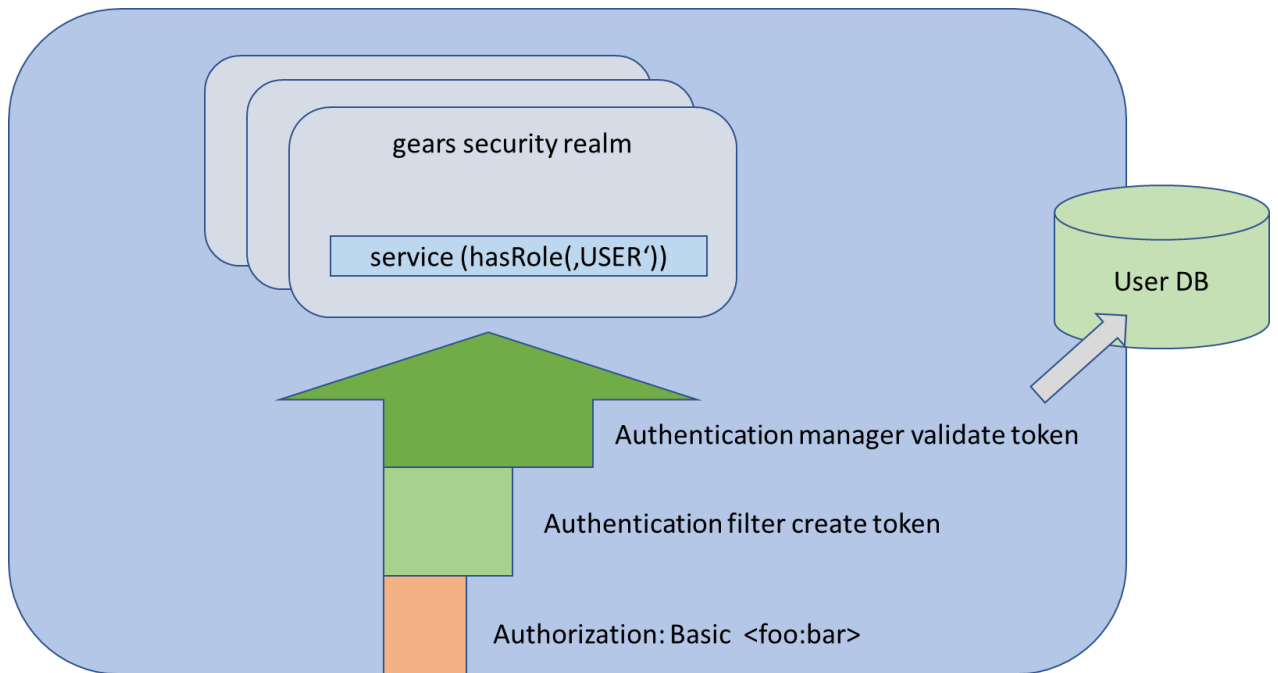
You can take the built-in gears security configuration as an example.

A very basic security definition is provided in “example/spring-beans/gears security simple”.

## 11.3 Concepts

To better understand the gears security and its mapping to spring security concepts, we give a short overview of the building blocks.





First, the gears application comprises different areas, such as the pure "business functions (signing, ...)", an operating area where one can inspect and control gears internal processes and a management area designed to provide status information to a management console (the concrete areas are described in sections below).

These areas are called **security realms**, each bundling a set of features available only to certain role(s). These security realms are pre-defined by gears, defining the security requirements of the application.

Now, we have a resource and a role to play. Here an installation dependent part comes into play, we need to extract information from the request on the server side that designates an entity that holds this role.

This information is transferred using standard communication protocols. Common protocols are "HTTP Basic Authentication" or "X.509 Client Authentication". Each of these protocols requires some input from the client in a specific format. The server extracts this information for further authentication steps using **an authentication filter**. As the decision which protocol to use (or none at all) is up to your installation, this is the first part that is provided by you.

The second important contribution from your side is the mapping to entities known in your company. E.g. if we receive a basic authentication request for user "foo", we must

- lookup a user in some repository
- check the password validity
- detect roles that are associated with the user

This is done using **an authentication manager**.

The good news is:

- If we really need to, we can plug in any implementation into the Spring security framework that we want.
- In most cases we are done with the predefined fine tunable components that are already available.

## 11.4 Security realms

## 11.4.1 Overview

A security realm is a group of services that share common security settings. Technically they are implemented as a distinct Spring "SecurityFilterChain" for each of them.

While each of the realms is provided by a named bean, you should really keep away if you do not know what you are doing.

In the following chapters, whenever we refer to a "realm name", use one of the defined realms

---

Flow | Control | Manage

---

## 11.4.2 Common properties

There are currently no properties available to directly configure the realm definitions.

## 11.4.3 Security realm "Flow"

This security realm bundles all "business functions", the features why you installed gears initially and made available to client applications.

More concrete these are services using the path prefixes

- /api/v1/flow/signer/\*\*
- /api/v1/flow/viewer/\*\*
- /api/v1/flow/explorer/\*\*

The services in this realm require the role "USER" to be available.

The realm bean is "securityRealmFlow".

## 11.4.4 Security realm "Control"

This security realm provides services to inspect and control the running application, like "suspend signature pool".

More concrete these are services using the path prefixes

- /api/v1/control/\*\*

The services in this realm require the role "OPERATOR" to be available.

The realm bean is "securityRealmControl".

## 11.4.5 Security realm "Manage"

This security realm hosts the services required to manage and automate production. You can request information if the system is up and if certain features are available, for example for driving a load balancer or an alert system.

More concrete these are services using the path prefixes

- /manage/\*\*

The services in this realm require the role "MANAGER" to be available.

The realm bean is "securityRealmManage".

## 11.5 Logout

To ensure that no authentication is cached on the server side, you can leverage the logout endpoint. If there is any user in an associated session, the authentication information is discarded.

The endpoint is available at the path

---

```
/logout
```

---

Please note that this has no effect on data stored by your browser. If you enter authentication data in the dialog offered by the browser, the behavior is totally up to the browser. It may (or not) cache this information and you may (or not) be able to clear this information in a more or less simple way.

If you want to play around with the services using a plain browser, you may want to activate incognito mode upfront.

## 11.6 Authentication filter

The task of an authentication filter is to extract protocol specific information and inject it into the Spring security framework APIs.

With Spring you have a variety of authentication filters available. Details regarding the Spring security framework you can find here:

<https://docs.spring.io/spring-security/site/docs/5.3.x/reference/html5/>

Each security realm has an authentication filter of its own that needs to be replaced with the one required for your concrete security design. The id for the authentication filter bean is build using the following pattern:

**springRealm<realm name>AuthenticationFilter**

resulting in the following beans that need to be available:

- securityRealmControlAuthenticationFilter
- securityRealmManageAuthenticationFilter
- securityRealmFlowAuthenticationFilter

### 11.6.1 Static authentication

The first filter we want to introduce is provided by gears. As the security design requires an authenticated role to be present and we don't want to send authentication information, we inject the required authentication token and the authorities (roles) statically to satisfy the security checks.

The filter definition below injects an authenticated token with role "OPERATOR" for every call in the realm "control" (see reference section). This is the **default filter definition** for all security realms (with the respective required roles), you don't need to add this yourself.

---

#### Spring XML fragment

---

```
<bean
  id="securityRealmControlAuthenticationFilter"
  class="de.intarsys.spring.security.StaticAuthenticationFilter">
  <property name="authorities" value="ROLE_OPERATOR" />
</bean>

<!-- other realms look similar -->
```

---

If you are using this filter, **no** security checks are made on behalf the gears application. You should verify that the execution environment has the ability to filter all invalid requests (e.g. using VPNs, secured proxies, ...).

## 11.6.2 Basic authentication

Basic authentication is one of the most common authentication protocols used in the web (and its ok as long as you are using transport level security).

Basic auth requires a header

**Authorization: Basic <base64(user:password)>**

to be sent to the server. The server then extracts this information and looks up in an authentication manager (see below) if this combination is valid and what authorities are attached.

To activate this feature you can leverage the Spring framework  
**org.springframework.security.web.authentication.www.BasicAuthenticationFilter.**

---

### Spring XML fragment

---

```
<bean id="securityRealmControlAuthenticationFilter"
class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">
  <constructor-arg ref="securityRealmControlAuthenticationManager" />
</bean>
```

---

This definition injects a basic authentication filter in the security realm "control". For authentication purposes the default gears authentication manager is used (more on authentication managers in the next section).

Now we have basic authentication in place, accepting only user/password combinations that are defined in the authentication manager.

In much the same way you can tweak the configuration for the other realms.

Detailed scenarios using BasicAuth are described in [5].

## 11.6.3 OAuth2 authentication

Spring has broad support for OAuth2 token support.

Detailed scenarios using OAuth2 are described in [5].

## 11.6.4 X.509 authentication

Besides the importance of TLS as a secure transport level, you can use the TLS information for authentication purposes, too. In this case you have to adopt client-side TLS. You can find more information on TLS setup in [5]. Here we will only have a look at the Spring security specific part that layers on top of TLS transport.

Detailed scenarios using X.509 certificates are described in [5].

## 11.7 Authentication manager

The task of an authentication manager is to take the provided IDs and credentials that have been extracted by the authentication filter, and decide if they are valid and what authorities are associated (in our case, roles).

Again, Spring provides a framework and lots of default authentication providers to quickly get up and running.

Again, each security realm has an authentication manager of its own that needs to be replaced with the one required for your concrete security design. The id for the authentication manager bean is built using the following pattern:

**springRealm<realm name>AuthenticationManager**

resulting in the following beans that need to be available:

- securityRealmControlAuthenticationManager
- securityRealmManageAuthenticationManager
- securityRealmFlowAuthenticationManager

By **default**, all of them are mapped to an in-memory repository of three users, namely "user" (password "user"), "operator" (password "operator") and "manager" (password "manager").

---

### Spring XML fragment

```
<security:authentication-manager id="securityRealmControlAuthenticationManager">
  <security:authentication-provider>
    <security:user-service>
      <security:user name="operator" password="{noop}operator"
authorities="ROLE_OPERATOR" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>

<!-- other realms look similar -->
```

For sure you do not want to use this in production (except you have set up external authentication, but in this case, you should have a "StaticAuthenticationFilter" in place anyhow, effectively disabling authentication manager lookup).

## 11.7.1 In-memory repository

Let's assume you want to introduce another in-memory representation of your users, but only for the security realm "control". We enhance our example from the authentication filter chapter with an authentication manager.

---

### Spring XML fragment

```
<security:authentication-manager id="securityRealmControlAuthenticationManager">
  <security:authentication-provider>
    <security:user-service>
      <security:user name="foo" password="{noop}bar" authorities="ROLE_OPERATOR" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>

<bean id="securityRealmControlAuthenticationFilter"
class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">
  <constructor-arg ref="securityRealmControlAuthenticationManager" />
</bean>
```

Now if you want to access the "control" section of the gears UI, you can enter with user "foo", password "bar".

You can find this configuration in the "/example/spring-beans/gears security control basicauth inmemory" folder of the gears product bundle.

## 11.7.2 JDBC repository

A more production relevant example is the use of a JDBC based repository. Spring comes with a default component that allows to

- map the user token data to a database schema
- check the password against the encoded database version
- attach roles read from the database

To use an existing user database, you can leverage Spring's

**org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl**. Here's an example that uses the internal demo tables provided by the gears "demo" profile.

### Spring XML fragment

```
<security:authentication-manager id="securityRealmControlAuthenticationManager">
  <security:authentication-provider user-service-ref="myUserDetailsService" >
    </security:authentication-provider>
  </security:authentication-manager>

  <bean id="myUserDetailsService"
class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
    <property name="dataSource" ref="dataSource" />
    <property name="usersByUsernameQuery" value="select name, password, enabled from
DemoUser where name=?" />
    <property name="authoritiesByUsernameQuery" value="select user_fk, authority from
DemoAuthority where user_fk=?" />
  </bean>

  <bean id="securityRealmControlAuthenticationFilter"
class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">
    <constructor-arg ref="securityRealmControlAuthenticationManager" />
  </bean>
```

You can find this configuration in the "example/spring-beans/gears security control basicauth jdbc" folder of the gears product bundle.

## 11.8 Principal integration

Now that we have authentication in place, we have the **option** to derive a principal for further handling in gears. You can find more information on principals in chapter "Principals".

gears has a dedicated principal provider that relies on Spring security authentication information. We only have to fill in how to derive the principal from the authentication data. This can be done using a key converter and the well-known principal DAOs.

### 11.8.1 Dao based principal lookup

You can extract the name from the Spring security authentication token and lookup a principal. This is done by using the key converter

```
de.intarsys.cloudsuite.gears.security.spring.AuthenticationToStringConverter
```

and then a DAO that you have set up to deal with this key.

Example

## Spring XML fragment

```
<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="keyConverter">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.\
AuthenticationToStringConverter"/>
  </property>
  <property name="principalDao" ref="modelPrincipalDao" />
</bean>
```

## 11.8.2 JWT based principal lookup

When using token based authentication, you can derive the complete principal from the authentication token.

There is a special DAO to do this.

```
de.intarsys.cloudsuite.gears.security.spring.JwtPrincipalDao
```

### Example

## Spring XML fragment

```
<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>
```

No key conversion is applied.

## 11.9 Well known security beans

Just in case you feel tempted to tweak the built-in spring security definition, here are the well-known entry points. These beans can be overwritten in your own definition.

Keep in mind, that a security:http bean can not be overwritten or defined with an overlapping pattern. In this case, you have to add the profile “customSecurity” and rewrite a security definition from scratch.

Bean	Description
securityRealmControlAuthenticationManager	A spring <b>security:authentication-manager</b> definition for the “ <b>control</b> ” realm
securityRealmManageAuthenticationManager	A spring <b>security:authentication-manager</b> definition for the “ <b>manage</b> ” realm
securityRealmFlowAuthenticationManager	A spring <b>security:authentication-manager</b> definition for the “ <b>flow</b> ” realm
securityRealmControlAuthenticationFilter	A servlet filter that provides the request authentication information for “ <b>control</b> ” realm
securityRealmManageAuthenticationFilter	A servlet filter that provides the request authentication information for “ <b>manage</b> ” realm

securityRealmFlowAuthenticationFilter	A servlet filter that provides the request authentication information for “ <b>flow</b> ” realm
---------------------------------------	---



## 12. Authorization

### 12.1 Overview

Beginning with version 8.10.0, gears comes along with an authorization feature, based on the authentication described in the chapter before.

To enable authorization, spring security based authentication must be in place *and* the intarsys provided gateway to the ACL implementation must be in place.

```
<!--  
The gateway between spring security and intarsys aaa.  
  
This is required if you want to define additional ACLs (authorization)  
-->  
<bean id="authenticationProvider"  
class="de.intarsys.spring.security.SpringAuthenticationProvider" />
```

While authentication is about providing a proofed identity, authorization allows to control access to resources based on properties of the authenticated entity.

The distinction is a little bit blurry – our authentication above also contains access control on a very coarse level. You can restrict access to API methods based on the membership in a group. The authorization feature goes in a more detailed level, where the system can control (and log) the usage of important resources like

- Devices
- Configurations
- Direct arguments

### 12.2 Concepts

After authentication we have an authenticated principal available in the call context.

Depending on the authentication mechanism and the user repository, this principal is instrumented with properties and/or groups. These properties and groups (or roles) can be used to make a decision if an operation on a certain resource can be applied.

#### 12.2.1 Authentication

An authentication context is created as the result of the authentication process in the chapter before.

It is important to know that authorization is not supported without spring authentication in place.

#### 12.2.2 Authority

The authentication context comes with a set of authorities that describe the groups (or roles) the authenticated principal is decorated with.

When using spring-based authentication you can easily map the concept on the configuration.

The authority is used later to define if a certain principal is allowed to access a resource.

### 12.2.3 Resource

A resource may be anything in the system, typical resources are

- A device
- A configuration

It is an abstraction for all things that deserve to be under access control.

The resource is described by a type and an id. A device for example has type of "de.intarsys.security.device.IDevice", the id is the one configured for the device bean, e.g. "default@demo".

The id "\*" is a wildcard that designates all instances.

### 12.2.4 Operation

An operation can be executed on a resource, for example you can "sign" with a device.

There may be many operations for a resource. A "PersonDB" resource for example may support "create", "read", "update" and "delete" – a well-known and flexible concept.

### 12.2.5 Authorization strategy

The strategy that decides if an access is granted or denied is pluggable.

Some are useful only for testing and debugging, like

- de.intarsys.aaa.authorization.impl.DenyAuthorization  
Deny any access control request
- de.intarsys.aaa.authorization.impl.GrantAuthorization  
Grant any access control request

Useful strategies in a production environment

- de.intarsys.aaa.authorization.impl.AuthenticatedAuthorization  
Grant access if an authenticated principal is available
- de.intarsys.aaa.authorization.acl.AclAuthorization  
Grant access based on a detailed access control list

## 12.3 Integration (observation)

Based on the authorization strategy, you can observe and handle all authorization induced events (see chapter "Integration").

The observation raised has the following properties

source	
String	"authorization"
code	
String	The outcome of the access control check, either "failed" or "success"
resourceType	
String	The resource type that was checked
resourceId	
String	The resource id that was checked
operationId	

String	The operation that was checked
--------	--------------------------------

Example:

This example shows a spring configuration that creates a dedicated “authorization.log”.

First the observer is filtering “authorization” events via the “source” attribute.

For all these events the current user and conversation id are injected in the context.

Finally, the observation is logged using a dedicated simple pattern.

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="authorization" />
  <property name="view">
    <list>
      <bean class="de.intarsys.tools.observation.impl.IncludeStatement">
        <property name="pattern" value=".*" />
      </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="conversation" />
        <property name="value" value="{flow.id}" />
      </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}/authorization.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} %arg{code} %arg{user} %arg{conversation}
%arg{resourceType} %arg{resourceId} %arg{operationId}%n" />
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

## 12.4 ACL Authorization

### 12.4.1 Strategy

If authorization is in place, this is currently the most flexible and useful option.

With ACL (access control list) you describe in detail which resource is accessible for which authority.

The decision algorithm is like this:

- Lookup the access control list for the requested resource
- Filter all entries that match “operation” and “authority”
- If any “grant” is found, grant access
- If only “deny” is found, deny access
- If none is found
  - If “denyIfNotFound” is true
    - deny access
  - Else
    - grant access

This decision process is executed three times to simplify administration with wildcards:

- First check is done with “type:id”
- Second with “type:\*”
- Third with “\*:.”

Let’s examine in detail the important flag “denyIfNotFound”. There are two basic approaches to access control:

- Everything that is not granted explicitly is implicitly denied (corresponds to denyIfNotFound=true)
- Everything that is not denied explicitly is implicitly granted (corresponds to denyIfNotFound=false)

If you follow the first approach, your system is more “closed” and you have an explicit list of all things allowed. But it is harder to maintain – e.g., when a new gear release comes with a new resource type, you may first have to adapt your list to accommodate for the new feature.

The second approach does not show all resources that are accessible – anything not configured is granted. This eases administration but may leave holes in your security concept.

Wildcards come to the rescue to blur the lines between the two.

Example: This is equivalent to an empty ACL with “denyIfNotFound=true”.

```
<bean class="de.intarsys.aaa.authorization.acl.Deny">
  <property name="authority" value="*" />
  <property name="resource" value="*:." />
  <property name="operation" value="*" />
</bean>
```

Example: To completely control access to devices, even when “denyIfNotFound=false”.

```
<bean class="de.intarsys.aaa.authorization.acl.Deny">
  <property name="authority" value="*" />
  <property name="resource" value="de.intarsys.security.device.IDevice:*" />
  <property name="operation" value="*" />
</bean>
```

Using wildcards, you can set grant/deny defaults for a complete subset of devices.

## 12.4.2 Configuration

ACL based authorization is activated by both adding an “aclManager” and the appropriate “authorizationStrategy” in a spring configuration.

Now for any supported resource, ACL access control is in place.

```
<bean id="aclManager" class="de.intarsys.aaa.authorization.acl.SimpleAclManager">
</bean>

<bean id="authorizationStrategy"
class="de.intarsys.aaa.authorization.acl.AclAuthorization">
  <property name="denyIfNotFound" value="false" />
</bean>
```

For sure, you now have to define the rules that determine whether an access is granted or denied. This is done in the spring configuration, too. You create a list with tuples that define authority/resource pairs that are granted or denied.

The rules you add are either

- de.intarsys.aaa.authorization.acl.Grant  
Add an explicit grant to the described resource for the authorities

or

- `de.intarsys.aaa.authorization.acl.Deny`  
Add an explicit denial to the described resource for the authorities

The attributes you need are the same for both rules

authority	
String	<p>A “,” separated list of authorities that are matched with the authorities provided by the authenticated principal.</p> <p>An authority may be given as “*”, matching any authority provided by the principal.</p>
resource	
String	<p>A resource in “type:id” notation, see the description of supported resources below.</p> <p>The id may be given as a wildcard (resulting in “type:”), matching any resource of this type.</p> <p>The type may be given as a wildcard (resulting in “*:”), matching any resource.</p>
operation	
String	<p>An operation id, see the description of resources below.</p> <p>The id may be given as a wildcard (resulting in “”), matching any operation on the resource.</p>

Example: Grant the sign operation to the device “default@demo” for an authority “ROLE\_SIGN”.

```
<bean class="de.intarsys.aaa.authorization.acl.Grant">
  <property name="authority" value="ROLE_SIGN" />
  <property name="resource" value="de.intarsys.security.device.IDevice:default@demo" />
  <property name="operation" value="sign" />
</bean>
```

Example: Grant the sign operation to the device “default@demo” for any authority.

```
<bean class="de.intarsys.aaa.authorization.acl.Grant">
  <property name="authority" value="*" />
  <property name="resource" value="de.intarsys.security.device.IDevice:default@demo" />
  <property name="operation" value="sign" />
</bean>
```

Example: Deny the sign operation for any device for any authority.

```
<bean class="de.intarsys.aaa.authorization.acl.Deny">
  <property name="authority" value="*" />
  <property name="resource" value="de.intarsys.security.device.IDevice:*" />
  <property name="operation" value="sign" />
</bean>
```

## 12.4.3 Resources

### 12.4.3.1 Device

#### 12.4.3.1.1 Resource

A device can be a security relevant resource if there’s no additional authentication before key usage, as it is the case for e.g.

- Smartcard pools
- Seals

In these cases, you could restrict access to the “signer/create” service via authentication, but this may be too coarse when hosting different pools or seals. Device authorization is coming to rescue.

As usual, a device resource is described by a type

```
de.intarsys.security.device.IDevice
```

and an id, which is the id of the device you want to address, e.g.

```
default@demo
```

So, a correctly qualified device resource reads like

```
de.intarsys.security.device.IDevice:default@demo
```

Remember that you can use a wildcard instead of the id to address all devices.

```
de.intarsys.security.device.IDevice:*
```

### 12.4.3.1.2 Operations

For the device you can control the “sign” operation, Access is checked immediately before the signing takes place.

```
sign
```

### 12.4.3.1.3 Examples

These are some examples for device ACL configuration in spring

Example: Allow access to default@demo for the authority ROLE\_SIGN.

```
<bean class="de.intarsys.aaa.authorization.acl.Grant">
  <property name="authority" value="ROLE_SIGN" />
  <property name="resource" value="de.intarsys.security.device.IDevice:default@demo" />
  <property name="operation" value="sign" />
</bean>
```

Example: Deny access to all devices.

```
<bean class="de.intarsys.aaa.authorization.acl.Deny">
  <property name="authority" value="*" />
  <property name="resource" value="de.intarsys.security.device.IDevice:*" />
  <property name="operation" value="sign" />
</bean>
```

## 12.4.3.2 Service Arguments

### 12.4.3.2.1 Resource

It may not be evident at the first glimpse, but service arguments are a sensitive target.

gears security protocols allow very detailed addressing of low-level features of the respective devices. For some scenarios this can result in the fact that a client can request services that he is not intended to use. The simplest solution is to forbid client-side

arguments and to use server-side templates (the well-known configurations). Now the argument set is fix but still can be flexible by using the client-side variables.

Argument access is controlled via the “Service” resource type

---

Service

---

The id is the REST path to the service, e.g.

---

/v1/flow/signer/create

---

There are two services available under access control

- /v1/flow/signer/create
- /v1/flow/viewer/create

So, a correctly qualified resource reads like

---

Service:/v1/flow/signer/create

---

Remember that you can use a wildcard instead of the id to address all resources.

---

Service:\*

---

### 12.4.3.2.2 Operations

For the “Service” resource you can use the “args” operation, controlling permission to use an argument list.

---

args

---

Be aware that all paths for injecting arguments into the service are guarded:

- Direct service args (REST)
- Document specific args via service “tag” document properties
- Document specific args embedded in the document itself
- Indirectly added args like the “serviceCreate.args” argument to the SignAction in a viewer

### 12.4.3.2.3 Examples

These are some examples for service ACL configuration in spring

Example: Allow access to use args for signer creation for the authority ROLE\_SIGN.

---

```
<bean class="de.intarsys.aaa.authorization.acl.Grant">
  <property name="authority" value="ROLE_SIGN"/>
  <property name="resource" value="Service:/v1/flow/signer/create"/>
  <property name="operation" value="args"/>
</bean>
```

---

Example: Deny access to argument creation.

---

```
<bean class="de.intarsys.aaa.authorization.acl.Deny">
  <property name="authority" value="*" />
  <property name="resource" value="Service:*" />
  <property name="operation" value="args" />
</bean>
```

---

### 12.4.3.3 Configuration

#### 12.4.3.3.1 Resource

As stated in the chapter on service arguments, gears has very detailed access to sensitive parameters of its devices.

You can use configurations to hide clients from this complexity, but with regard to access control they serve another purpose. After restricting access to client-side arguments (see above), you can collect the required arguments in a configuration, optionally with variables. Now you have access to the rich gears features without direct access to sensitive stuff.

This leaves only the fact that you may have different configurations that are not intended to be used by anybody (e.g. two configurations targeting different seals or pools).

Now you can add access control to the configuration itself.

A configuration resource is described by a type

```
FlowSignerConfiguration
```

There are two well known configuration types:

- FlowSignerConfiguration
- FlowViewerConfiguration

The id is the id of the configuration you want to address, e.g.

```
demoPlain
```

So, a correctly qualified resource reads like

```
FlowSignerConfiguration:demoPlain
```

Remember that you can use a wildcard instead of the id to address all resources.

```
FlowSignerConfiguration:*
```

You can request a configuration in the service call via reference to an existing id as well as a literal object – this is why we have a special id “\_transient\_”. You must grant access to this virtual configuration id if you have access control enabled and need to send a literal configuration with your request.

```
FlowSignerConfiguration:_transient_
```

Be aware that sending literal configurations undermines the argument access control as you can add argument definitions to the configuration!

#### 12.4.3.3.2 Operations

For a configuration you can control the “execute” operation. “execute” controls the application of a configuration to a new flow via a “signer/create” or “viewer/create” call.



---

`execute`

---

### 12.4.3.3.3 Examples

These are some examples for configuration ACL in spring.

Example: Allow access to demoPlain for the authority ROLE\_SIGN.

---

```
<bean class="de.intarsys.aaa.authorization.acl.Grant">
  <property name="authority" value="ROLE SIGN"/>
  <property name="resource" value="FlowSignerConfiguration:demoPlain"/>
  <property name="operation" value="execute"/>
</bean>
```

---

Example: Deny access to all signer configurations.

---

```
<bean class="de.intarsys.aaa.authorization.acl.Deny">
  <property name="authority" value="*" />
  <property name="resource" value="FlowSignerConfiguration:*" />
  <property name="operation" value="execute" />
</bean>
```

---

## 13. Services

### 13.1 Overview

All operations and data structures relevant to service requests and responses are documented in the online OpenAPI document.

The service reference is available via "swagger".

The documentation is installed by default and is available at

**`http://<host>/<gears context>/apidoc/index.html`**

By default, the "swagger" UI is provided at "index.html".

You can forcibly switch to the "ReDoc" look and feel using "redoc.html".

The corresponding JSON and YAML descriptions are available at

**`http://<host>/<gears context>/api/openapi.[json|yaml]`**

### 13.2 API

The services are documented in OpenAPI and can be inspected using the swagger UI.

This is the authoritative description and guide.

### 13.3 Protocol

This chapter gives detailed advice how to handle the conversational protocol in your client.

#### 13.3.1 Flow creation

The "create" style methods all return "conversational responses". This means that you can't expect a direct result, but information on the state of the ongoing conversation.

The conversation snapshot returned **may** contain the result object if execution was synchronous, but it is far more likely that you receive some indication of what to do to drive the flow forward.

#### 13.3.2 Conversational response

A "conversational response" holds the conversation reference, together with a "stage", representing the current conversation state.

In its serialized form it looks like

## service response

```
{
  "snapshot": {
    "conversation": "hds-342d-er-453-fdsd-234",
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:HttpRedirect",
      "id": 222,
      "url": "https://service.intarsys.de/cloudsuite-gears/core/explorer?state=838383838",
      "outOfBand": false
    }
  }
}
```

The conversation represents the whole "workflow" that was created by calling the conversational API. The reply stage represents the current state of the workflow, reduced to the next task required to fulfill the communication between the service and the client.

The most important part is the "scheme" property, indicating the type of state for the conversation.

Well-known schemes are:

- urn:intarsys:names:conversation:1.0:schemes:Result
- urn:intarsys:names:conversation:1.0:schemes:Cancel
- urn:intarsys:names:conversation:1.0:schemes:Error
- urn:intarsys:names:conversation:1.0:schemes:Processing
- urn:intarsys:names:conversation:1.0:schemes:HttpRedirect

These schemes map to the data transfer objects used in the protocol:

- DtoResultStage
- DtoCancelStage
- DtoErrorStage
- DtoProcessingStage
- DtoHttpRedirectStage

### 13.3.3 "Final" stages

There are three final stages for a conversation:

- urn:intarsys:names:conversation:1.0:schemes:Result
- urn:intarsys:names:conversation:1.0:schemes:Cancel
- urn:intarsys:names:conversation:1.0:schemes:Error

After receiving a final stage, the conversation is unpublished immediately from the server – subsequent calls referencing this conversation id will fail.

A successful conversation will return a "Result" type object like this

---

**service response**

---

```
{
  "snapshot": {
    "conversation": "hds-342d-er-453-fdsd-234",
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Result",
      "id": 222,
      "result": {
        "@class": "de.intarsys.cloudsuite.gears.core.service.signer.api.ResultSigner",
        "value": {
          "documentNames": [ "mydoc.pdf" ],
          "signatures": [
            {
              "type": "d",
              "name": "mydoc.pdf",
              "content": "<base64 content>",
              "properties": {
                "signature": {
                  "targetName": "mydoc.pdf"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

The "result" property holds a typed wrapper object with a "@class" and "value" combination. The "value" then holds the serialized conversation result.

A failed conversation will return an "Error" type object

---

**service response**

---

```
{
  "snapshot": {
    "conversation": "hds-342d-er-453-fdsd-234",
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Error",
      "id": 222,
      "error": {
        "code": "ReallyBad-12",
        "message": "'foo' not supported"
      }
    }
  }
}
```

Last, a conversation could have been cancelled by some entity, for example if the user refuses authentication.

## service response

```

{
  "snapshot": {
    "conversation": "hds-342d-er-453-fdsd-234",
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Cancel",
      "id": 222
    }
  }
}

```

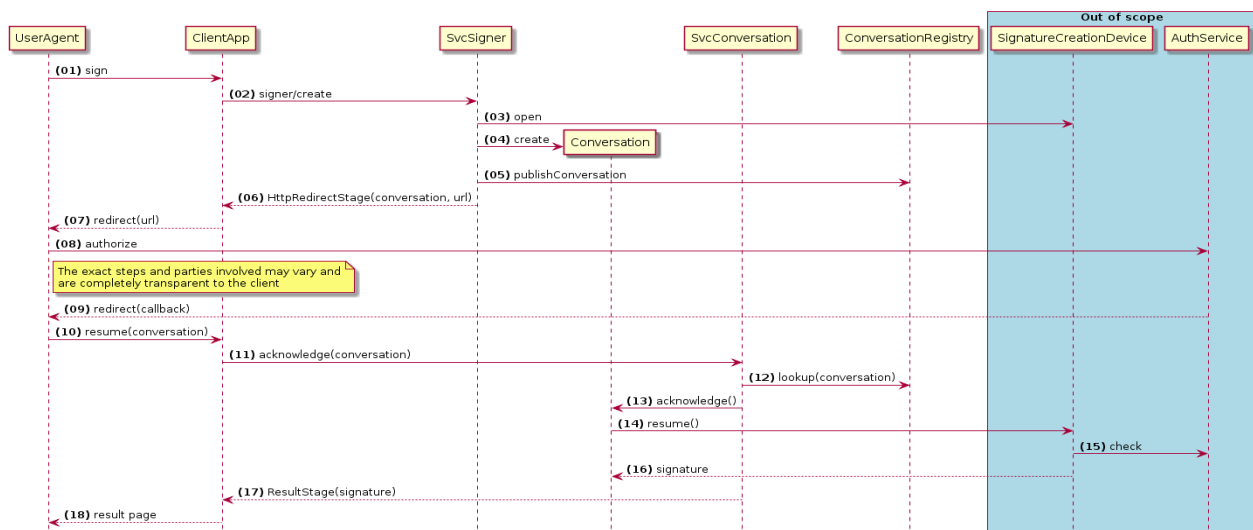
### 13.3.4 Multipage in-band

The interesting part starts when a conversation spans multiple steps and even web pages from different providers. The complex interactions between gears, service backends and authentication backends are encapsulated in

- urn:intarsys:names:conversation:1.0:schemes:Processing
- urn:intarsys:names:conversation:1.0:schemes:HttpRedirect

The simplest example is a service call that needs an in-band authorization. In this case you will be redirected to a consent gathering web page. After confirmation, your client resumes the conversation and will eventually get a final stage.

This is a (slightly simplified 😊) sequence diagram of an in-band authentication process



When initiating a multistep conversation (02), the return value shows the conversation and the reply stage valid at the moment of the flow creation, together with a URL to redirect to (06).

The client must take appropriate actions to handle the reply stage, in this case redirecting the browser to the page requested in the "url" property.

Control is now delegated to 3<sup>rd</sup> party systems and the exact steps following may vary. Important is that upon redirecting to your client redirectUri, gears ensures that the conversation, stage and outcome is injected, using the query parameters **cs\_conversation**, **cs\_stage** and **cs\_outcome** respective. The final redirect address looks like

---

```
https://my.domain.de/myapp/mypath?cs_conversation=xxx&cs_stage=yyy&cs_outcome=zzz
```

---

Your application now (11) collects the current state of the conversation using the special "acknowledge" service of the SvcConversation.

On one hand this resumes the process on the server side, on the other hand the client will receive the next stage in the process (17). Most often this will be a ResultStage, but nevertheless may be another HttpResponseRedirectStage.

To summarize, to handle this scenario regardless of inner complexity you need to

- create a flow (02)
  - handle the reply stage (06)
- acknowledge the conversation (11)
  - and handle the reply stage (17)

### 13.3.5 Redirect URI

As seen above, when initiating a flow, you may encounter a context switch to another application, eventually returning to your app. For this you need to provide an address that allows the browser to redirect to.

The directive to switch to another page is the "HttpRedirectStage" – it contains a URL that is to be visited next to continue the workflow.

If this step is in-band (the new page will redirect to your calling application afterwards), then you must supply a redirectUri. This is simply a URL indicating where to resume your application. This redirectUri is called after flow termination with additional query parameters,

- **cs\_conversation**
- **cs\_stage.**
- **cs\_outcome**

Your redirectUri:

---

```
http://myserver/mypath
```

---

Redirect from the gears flow

---

```
http://myserver/mypath?cs_conversation=xxx&cs_stage=yyy&cs_outcome=zzz
```

---

Now your application can request the current conversation stage to proceed using the cs\_conversation and cs\_stage id's.

The address itself must be provided in the redirectUri option when creating the flow.

Example request (recommended)

## service call

```

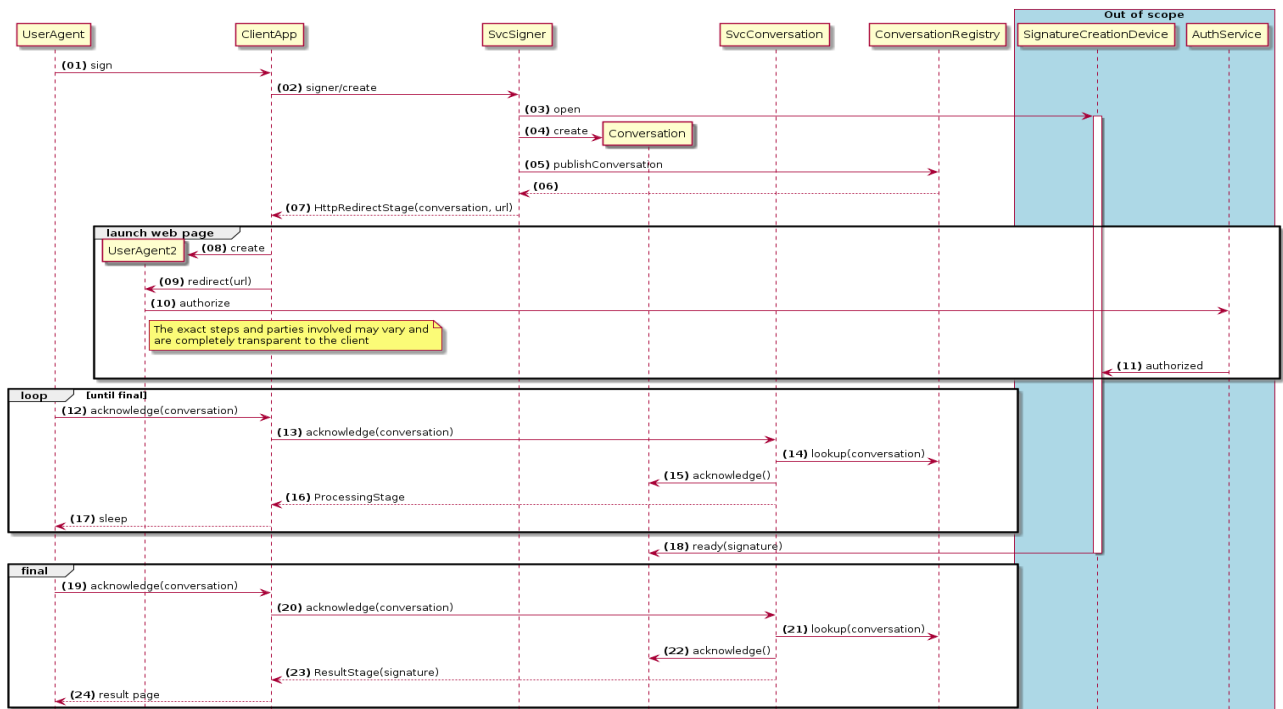
POST /cloudsuite-gears/core/api/v1/flow/signer/create HTTP/1.1
Content-Type: application/json

{
  "options": {
    "redirectUri": "http://myserver/mypath"
  },
  "documents": [{
    "type": "d",
    "name": "mydoc.txt",
    "content": "<base64 content>"
  },
  ...
]
}

```

### 13.3.6 Multipage out-of-band

The out-of-band case is very similar to the in-band case, but now you have to poll the SvcConversation as long as you receive the "ProcessingStage".



An out-of-band process does not necessarily start with a redirect, though. Imagine a SMS authentication process for example – in this case you would receive a ProcessingStage immediately while the server waits for an answer to the SMS.

A redirectUri is not used in either case (but you should provide one in the options anyway as you cannot necessarily know the installed authentication processes on the server – just in case).

### 13.3.7 Summary

So, let's summarize the steps how to drive a conversation with gears:

- 1) create a flow (e.g. via "signer/create") and do not forget the redirectUri option, just in case it is asynchronous in-band in its nature.

- 2) switch on the reply stage scheme
  - Result/Error/Cancel
    - You are ready. Display the outcome...
  - HttpResponseRedirect in-band
    - Drive the browser to the url and wait for the browser calling your redirectUri back
    - When the redirectUri is activated, call "acknowledge" with the "cs\_conversation" and "cs\_stage" query parameters.
    - goto 2)
  - HttpResponseRedirect out-of-band
    - Open new window and set it to the URL
    - call "acknowledge" with the conversation and stage from the HttpResponseRedirect
    - goto 2)
  - Processing
    - sleep a while
    - call "acknowledge" with the conversation and stage from the Processing stage
    - goto 2)

That's all.

You will find exactly this sequence implemented in the demo code that comes with this product.

## 13.4 Client helper

### 13.4.1 JavaScript Client

The JavaScript client helper is implemented in "csconversation-<version>.js". It is deployed in the Sign Live! cloud suite gears SDK.

### 13.4.2 Java Client

At the moment there's no standard Java client. The code to handle conversations is quite simple and an initial implementation can be looked up in the demo application.



## 13.5 Serialization issues

The serialization for this API holds surprisingly many challenges that must be managed to get a scalable and secure implementation.

We have made our best effort to get this done transparently for a client. Some topics still bubble up or are important enough to get mentioned here explicitly.

The base of the serialization implementation is Jackson, a widely used library for JAX-RS based applications.

### 13.5.1 Scaling in a Java environment

Plain out-of-the-box JSON unmarshalling is not scalable with the data model of the API, sending many or large documents will kill a server soon.

To circumvent this restriction while still keeping up the plain and simple API and protocol, we have tweaked the serialization to stream document content directly to or from external resources, like files or the repository.

The serialization part will stream bytes directly from your locator sources to the transport layer (for confidentiality be sure to use TLS connections). You should not have to tweak the default settings here.

The deserialization reads directly from the transport layer and writes to a target locator. The default implementation writes to a "ByteArrayLocator", holding the data completely in memory. This may be fine for most client applications.

In case you need to optimize this (as we did on the server-side), you have to establish your own locator creation strategy that returns another implementation than "ByteArrayLocator".

Besides establishing your own serialization implementation with your client framework, you have two options if you stay with our default implementation.

First, you can globally switch the "ITransportItemLocatorFactory" in "TransportDocumentLocatorDeserializer".

---

#### Java code fragment

```
TransportDocumentLocatorDeserializer.setLocatorFactory(myFactory);
```

---

where

---

#### Java code fragment

```
class MyFactory implements ITransportItemLocatorFactory {
    @Override
    public ILocator createLocator(JsonParser parser, DeserializationContext ctxt)
        throws IOException {
        return new MyLocator();
    }
}
```

---

If you need even more detailed access to the serialization process, you can attach this locator factory to the Jackson context.

## Java code fragment

```
ObjectMapper mapper = new ObjectMapper();
DeserializationConfig dc = mapper.getDeserializationConfig();
ContextAttributes attrs = ContextAttributes.getEmpty();
attrs = TransportDocumentLocatorDeserializer
    .setLocatorFactory(attrs, new RepositoryTransportItemLocatorFactory());
dc = dc.with(attrs);
mapper.setConfig(dc);
```

Our deserializer will try to detect a context specific factory first.

This is the way we stream data for transport directly to the repository on the server.

### 13.5.2 Confidentiality

The next challenge is confidentiality. Some scenarios require to store the data on the server encrypted only.

On one hand this completely defies the use of the standard JAX-RS stack (e.g. file upload), as these components always create clear text temporary files on the target side.

On the other hand, this requirement must not reduce service performance more than absolutely required. Creating encrypted temp files on the protocol layer and then re-encrypt and copy to the repository is unacceptable.

So, we have injected a special locator factory (see above) that streams and encrypts on the fly. The encryption key is completely random. Repository access is provided using a cryptographic indirection, known from techniques like CMS based encryption, using key wrapping.

The random key is encrypted with a well-known key for each potential recipient. The "system recipient", needing access to document data to act on it on behalf of the client uses a key derivation mechanism that allows both pure static secrets as well as adding key derivation material from the client.

You can find more on this topic in the API description and the configuration section.

### 13.5.3 Property order

As a direct consequence of the above, ordering of property serialization is important to make certain optimizations work.

When sending a document, you must serialize the properties in order:

1. type
2. name
3. properties
4. content/path/handle

This ordering is enforced by the server.

## 13.6 Error handling

### 13.6.1 Overview

A list of expected error codes can be found in the appendices.

### 13.6.2 ErrorDetail object

#### *ErrorDetail*

An object describing error conditions.

This object is used both in synchronous and conversational scenarios.

#### Properties

code	
string	A unique code for the error condition encountered
message	
string	A message with some descriptive information for the error codes

#### Example

```
{
  "code": "UniqueCode",
  "message": "some non NLS message",
}
```

### 13.6.3 ResponseError object

#### *ResponseError*

The response object in case of synchronous error conditions.

#### Properties

_error	
ErrorDetail	An ErrorDetail object describing the error for this request

### 13.6.4 Synchronous error handling

Synchronous error handling applies when you are calling non-conversational services. In the response you expect either directly the result or an error condition.

In this case, the HTTP status code is set accordingly to the error state.

- Client-side errors always result in **4xx** HTTP error codes.
- Server-side errors always result in **5xx** HTTP error codes.

In the HTTP payload you will find a JSON encoded "ResponseError".

Be aware that you can receive a "ResponseError" only if the service is executed at all, e.g. calling the service with a malformed URL will still result in a plain "404" from your service container.

## Example

```
{
  "_error": {
    "code": "unique code",
    "message": "some non NLS message",
  }
}
```

### 13.6.5 Conversational error handling

When calling conversational services, you can always expect a wrapped response, either for a result, an error or any other reply stage. So an error result is reported with an HTTP status 200 (because it is a valid conversational response). In the HTTP payload you find an `ErrorStage`

## Example

## service response

```
{
  "snapshot": {
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Error",
      "id": "2240",
      "error": {
        "code": "PoolResourceNotAvailable",
        "message": "Pool 'demoSmartcard' kein Gerät verfügbar"
      }
    },
    "conversation": "d2affb09-9c1a-45b3-9124-a277f9cce7e0"
  }
}
```

## 13.7 Request options

Request options are generic contextual information to the request, that are not directly related to the business function that is executed.

### 13.7.1 Redirect URI

You know by now that the gears protocol is asynchronous in its nature and may direct to (various) other web pages while processing the flow you initiated.

There is a detailed description of the general protocol flow above. Here we only repeat how to set a "redirectUri" option upfront, the recommended approach to handle asynchronous in-band flows.

## service call fragment

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

{
  "options": {
    "redirectUri": "http://myserver/mypath"
  },
  ...
}
```

The gears service will derive the correct complete URL for a later redirect to your application in the form

---

```
http://myserver/mypath?cs_conversation=xxx&cs_stage=yyy&cs_outcome=zzz
```

---

## 13.7.2 Restricted identification

"restrictedIdentification" allows to inject client defined execution context into a single request. This may be for example a user name or some derivation of it.

The "restrictedIdentification" property can then be used for example throughout the flow to ensure encryption of the document content on the server with a request specific key (see Repository encryption).

---

### service call fragment

---

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

{
  "options": {
    "restrictedIdentification": "foobar"
  },
  ...
}
```

---

## 13.7.3 Principal

You can provide a per-call dedicated principal for a flow. The principal can be mapped to any role using the configuration. For more information, see the chapter "Principals".

Principal via reference

---

### service call fragment

---

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

{
  "options": {
    "principal": "someId"
  },
  ...
}
```

---

Principal literal. You have to use the LiteralDao to accept this kind of principal.

---

**service call fragment**

---

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

{
  "options": {
    "principal": {
      "name": "someName",
      "claims": {
        "foo": "bar"
      }
    }
  },
  ...
}
```

---

### 13.7.4 Language

"lang" allows to select a preferred language for the flow execution. The language is used in all components that support language selection. This feature may not be supported for all devices (external TSPs).

The "lang" value must be a valid locale token for the active server VM.

---

**service call fragment**

---

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

{
  "options": {
    "lang": "en"
  },
  ...
}
```

---

## 14. Devices

---

Sign Live! cloud suite gears supports a variety of devices, each targeting the creation of digital signatures by different means. While some devices rely on the usage of soft- or hardware tokens managed by the application, others leverage signing services provided by external trust service providers (TSP).

The following chapters focus on the available signature devices and their details. They shall guide you through specific setup requirements and highlight typical usage patterns.

## 14.1 Overview

The device abstraction acts as a factory for algorithms and implementations that you use throughout all security applications.

To create such an application, you reference a well-known factory and provide the arguments required to fulfill the request.

The factory to be used for creating a security application may differ from device to device.

### 14.1.1 Device provider

The device provider is the factory for devices. It represents the ability to support a certain technical implementation of a gears feature.

Examples of device providers are

- The demo device provider. It can create devices that sign with on-the-fly created key pairs.
- The AIS device provider. It manages devices that interact with the Swisscom AIS remote signature server.

The device providers are registered automatically if the respective module is available when the server starts up.

To prevent activation of the device provider, you can either remove the respective module from the web application or you can explicitly disable the initialization using

#### Spring properties

```
deviceProviders.<id>.enabled=false
```

Depending on the device provider type there may be more property definitions you can use to fine tune the behavior. You find these settings in the respective device chapters.

The common property definitions for a device provider are

deviceProviders.<id>.enabled	
Boolean	Flag if this device will get initialized in the gears application
deviceProviders.<id>.policies.<application type>	
object	Configure general security application policies. You can find more information in chapter 14.1.5, Policy
deviceProviders.<id>.devices.<device id>	
object	Property definitions for individual devices. You can find more information in chapter 14.1.2, Device

### 14.1.2 Device

A device is a logical entity that encapsulates hardware or software that implements a gears feature, e.g., a signature algorithm.

The device represents a concrete hardware item or software account that can be used for performing the desired task. Examples of this are

- a demo device, equipped with a concrete key pair
- a smartcard device, consisting of a card terminal with card
- an AIS account, consisting of a server endpoint and account credentials.

The device is always created by the device provider. The concrete creation is documented in the respective chapters for the devices.

The device can be monitored using the health endpoint.



The common property definitions for a device are

deviceProviders.<provider id>.devices.<device id>.*	
object	There are no common property definitions. See the respective device chapters for more.

### 14.1.3 Application

The device is a factory for applications. Applications are created and used by gears to process a client request.

For example, a signature request is processed by

- preparing the document
- looking up the device provider and device
- create the appropriate application
- run the application
- take the result and embed in the document

### 14.1.4 Application "signer"

The typical application used in the scope of this document is a signer application, i.e. an implementation that is able to create a cryptographic signature for a hash value.

Such an application will be plugged into the **digestSigner** argument of a document signer, where the document signer creates the document specific hash and reassembles the signed document from the signature data structure (which is for example a PKCS#1 or CMS encoded result).

The **documentSigner** is again the argument to the "sign/create" service.

#### Factory

The most common and simple approach for a **digestSigner** will use the

**de.intarsys.security.app.signature.SignerFactory**

This is a generic implementation that requires the **device** itself and optional, device dependent arguments. All arguments beside "device" will be forwarded to the application creation step for the selected device.

There may be some devices that still require a special factory – this is documented with the device in the chapters below.

#### Arguments

These are arguments that are available for all signer applications of all devices.

device	
string required	The id of the device that will create the signer application. The id's of well-known devices are documented in the chapters below.
processingMessage	
string optional	An optional string that may be used for communicating state when a <b>ProcessingStage</b> is returned. The string is expanded and returned with each ProcessingStage from the signer.

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",
            "processingMessage": "chill, i'm in...",
            "arg0" : <arg0>,
            "argN" : <argN>
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}

```

### 14.1.5 Policy

An application created by the device is associated with a policy that controls the behavior of the application. This policy is attached by the device and created by the device provider.

Each application is controlled by a special policy, e.g. the signer application has a signer specific policy.

You can statically configure some of the policy behavior using property definitions. The property name is created using this template

**Spring properties**

```
deviceProviders.<device provider id>.policies.<application type>.*
```

**Example****Spring properties**

```
deviceProviders.pool.policies.signer.*
```

Please be aware that for some providers (especially the "pool") and some properties you must configure policy properties both for the device and the wrapped device as both are involved in the application process.

**Example****Spring properties**

```

deviceProviders.pool.policies.signer.validator.certificatePeriod.notAfter.action=ignore
deviceProviders.smartcard.policies.signer.validator.certificatePeriod.notAfter.action=ignore

```

### 14.1.6 Policy "signer"

The signer policy controls the signer application. The signer application type is "signer", so all property definitions look like this:

#### Spring properties

```
deviceProviders.pool.policies.signer.*
```

#### 14.1.6.1 Certificate period

The policy will check the certificate after creation of the application. The following checks are performed and can be fine tuned:

- If the certificate period has not yet started
- If the certificate period is already over
- How many days are left until the certificate expires

For every check you can define if a violation is ignored, a warning is created or an error is created.

A warning is logged, an error will terminate the security application. The "prefix" is the configuration property prefix described in the chapter above.

<prefix>.validator.certificatePeriod.notBefore.action	
string	ignore   warning   error The default is error, resulting in an error if the certificate is not yet valid.
<prefix>.validator.certificatePeriod.notAfter.action	
string	ignore   warning   error The default is error, resulting in an error if the certificate is expired.
<prefix>.validator.certificatePeriod.daysLeft.action	
string	ignore   warning   error The default is warning, resulting in a warning if the certificate is about to expire in the next "value" (see below) days
<prefix>.validator.certificatePeriod.daysLeft.value	
int	the number of days that must be left before the check triggers The default is 30 days.

### 14.1.7 Monitoring

Each device is available in the monitoring mechanisms of gears. Monitoring here means, that you can at least inspect the state of your application using standards based protocols. Some protocols (JMX) even allow active notification of the operator.

For more information you can see the chapter 16.3.4 dealing with spring actuator integration.

#### 14.1.7.1 Properties

Each device supports a set of properties and notifications that are published using the monitoring mechanism of choice. These properties and notifications are documented with each device.

This chapter summarizes the ones that are common to all devices.

enabled	
Boolean	flag if this device is enabled. A device is enabled by default
id	
string	The configured id for this device
licensed	
Boolean	flag if there is any valid license for this device.

state	
string	An internal state string available   unavailable
stateReason	
string	If the state is unavailable, this may provide a reason for the failure
type	
string	The implementation type of the device

#### 14.1.7.2 Observations

These are standard observations for the processing stages of a security application.

Depending on the device and application, we may have additional arguments. These are specified in the device specific chapters.

source	device.<provider>.<device>
	The identification of the device that originates the observation.  Example "device.signme.default"
code	
	A property which may resolve to a NLS specific message text.  Examples: <ul style="list-style-type: none"> <li>• "sign.ok"</li> <li>• "sign.signme.verificationProcess.ok"</li> </ul>
operation	
	A property which indicates the kind of security application Examples: <ul style="list-style-type: none"> <li>• "sign"</li> <li>• "authenticate"</li> </ul>
subcode	
	A property which indicates the state or outcome of the operation Examples: <ul style="list-style-type: none"> <li>• "start"</li> <li>• "ok"</li> <li>• "cancel"</li> <li>• "fail"</li> </ul>
app	
	The security application reference. You can collect further information from this object.

### 14.1.7.3 Background health check

Some devices support background a health check. This means that gears monitors the availability of the service used by the device, e.g. a remote API, an HSM or whatever. You can find more information on concrete checks in the device description.

The result of the health check is reflected in the device monitoring properties (health properties). This means that the device reports a “DOWN” state when the API is not reachable. The device goes back “UP” when the service appears again.

The health check is active by default and ensures that gears gets informed of service availability even if there’s currently no activity.

You can configure the health check using the “healthMonitor” property of the device. By default, the device selects an appropriate monitor and properties itself.

To overwrite the properties, simply provide a map as the “healthMonitor” value.

Example:

```
<entry key="healthMonitor">
  <map>
    <entry key="delay" value="42" />
  </map>
</entry>
```

This overwrites the “delay” between two attempts to receive the backend health state.

To set a dedicated monitor (which is normally not required)

Example:

```
<entry key="healthMonitor">
  <value>
    <bean class="de.intarsys.security.device.health.NoopHealthMonitor"/>
  </value>
</entry>
```

#### 14.1.7.3.1 All background health monitors

##### Common properties for background health monitors

delay	
Integer	The delay in seconds between two attempts to reach the backend of the device. Be aware the this may produce significant traffic if too low and that a service provider may put limits on the load.

#### 14.1.7.3.2 de.intarsys.security.device.health.NoopHealthMonitor

This health monitor disables all checks

Example:

```
<entry key="healthMonitor">
  <bean class="de.intarsys.security.device.health.NoopHealthMonitor"/>
</entry>
```

- no properties	

#### 14.1.7.3.3 de.intarsys.security.device.health.ConnectHealthMonitor

This health monitor tries to connect to a host:port. If the connection succeeds, it is closed and the backend is “UP”.

Example:

```
<entry key="healthMonitor">
  <bean class="de.intarsys.security.device.health.ConnectHealthMonitor">
    <property name="host" value="google.com" />
    <property name="port" value="80" />
    <property name="timeout" value="1000" />
    <property name="delay" value="100" />
  </bean>
</entry>
```

host	
string	A host name or IP address
port	
Integer	A TCP port
timeout	
Integer	Timeout in milliseconds for the connect attempt
delay	
Integer	Time in seconds between two connection attempts

#### 14.1.7.3.4 de.intarsys.security.device.health.ActuatorHealthMonitor

This health monitor tries to derive device health from a spring actuator health endpoint.

Example:

```
<entry key="healthMonitor">
  <bean class="de.intarsys.security.device.health.ActuatorHealthMonitor">
    <property name="delay" value="100" />
  </bean>
</entry>
```

path	
string	A path to the health endpoint. If this is relative, the path is interpreted using the device base URL.  The default is “manage/health”
jsonPtr	
String	Optional path into the health result structure if only a sub-node is to be used.  Example This JSON pointer selects the information for the default@signme device on the server  <code>/components/deviceProviders/components/signMe/details/devices/default</code>  Default empty
delay	
Integer	Time in seconds between two connection attempts

## 14.2 Demo Device

### 14.2.1 Overview

The demo device is a versatile prototyping and test tool.

Basically, it provides an RSA key pair and a self-signed certificate. The key pair and certificate are created with the first start and stored in your `$(cloudsuite.data.shared)` directory. By default, the id is "demo".

Now the device can be used for signing...

## 14.2.2 Device Configuration

### 14.2.2.1 Device provider

A device is created by the respective device provider. The demo device provider is registered with the Spring bean id **deviceProvider.demo**.

The id of the device provider is **demo**.

A concrete demo device can be addressed as **<device id>@demo**.

There is no configuration you can apply to the device provider itself.

### 14.2.2.2 Device properties

id	
string required	The id of the new device. Only ASCII-alphanumeric, '_' and '-' is allowed.

### 14.2.2.3 Default device

The system comes with two preconfigured devices, which can be addressed as **default@demo** and **tsa@demo**. While the default device is appropriate for document signing, the tsa device is meant to be used in the context of timestamp creation.

### 14.2.2.4 Custom device

Currently you cannot create custom demo devices.

## 14.2.3 Usage

### Factory

For signature creation, use the generic **de.intarsys.security.app.signature.SignerFactory**.

### Arguments

device	
string required	The id of the device that will create the signer application. For the preconfigured demo device, you should refer to "default@demo".

## 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",
    "content": "QUJDLi4u"
  }]
}
```

## 14.2.4 Signer Configuration

To further abstract from a concrete device and concrete arguments, remember you can invent a "signer configuration" (see 9.11.1.1).

The gears demo environment comes with a predefined signer configuration "demoPlain":

## Spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="demoPlain" />
  <property name="defaultValue" value="true" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
value="de.intarsys.security.processor.signature.DocumentSignerFactory" />
      <entry key="documentSigner.args.digestSigner.factory"
value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device" value="default@demo" />
      <entry key="documentSigner.args.timestampDevice" value="default@tsa" />
      <entry key="documentSigner.args.field.adjustForRotate" value="true" />
      <entry key="documentSigner.args.signatureLabel"
value="\${nlsmsg.de.intarsys.gears.core.demo.messages#documentSigner.signatureLabel}" />
    </map>
  </property>
</bean>
```

Instead of the "usage" example you can leverage this configuration:



---

**service call**

---

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

```
{
  "configuration": "demoPlain",
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}
```

### 14.2.5 Monitoring

This device supports the standard device properties and notifications.

It does not require background health check

### 14.2.6 Observations

This device supports the standard observations.

## 14.3 Bridge Device

### 14.3.1 Overview

The bridge device allows access to a local resource connected to your user's client computer, like a smartcard, a local token or the windows crypto API.

To achieve this, the client computer must be instrumented with a local process that is able to "bridge" the requests from the browser to the local resource. This process is called the "bridge agent".

Now that Java Applets and WebStart have both been discontinued there's no more official support for "zero configuration" deployment of this software component. Your client machine will need to have a basic "bridge application" pre-installed to use the bridge device features.

The bridge agent is available as a separate software component from intarsys.

### 14.3.2 gears configuration

This use case has some components integrated in the gears server, allowing the **browser** to communicate with the **bridge agent via the server**. This circumvents the browser sandbox restrictions and simplifies deployment in terminal server environments.

#### 14.3.2.1 Meeting factory

The first configurable component is the "meeting factory". For each flow that uses the bridge feature, a "meeting" is arranged between the participants "browser", "gears" and "bridge". To defend against (network) failures, a timeout is established to ensure graceful failure when a participant does not show up. The default timeout is 60 seconds – if a participant is not available at the meeting point, the process will fail. You can set this timeout via the property.

---

```
meeting.participantTimeout
```

---

#### 14.3.2.2 Message channel

The second configurable component is the "message channel" between the meeting point and the participants. Communication is based on a long-poll protocol. In this scenario very often intermediate networking devices like proxies, firewalls or routers close a connection after a certain time of inactivity. To keep these devices happy, a keep alive mechanism is built-in that ensures that at least every 50 seconds (default value) some messages are exchanged between the participants. Depending on the network device configurations you may need to tweak this setting via the property

---

```
meeting.messageChannel.keepAlive
```

---

### 14.3.3 Device Configuration

#### 14.3.3.1 Device provider

A device is created by the respective device provider. The bridge device provider is registered with the Spring bean id **deviceProvider.bridge**.

The id of the device provider is **bridge**.

A concrete bridge device can be addressed as **<device id>@bridge**.

There is no configuration you can apply to the device provider itself.

### 14.3.3.2 Device properties

id	
string required	The id of the new device. Only ASCII-alphanumeric, '_' and '-' is allowed.
agentLicense	
file name optional	An optional name for a resource containing the bridge agent license used by this bridge device. If no license is defined, a default fallback is looked up.
urlUi	
URL optional	An optional path to the location of the bridge device UI. If this is not absolute, it is interpreted relative to the gears server context path. The default is "bridge".

### 14.3.3.3 Default device

The system comes with a preconfigured bridge device "default", which can be addressed as **default@bridge**.

This is a standard suitable for most use cases.

### 14.3.3.4 License

The bridge agent (the client process) requires a license argument for performing certain tasks. This license must be available for the gears server to be forwarded to the bridge agent later on. The license is looked up in the following locations

- the **license** argument to the digest signer
- the license file referenced by the **agentLicense** property of the device configuration.
- any license for the product **de.intarsys.stage.product.bridge** that is deployed within the **licenses** directory of the gears configuration.
- the builtin demo license. This license allows working with web applications from "localhost" only.

For production use you must acquire a license and deploy it using one of the mechanisms described above.

### 14.3.3.5 Custom device

A custom device can be created using the device provider (e.g. **deviceProvider.bridge**) as a factory.

The factory method is "createInstance", all supported bean definitions are enumerated in a map argument.

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

#### Spring XML fragment

```
<bean factory-bean="deviceProvider.bridge" factory-method="createInstance">
  <constructor-arg>
    <map>
      <entry key="<key>" value="<value>" />
    </map>
  </constructor-arg>
</bean>
```

### 14.3.3.5.1 Example

Create a device "mydevice@bridge" with a dedicated license file.

## Spring XML fragment

```
<bean factory-bean="deviceProvider.bridge" factory-method="createInstance">
  <constructor-arg>
    <map>
      <entry
        key="id"
        value="mydevice" />
      <entry
        key="agentLicense"
        value="${cloudsuite.config.shared}/my-license.lic" />
    </map>
  </constructor-arg>
</bean>
```

### 14.3.4 Usage

#### Factory

For signature creation, use the generic **de.intarsys.security.app.signature.SignerFactory**.

#### Arguments

device	
string required	The id of the device that will create the signer application. For the preconfigured bridge device, you should refer to "default@bridge".
bridge	
object	<p>Here you can provide any arguments for parameterizing the session to the bridge agent itself (the connect arguments). Detailed documentation for the "bridge" and "bridglet" argument can be found in the documentation set of the cloud suite SDK.</p> <p>Default {}</p> <p>Example</p> <pre>{   "license": "&lt;base64&gt;" }</pre>
bridglet.factory	
string	<p>You can override the bridglet to be used for signing.</p> <p>Default bridglet.factory= de.intarsys.cloudsuite.bridge.control.workflow.MeetingSignatureBridglet</p>
bridglet.args	
object	<p>Here you can provide any supported argument of the bridglet. Detailed documentation for the "bridge" and "bridglet" argument can be found in the documentation set of the cloud suite SDK.</p> <p>Default {}</p> <p>Example</p> <pre>{   "ui": {     "reduced": "true"   } }</pre>

## 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"
              }
            }
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}

```

### 14.3.5 Signer Configuration

To further abstract from a concrete device and concrete arguments, remember you can invent a "signer configuration" (see 9.11.1.1).

The gears demo environment comes with a predefined signer configuration "demoBridge":

## Spring XML fragment

```

<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="demoBridge" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
value="de.intarsys.security.processor.signature.DocumentSignerFactory" />
      <entry key="documentSigner.args.digestSigner.factory"
value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device" value="default@bridge" />
      <entry key="documentSigner.args.timestampDevice" value="default@tsa" />
      <entry key="documentSigner.args.field.adjustForRotate" value="true" />
      <entry key="documentSigner.args.signatureLabel"

value="\${nlsmsg.de.intarsys.gears.core.demo.messages#documentSigner.signatureLabel}" />
    </map>
  </property>
</bean>

```

Instead of the "usage" example you can leverage this configuration:

**service call**

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

{
  "configuration": "demoBridge",
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}
```

### 14.3.6 Monitoring

This device supports the standard device properties and notifications.

It does not require background health check

### 14.3.7 Observations

This device supports the standard observations.

### 14.3.8 Testing

This component has a certain complexity – implementation is distributed between three entities that each may suffer failures.

Here are some hints how to survive

#### 14.3.8.1 Bridge installation

Ensure the bridge is installed and has the correct version by simply starting it from commandline or windows start menu.

After the start, you should find a log file in your users directory (within `“./CSBridge_<version>”`). The log file gives detailed information about startup success.

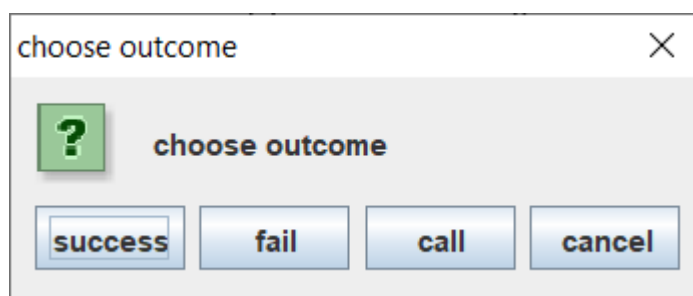
#### 14.3.8.2 Protocol handler

A protocol handler must be installed to enable the browser to trigger the bridge functionality.

You can check this behavior by typing the following (in a single line) in the browser address line.

```
csbridge:///app?noautostart=&enter=factory=de.intarsys.bridge.launch.local.LocalBridgletLauncher;
args.bridglet.factory=de.intarsys.cloudsuite.bridge.control.common.DebugBridglet
```

An internal debug window should come up like this



This shows that the protocol handler communication is working.

### 14.3.8.3 Gears communication

You can even ensure the communication to the gears platform is working:

```
csbridge:///app?noautostart=&enter=factory=MeetingBridgletLauncher;  
args.hostUrl=http://localhost:8080/cloudsuite-gears/core/;  
args.state=436d96ea-3949-47cc-8bae-8aa039fc7d90;
```

This should result in an error message from the server, indicating the requested meeting is not found.

## 14.4 Smartcard Device

### 14.4.1 Overview

The smartcard device is directly related to the handling of card terminals and smartcards supporting cryptographic signatures.

The implementation uses the PC/SC standard.

### 14.4.2 Device Configuration

#### 14.4.2.1 Device provider

A device is created by the respective device provider. The default smartcard device provider is registered with the Spring bean id **deviceProvider.smartcard**.

The id of the device provider is **smartcard**.

A concrete smartcard device can be addressed as **<device id>@smartcard**.

The device provider supports the following property definitions.

deviceProviders.smartcard.allowSecretFromCache	
Boolean	<p>This setting controls if the application accepts secrets (PIN) from a local cache. You need to set this to "true" if you want to drive a smartcard pool that automatically authenticates upon card insertion or gears startup.</p> <p>Be warned that this flag needs careful design of process and execution environment to ensure a secure signature process. Some signature creation devices (e.g. certain smartcards) require that the PIN is entered by the key holder to explicitly signal consent to the key usage.</p> <p>The default is "false"</p>
deviceProviders.smartcard.allowSecretFromApi	
Boolean	<p>This setting controls if the application accepts secrets (PIN) from a calling application via arguments.</p> <p>Be warned that this flag needs careful design of process and execution environment to ensure a secure signature process. Some signature creation devices (e.g. certain smartcards) require that the PIN is entered by the key holder to explicitly signal consent to the key usage.</p> <p>The default is "false"</p>

#### 14.4.2.2 Devices

The smartcard devices are not preconfigured, but automatically created based on PC/SC meta information.

For every card terminal registered with PC/SC, a smartcard device is created.

There are no property definitions for smartcard devices.

### 14.4.3 Usage

The smartcard device cannot be used directly at the moment.

### 14.4.4 Monitoring

This device supports the standard device properties and notifications.



It does not require background health check

#### 14.4.5 Observations

This device supports the standard observations.

## 14.5 Pool Device

### 14.5.1 Overview

A pool device allows to share pre-authenticated security applications to be used in gears requests from your client.

Most often the pool is based on one or more physical smartcards with a mass or batch signature feature, creating a "smartcard HSM" without the price.

### 14.5.2 Device authentication

For a pool device in a server application, authentication is an especially problematic topic. At some indeterminate moment in time (e.g., on startup or upon availability of a smartcard), the system has to provide a secret (PIN) to add the authenticated security application to the pool.

The options for providing a PIN include:

- The pooled device has PIN entry mechanics of its own, like a card terminal with a key pad. Immediately after application creation, authentication takes place. If authentication fails, the security application is disposed. The system will not retry until restart or re-insertion of the card.
- There's no support to set the PIN via call arguments. This would collide with the asynchronous, shared nature of a pooled device.
- The pooled device has its PIN caching feature enabled and the PIN was set & cached from the gears control console. See the chapter for the respective device if PIN caching is supported and how to enable it.

### 14.5.3 Device Configuration

#### 14.5.3.1 Device provider

A device is created by the respective device provider. The default pool device provider is registered with the Spring bean id **deviceProvider.pool**.

The id of the device provider is **pool**.

A concrete pool device can be addressed as **<device id>@pool**.

There is no configuration you can apply to the device provider itself.

#### 14.5.3.2 Device properties

id	
string required	The id of the new device. Only ASCII-alphanumeric, '_' and '-' is allowed.
autostart	
Boolean	Flag if this pool is automatically started when the application is started. The default is "true"
certificateFilter	
Object	Any object that can be converted to an IX509CertificateFilter. This filter is used for selecting the keys on the smartcard that will get authenticated and provided in the pool. If there are multiple suitable keys on a smartcard and no filter is given, the smartcard is rejected. The default is empty.
deviceProvider	
PoolDeviceProvider	The device provider to be used for creating the pool device. If this is empty, the first (and most of the time, the only) pool device provider is selected.
disableSPE	

Boolean	Flag if for this pool secure pin entry is disabled. Default is false
maxActiveEntryCount	
integer	The maximum number of active (in use) security applications for the pool, -1 for unrestricted. You can for example have 2 smartcards attached to your pool for failover reasons, but only one of them may be active in parallel. This property is bound by license restrictions. Default is -1.
maxEntryCount	
integer	The maximum number of authenticated security applications waiting in the pool, -1 for unrestricted. This is for example the total number of smartcards you can attach to a pool. Default is -1.
maxUptime	
integer	The maximum lifetime of a security application after added to the pool in minutes, -1 for unrestricted. After this time, the application is automatically de-authenticated and disposed. Default is -1.
maxUsageCount	
integer	The maximum number of uses for a security application in the pool, -1 for unrestricted. After this, the application is automatically de-authenticated and disposed. Default is -1.
notificationPercentage	
integer	A percentage for creating notifications. After reaching this percentage either of "maxUptime" or "maxUsageCount" a notification event is created. The default is 10%.
timeout	
integer	The maximum number of milliseconds to wait for a security application to be available from the pool, -1 for unrestricted. If no security application is available after timeout, an exception is raised. The default is -1.

#### 14.5.3.3 Default device

The system comes along with a preconfigured pool device "smartcard", which can be addressed as **smartcard@pool**. As this device would be greedy with respect to any card terminal and smartcard, creation of this device is guarded by a special Spring profile "smartcardPool". To enable the device, add a line like this to your gears.properties

#### Spring properties

```
spring.profiles.active=smartcardPool
```

#### 14.5.3.4 License

For use you must acquire a license. The license file can be copied to the systems license directory (e.g. \${cloudsuite.config.shared}/licenses) and will be picked up from the device dynamically.

There are two device property definitions that are affected by a license:

- The total number of pools supported (default = 0)
- The total number of active applications per pool (default = 1)

You cannot create a pool device without license and, by default, you cannot have more than one active application in the pool (no parallel signing on multiple cards).

#### 14.5.3.5 Custom device

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

Using the **de.intarsys.security.device.pool.smartcard.SmartcardPoolDeviceFactoryBean**, the task of creating a new smartcard pool is greatly simplified.

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

#### Spring XML fragment

```
<bean
class="de.intarsys.security.device.pool.smartcard.SmartcardPoolDeviceFactoryBean">
  <property name="id" value="mypool" />
  <property name="autostart" value="true" />
  <property name="certificateFilter">
    <bean class="de.intarsys.security.certificate.filter.standard.X509CertificateSelector"
      factory-method="createNonRepudiation">
    </bean>
  </property>
</bean>
```

### 14.5.4 Usage

#### Factory

For signature creation, use the generic **de.intarsys.security.app.signature.SignerFactory**.

#### Arguments

device	
string required	The id of the device that will create the signer application. For the preconfigured smartcard pool device, you should refer to "smartcard@pool".

## 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": "smartcard@pool"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}
```

## 14.5.5 Signer Configuration

To further abstract from a concrete device and concrete arguments, remember you can invent a "signer configuration" (see 9.11.1.1).

The gears demo environment comes with a predefined signer configuration "demoSmartcardPool". Instead of the "usage" example you can leverage this configuration:

## service call

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

{
  "configuration": "demoSmartcardPool",
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}
```

## 14.5.6 Monitoring

This device has some additional properties to inspect the pool state.

It does not require background health check

### 14.5.6.1 Properties

activeEntryCount	
integer	The number of currently active registered devices. These are devices that are currently used in a security operation
entryCount	
integer	The number of potential registered devices, e.g. the number of smartcard terminals attached to the pool
licensedActiveEntryCount	

integer	The number of licensed active devices. This controls the licensed concurrency level of the pool.
maxEntryCount	
integer	The maximum of registered devices this pool will support. See device configuration for more information.
maxActiveEntryCount	
integer	The maximum of active devices this pool will support. This controls the configured concurrency level of the pool. See device configuration.
pending	
integer	The number of pending requests to the pool.
runState	
string	A string indicating the run-state of the pool which is one of stopped   started   suspended

#### 14.5.6.2 Notifications

pool.started	
	Sent after a pool has started.
pool.stopped	
	Sent after a pool has stopped.
pool.resumed	
	Sent after a pool has resumed.
pool.suspended	
	Sent after a pool has been suspended.
pool.entryAdded	
	A new entry (smartcard) has been added to the pool.
pool.entryRemoved	
	An existing entry has been removed from the pool. This can be caused by an expected event (e.g. card removal) or a smartcard application failure.
pool.resourceNotAvailable	
	A security application was requested from the pool (e.g. a smartcard signer) but the request could not be fulfilled within the timeout. If the timeout is set to "-1", this event will never be sent because the request will wait indefinitely for a resource.
certificate.weakSignatureAlgorithm	
	The certificate used for signing is itself signed with a weak algorithm. This may enable using forged certificates.
certificate.willExpire	
	The certificate requested for signing will expire in the next 30 days.
certificate.expired	
	The certificate requested for signing has already expired.
certificate.notYetValid	
	The certificate requested for signing is not yet valid.
application.authenticationRequired	
	A security application needs authentication.
application.authenticationFinished	
	An authentication attempt succeeded
application.authenticationFailed	
	An authentication attempt failed.
application.failed	
	Performing the applications operations have failed.

### 14.5.7 Observations

This device supports the standard observations.

## 14.6 Swisscom All-in Signing Service Device

### 14.6.1 Overview

With the All-in Signing (AIS) Service, Swisscom provides a trust service in accordance with the requirements of the EU's eIDAS regulation and the Swiss signature law (ZertES). The services offered comprise the remote creation of

- electronic signatures,
- electronic seals and
- qualified timestamps.

Sign Live! cloud suite gears currently supports the usage of signature and seal creation services.

To get access and make use of these services (or parts of them), you have to register with Swisscom. During the registration process, you will

- provide administrative information to Swisscom,
- generate TLS client credentials,
- send the corresponding certificate to Swisscom,
- get authorized by whitelisting your certificate and
- receive access information which can be used to request the services.

Please see [6] for details on how to create a suitable TLS client certificate and for detailed information on the TSP service.

Once you have successfully registered, you can configure Sign Live! cloud suite gears for your account. See chapter 14.6.2 for details on how to set up access.

Depending on the requested services, different usage patterns apply. See chapter 14.6.5 for an in-depth view of the typical processes.

### 14.6.2 Device Configuration

There exist standard bean definitions to access the AIS service, so you can set up everything in your custom property definitions. The following properties are available:

ais.serviceAddress	
URL optional	The URL of the service endpoint accepting OASIS-DSS requests over SOAP/HTTP. Default value: <a href="https://ais.swisscom.com/AIS-Server/ws">https://ais.swisscom.com/AIS-Server/ws</a>
ais.tls.keyStoreType	
string required	The type of key store containing the TLS client credentials. Possible values: PKCS12   JKS
ais.tls.keyStoreName	
string required	Absolute path of the key store file containing the TLS client credentials.
ais.tls.keyStorePassword	
string required	Masked password for the key store file containing the TLS client credentials.
ais.tls.keyPassword	
string required	Masked password for the key for the TLS client credentials.
ais.customerName	

string required	The customer's name provided by Swisscom as a result of the registration process.
--------------------	---

The following properties are deprecated and should no longer be used

ais.clientKeyStoreType (deprecated, use ais.tls.keyStoreType)	
string required	The type of key store containing the TLS client credentials. Possible values: PKCS12   JKS
ais.clientKeyStorePath (deprecated, use ais.tls.keyStoreName)	
string required	Absolute path of the key store file containing the TLS client credentials.
ais.clientKeyStorePassword (deprecated, use ais.tls.keyStorePassword)	
string required	Plain-text password for the key store file containing the TLS client credentials.
ais.clientKeyPassword (deprecated, use ais.tls.keyPassword)	
string required	Plain-text password for the key for the TLS client credentials.
ais.healthMonitor.delay	
integer optional	Number of seconds between two health checks of the AIS DSS service. The default is 300, which amounts to 5 minutes. A value less than 1 disables health monitoring.

A properties file could look like the following:

### Spring properties

```
ais.serviceAddress=https://ais.swisscom.com/AIS-Server/ws
ais.tls.keyStoreType=PKCS12
ais.tls.keyStoreName=${cloudsuite.config.shared}/ssl/Demo ELDIV, Swisscom (Schweiz)
AG.p12
ais.tls.keyStorePassword=myPassword
ais.tls.keyPassword=myPassword
ais.customerName=AIS-Demo
```

## 14.6.3 Signer Configuration

Recurring configuration semantics can be predefined and reused through the definition of dedicated signer configurations (see chapter 9.11.1). In order to ease the setup of the most common use cases, Sign Live! cloud suite gears comes equipped with two default AIS signer configurations: 'aisOnDemand' and 'aisStatic'.

### 14.6.3.1 Default signer configuration "aisOnDemand"

This configuration defines a template for signatures based on on-demand certificates. Applying this configuration will result in an LTV-enabled signature incl. complete validation data and a signature timestamp. It uses the most important arguments of the signature device.



## Spring XML fragment

```

<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="aisOnDemand" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
value="de.intarsys.security.processor.signature.DocumentSignerFactory" />
      <entry key="documentSigner.args.signatureLabel" value="\${ais.ondemand.text:}" />
      <entry key="documentSigner.args.conformanceLevel"
value="\${ais.ondemand.conformanceLevel:LT}" />
      <entry key="documentSigner.args.digestSigner.factory"
value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device" value="default@ais" />
      <entry key="documentSigner.args.digestSigner.args.keyEntity"
value="\${ais.ondemand.keyEntity:}" />
      <entry key="documentSigner.args.digestSigner.args.mode" value="onDemand" />
      <entry key="documentSigner.args.digestSigner.args.distinguishedName"
value="\${ais.ondemand.distinguishedName:}" />
      <entry key="documentSigner.args.digestSigner.args.stepUpMSISDN"
value="\${ais.ondemand.stepUpMSISDN:}" />
      <entry key="documentSigner.args.digestSigner.args.stepUpMessage"
value="\${ais.ondemand.stepUpMessage:}" />
      <entry key="documentSigner.args.digestSigner.args.stepUpMessageMaxLength"
value="\${ais.ondemand.stepUpMessageMaxLength:}" />
      <entry key="documentSigner.args.digestSigner.args.stepUpMessageMaxLengthNonGSM"
value="\${ais.ondemand.stepUpMessageMaxLengthNonGSM:}" />
      <entry key="documentSigner.args.digestSigner.args.stepUpLanguage"
value="\${ais.ondemand.stepUpLanguage:en}" />
      <entry key="documentSigner.args.digestSigner.args.rasAssuranceLevel"
value="\${ais.ondemand.rasAssuranceLevel:4}" />
      <entry key="documentSigner.args.digestSigner.args.rasJurisdiction"
value="\${ais.ondemand.rasJurisdiction:#{null}}" />
    </map>
  </property>
</bean>

```

This default configuration can be customized through the use of the following custom property definitions

ais.ondemand.conformanceLevel	
string optional	The conformance level to achieve. Supported values are: <ul style="list-style-type: none"> <li>• B (for Basic Signature)</li> <li>• T (for Signature with Time)</li> <li>• LT (for Signature with Time and Long-Term validation data)</li> </ul> Default value: LT
ais.ondemand.keyEntity	
string required	The name of the key entity used as provided by Swisscom with your account information.
ais.ondemand.distinguishedName	
string required	The subject distinguished name to encode into the certificate. Please make sure the name matches the DN pattern set at Swisscom.
ais.ondemand.text	
string optional	A text displayed in the signature field (in case of a PDF signature). Default value: <empty>
ais.ondemand.stepUpMSISDN	
string optional	The phone number used to authorize the signature creation. Can be omitted if no two factor authentication is required.
ais.ondemand.stepUpMessage	
string required	A text message displayed to the signatory on the consent screen.
ais.ondemand.stepUpMessageMaxLength	
integer optional	The maximum length of the consent text message after variable expansion. Messages longer than that will be cut off. The default of 239 is the maximum number of

	characters that Mobile ID on the AIS side can handle at the time of writing. A value of -1 disables the feature. Default value: 239
<b>ais.ondemand.stepUpMessageMaxLengthNonGSM</b>	
integer optional	The maximum length of the consent text message after variable expansion for messages with characters outside of the GSM 03.38 character set. Messages longer than that will be cut off. The default of 119 is the maximum number of characters that Mobile ID on the AIS side can handle at the time of writing in case of non-GSM character use. A value of -1 disables the feature. If disabled or greater than stepUpMessageMaxLength, the maximum length is initialized to the value of stepUpMessageMaxLength. Default value: 119
<b>ais.ondemand.stepUpLanguage</b>	
string optional	The language used to display the consent screen. Possible values: en   de   fr   it Default value: en
<b>ais.ondemand.rasAssuranceLevel</b>	
integer optional	The assurance level to request in RAS evidence lookup. Set to 3 for advanced signature or to 4 for qualified signature. Possible values: 3   4 Default value: 4
<b>ais.ondemand.rasJurisdiction</b>	
String optional	The jurisdiction to request in RAS evidence lookup. skip for any jurisdiction. Set to value "eidas" for eIDAS jurisdiction only or "zertes" for ZertES jurisdiction only. Possible values: <empty>   zertes   eidas Default value: <empty>

Example:

### Spring properties

```
ais.ondemand.keyEntity=OnDemand-Advanced
ais.ondemand.distinguishedName=cn=TEST #{'$' + '{principal.user.name}'},givenname=TEST
#{'$' + '{principal.user.claims.givenName}'},surname=TEST #{'$' + '{
principal.user.claims.familyName}'},c=de,emailaddress=#{'$' + '{
principal.user.claims.email}'},serialnumber=RAS#{'$' + '{rasEvidence.evidenceId}'}
ais.ondemand.text=Signed by #{'$' + '{principal.user.name}'}
ais.ondemand.stepUpMSISDN=#{'$' + '{principal.user.claims.msisdn}'}
ais.ondemand.stepUpMessage=Please authorize the signature as #{'$' + '{
principal.user.name}'} for: #{'$' + '{signatureEvidence.items}'}
ais.ondemand.stepUpLanguage=en
ais.ondemand.rasAssuranceLevel=4
ais.ondemand.rasJurisdiction=eidas
```

This example relies on the presence of a named principal (see chapter 9.10) with at least the following claims:

Claim	Description
givenName	The signatory's given name.
familyName	The signatory's family name.
email	The signatory's e-mail address.
msisdn	The signatory's mobile phone number used to authorize the signature.

#### 14.6.3.2 Default signer configuration "aisStatic"

This configuration defines a template for signatures based on static certificates. Applying this configuration will result in an LTV-enabled signature incl. complete validation data and a signature timestamp. It uses the most important arguments of the signature device.

## Spring XML fragment

```

<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="aisStatic" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
value="de.intarsys.security.processor.signature.DocumentSignerFactory" />
      <entry key="documentSigner.args.conformanceLevel"
value="${ais.static.conformanceLevel:LT}" />
      <entry key="documentSigner.args.signatureLabel" value="${ais.static.text:}" />
      <entry key="documentSigner.args.digestSigner.factory"
value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device" value="default@ais" />
      <entry key="documentSigner.args.digestSigner.args.keyEntity"
value="${ais.static.keyEntity:}" />
      <entry key="documentSigner.args.digestSigner.args.mode" value="static" />
    </map>
  </property>
</bean>

```

This default configuration can be customized through the use of the following custom property definitions

ais.static.conformanceLevel	
string optional	The conformance level to achieve. Supported values are: <ul style="list-style-type: none"> <li>• B (for Basic Signature)</li> <li>• T (for Signature with Time)</li> <li>• LT (for Signature with Time and Long-Term validation data)</li> </ul> Default value: LT
ais.static.keyEntity	
string required	The name of the key entity used as provided by Swisscom with your account information.
ais.static.text	
string optional	A text displayed in the signature field (in case of a PDF signature). Default value: <empty>

Example:

## Spring properties

```

ais.static.keyEntity=kpl-ais-demo
ais.static.text=EldI-V-signature with AIS

```

## 14.6.4 Timestamp Configuration

The AIS timestamp allows the creation of a document timestamp.

## Spring XML fragment

```

class="de.intarsys.cloudsuite.gears.core.service.timestamp.impl.FlowTimestampConfiguration">
  <property name="id" value="aisTimestamp"/>
  <property name="definitions">
    <map>
      <entry key="documentTimestamp.factory"
        value="de.intarsys.security.processor.timestamp.DocumentTimestampCreatorFactory"/>
      <entry key="documentTimestamp.args.timestampDevice"
        value="default@aisTimestamp"/>
    </map>
  </property>
</bean>

```

## 14.6.5 Usage

### 14.6.5.1 Signature device integration

The signature creation is performed and controlled using the generic signer implementation with two dedicated modes. These are passed to the document signer in the "digestSigner" request argument.

There are two modes available. Please use the generic **de.intarsys.security.app.signature.SignerFactory** with mode **static**

in order to create electronic seals or signatures with their certificate statically stored in Swisscom's TSP environment.

Use **de.intarsys.security.app.signature.SignerFactory** with mode **ondemand**

in order to create personal signatures relying on the just-in-time creation of short-lived certificates.

Please see [4], chapter 2.4 for details on these devices and their related arguments.

### 14.6.5.2 Static signature flow

The following sample flow shows the usual steps when requesting a signature based on a static certificate. Upon sending the request, Sign Live! cloud suite gears synchronously answers with the resulting signature.

#### 14.6.5.2.1 Send signature request and...

**service call**

```

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

{
  "args": {
    "documentSigner": {
      "args": {
        "conformanceLevel": "LT",
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@ais",
            "keyEntity": "kpl-ais-demo",
            "mode": "static"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}

```

**14.6.5.2 ...receive resulting signature****service response**

```

HTTP/1.x 200 OK
Content-Type: application/json

{
  "snapshot": {
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Result",
      "id": "149",
      "result": {
        "@class": "de.intarsys.cloudsuite.gears.core.service.signer.api.ResultSigner",
        "value": {
          "signatures": [{
            "type": "d",
            "name": "test.txt.p7s",
            "properties": {
              "signature": {
                "targetName": "test.txt"
              }
            }
          },
          "content": "MIJXxQYJKoZIh..."
        }
      }
    }
  }
}

```

**14.6.5.3 On-demand signature flow with Password-OTP authorization**

The following sample flow highlights the usual steps when requesting a signature based on an on-demand certificate. You will have to provide a MSISDN that Swisscom All-in Signing Service uses to request consent for the signature. You can include a variable with prefix "rasEvidence" in the distinguished name to have Sign Live! cloud suite gears look up the given MSISDN in RA service and fill in the corresponding value.

Upon sending the request, Sign Live! cloud suite gears requires you to present a consent page to the signatory. As this interactive step might take a few seconds, you will have to poll for the result in the following.

#### 14.6.5.3.1 Send signature request and...

##### service call

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

{
  "args": {
    "documentSigner": {
      "args": {
        "conformanceLevel": "LT",
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@ais",
            "mode": "ondemand",
            "stepUpLanguage": "en",
            "distinguishedName": "cn=TEST Max Mustermann,givenname=TEST Max,surname=TEST Mustermann,c=de,emailaddress=max.mustermann@gears.de,serialnumber=RAS${rasEvidence.evidenceId}",
            "stepUpMSISDN": "49172111111",
            "keyEntity": "OnDemand-Advanced",
            "stepUpMessage": "Please authorize the signature."
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}
```

#### 14.6.5.3.2 ...receive redirection stage info

##### service response

```
HTTP/1.x 200 OK
Content-Type: application/json

{
  "snapshot": {
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:HttpRedirect",
      "id": "98",
      "outOfBand": true,
      "url": "https://ais-sas.swisscom.com/sas/web/ConsentURL/73f8ca53-a26a-4670-8890-4eb39c8798d2"
    },
    "conversation": "dbcdca10-5998-47e8-b55b-21a01d6ff935"
  }
}
```

#### 14.6.5.3.3 Redirect user agent

Send the user agent to the consent site denoted by *url*. The technique used depends on your system architecture. As this is an out-of-band redirection response, you will most likely spawn a pop-up window presenting the page to the user.

#### 14.6.5.3.4 Acknowledge client redirection and...

**service call**

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

{
  "conversation": "dbcdca10-5998-47e8-b55b-21a01d6ff935",
  "inReplyTo": "98"
}
```

**14.6.5.3.5 ...receive processing stage info****service response**

```
HTTP/1.x 200 OK
Content-Type: application/json

{
  "snapshot": {
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Processing",
      "id": "100",
      "pollDelay": 500
    },
    "conversation": "dbcdca10-5998-47e8-b55b-21a01d6ff935"
  }
}
```

This stage indicates that the signature is currently in creation or you just have to wait for the result. You can receive state updates by acknowledging this stage. Depending on the interaction duration and your polling interval, this can take a couple of calls. So -

**14.6.5.3.6 Acknowledge...****service call**

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

{
  "conversation": "dbcdca10-5998-47e8-b55b-21a01d6ff935",
  "inReplyTo": "100"
}
```

**14.6.5.3.7 ... until you receive the resulting signature**

**service response**

```

HTTP/1.x 200 OK
Content-Type: application/json

{
  "snapshot": {
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Result",
      "id": "143",
      "result": {
        "@class": "de.intarsys.cloudsuite.gears.core.service.signer.api.ResultSigner",
        "value": {
          "signatures": [{
            "type": "d",
            "name": "test.txt.p7s",
            "properties": {
              "signature": {
                "targetName": "test.txt"
              }
            }
          }],
          "content": "MIJR8AY..."
        }
      }
    }
  }
}

```

#### 14.6.5.4 On-demand signature flow with MobileID authorization

In case the mobile phone number (MSISDN) sent to the server is activated for MobileID usage, the signature creation process is executed asynchronously as within the previous flow. However, as authorization is performed through an out-of-band channel, no redirection step is involved and your signature request will directly be answered with a processing stage, indicating the need to poll for the result.

As in the previous example you can include a variable with prefix “rasEvidence” in the distinguished name to have Sign Live! cloud suite gears look up the given MSISDN in RA service and fill in the corresponding value.

This flow is outlined in the following.

##### 14.6.5.4.1 Send signature request and...



## service call

```

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

{
  "args": {
    "documentSigner": {
      "args": {
        "conformanceLevel": "LT",
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@ais",
            "mode": "ondemand",
            "stepUpLanguage": "en",
            "distinguishedName": "cn=TEST Max Mustermann,givenname=TEST Max,surname=TEST Mustermann,c=de,emailaddress=max.mustermann@gears.de,serialnumber=RAS${rasEvidence.evidenceId}",
            "stepUpMSISDN": "491721111111",
            "keyEntity": "OnDemand-Advanced",
            "stepUpMessage": "Please authorize the signature."
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}

```

## 14.6.5.4.2 ...receive processing stage info

## service response

```

HTTP/1.x 200 OK
Content-Type: application/json

{
  "snapshot": {
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Processing",
      "id": "100",
      "pollDelay": 500
    },
    "conversation": "dbcdca10-5998-47e8-b55b-21a01d6ff935"
  }
}

```

This stage indicates that the signature is currently in creation or you just have to wait for the result. You can receive state updates by acknowledging this stage. Depending on the duration of the MobileID interaction on the signatory's mobile device and your polling interval, this can take a couple of calls. So -

## 14.6.5.4.3 Acknowledge...

---

**service call**

---

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

{
  "conversation": "dbcdca10-5998-47e8-b55b-21a01d6ff935",
  "inReplyTo": "100"
}
```

---

**14.6.5.4.4 ... until you receive the resulting signature**

---

**service response**

---

```
HTTP/1.x 200 OK
Content-Type: application/json

{
  "snapshot": {
    "stage": {
      "scheme": "urn:intarsys:names:conversation:1.0:schemes:Result",
      "id": "143",
      "result": {
        "@class": "de.intarsys.cloudsuite.gears.core.service.signer.api.ResultSigner",
        "value": {
          "signatures": [{
            "type": "d",
            "name": "test.txt.p7s",
            "properties": {
              "signature": {
                "targetName": "test.txt"
              }
            }
          }],
          "content": "MIJR8AY..."
        }
      }
    }
  }
}
```

---

### 14.6.5.5 Timestamp creation

#### Example

#### service call

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

{
  "args": {
    "documentTimestamper": {
      "factory": "
de.intarsys.security.processor.timestamp.DocumentTimestampCreatorFactory",
      "args": {
        "timestampDevice": "default@aisTimestamp",
      }
    },
  },
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<Base64-encoded PDF document>"
  }
]
```

### 14.6.6 Monitoring

This device supports the standard device properties and notifications.

By default, this device monitors the health of the AIS DSS service.

### 14.6.7 Observations

This device supports the standard observations.

## 14.7 Bundesdruckerei sign-me Device

### 14.7.1 Overview

With sign-me, D-Trust provides a trust service in accordance with the requirements of the EU's eIDAS regulations. The services offered comprise the remote creation of

- electronic signatures,
- electronic seals
- registration and management features

Sign Live! cloud suite gears currently supports the usage of signature and seal creation services along with the automatic enrollment.

To get access and make use of these services, you have to sign a partner agreement with D-Trust. Upon obtaining partner status, you will get a set of credentials which must be provided to gears by leveraging the configuration options described in the following sections. Afterwards you should immediately be able to use the signature services.

Using the sealing facilities provided by D-Trust – marketed under the product name “seal-me” – requires additional steps. Please contact your D-Trust sales representative in order to obtain a remote seal and complementary authentication credentials. As a result, you should be equipped with

- a *Qualified Seal ID token* in PKCS#12 format (i.e. a file with .p12 extension containing a soft token),
- a *PIN* (which is delivered via SMS and will be used to decrypt the soft token above) and
- an *organization name* (as given in your application form).

It is possible to register multiple seals within one partner account, which can be selectively configured and accessed within gears.

Once you have successfully registered, you can configure Sign Live! cloud suite gears for your account. See chapter 14.7.2.2 for details on how to set up access.

Depending on the requested services, different usage patterns apply. See chapter 14.7.4 for an in-depth view of the typical processes.

### 14.7.2 Authorization flow

#### 14.7.2.1 Signature authorization page flow

The sign-me device has its own pages handling user consent. There are some options and restrictions on how to integrate these pages in your application.

##### 14.7.2.1.1 Single page

The browser switches the context completely to the sign-me page.

Your application receives a `HttpRedirectStage` with `outOfBand = false` and should accordingly redirect to the new location.

After the sign-me process finished, the browser redirects to your application.

The application can now request the flow state and in most cases should immediately get a result.

This is the default behavior.

##### 14.7.2.1.2 Separate Page

The sign-me consent is handled in a separate browser page, be it a window or tab (you cannot control of a window or tab is used from JavaScript). This may be necessary because of the "no frame" constraints of the sign-me consent handling.

After the sign-me process is finished, we try the best to close the now obsolete page.

To force this behavior, you must set the device property **consentWindowName** (respective the gears property **signme.consent.windowName**) to a non-empty value. This value will be forwarded in the `HttpRedirectStage.name` property and used as the target in the JavaScript `window.open()` call.

In addition, you can select between in-band and out-of-band handling here. Setting the device property **consentOutOfBand** (respective the gears property **signme.consent.outOfBand**) to "true" will result in `HttpRedirectStage.outOfBand=true` and a polling behavior of the client.

Setting the device property **consentOutOfBand** (respective the gears property **signme.consent.outOfBand**) to "false" will result in `HttpRedirectStage.outOfBand=false`. The default client will do nothing after the consent window is opened. After the sign-me process is finished, we try out best to redirect the original window to the requested `redirectUri` before closing the sign-me dialog. As this behavior may confuse the user (when the location is changed while he is currently active), we recommend always settings **consentOutOfBand** = true and **consentWindowName** together.

For a separate page you can always configure in addition the "specs" part of the JavaScript `window.open()` function. Use the **consentWindowSpecs** device property (respective the `signme.consent.windowSpecs` gears property).

Depending on the configuration and user state, a signature process may also encompass user enrollment. This is a two-step process consisting of registration and verification, each requiring interaction via dedicated UIs. The handling of the respective UIs adheres to the consent rules above, by default being controlled through the consent configuration. However, it is possible to specifically control these processes using the device properties **registrationOutOfBand**, **registrationWindowName**, **registrationWindowSpecs** and **verificationOutOfBand**, **verificationWindowName**, **verificationWindowSpecs** respectively.

Example gears properties

## Spring properties

```
signme.registration.outOfBand=true
signme.registration.windowName=registration
signme.verification.outOfBand=true
signme.verification.windowName=verification
signme.consent.outOfBand=true
signme.consent.windowName=consent
signme.consent.windowSpecs=location=no,menubar=no,status=no,toolbar=no
```

### 14.7.2.2 Seal authorization flow

In contrast to the signature creation process, there is no interaction involved in creating a seal. Seal creation is rather authorized by means of a challenge-response protocol, which is satisfied using configured authorization tokens as described in the following sections.

## 14.7.3 Device Configuration

### 14.7.3.1 Device provider

A device is created by the respective device provider. The sign-me device provider is registered with the Spring bean id **deviceProvider.signMe**.

The id of the device provider is **signMe**.

A concrete bridge device can be addressed as **<device id>@signMe**.

There is no configuration you can apply to the device provider itself.

### 14.7.3.2 Device properties

These properties are available for a sign-me device.

id	
String required	The id of the new device. Only ASCII-alphanumeric, '_' and '-' is allowed.
service.address	
URL required	The endpoint for the sign-me signing service.  There are two well-known endpoints: <b>Reference environment:</b> https://cloud-ref-sp.sign-me.de:443/api/v2 <b>Production environment:</b> https://cloud-sp.sign-me.de:443/api/v2
service.username	
String required	The user for service authentication
service.password	
String required	The password for service authentication
partner.username	
String required	The partner user for service accounting
partner.password	
String required	The partner password for service accounting
partner.role	
String required	The role that this partner account uses for the installation. Currently only "PARTNER" is supported.
authorization.tokenPath	
String	Default: \${cloudsuite.config.shared}/devices/signMe/<device-id>/authorization-tokens The path to the directory where the <i>Qualified Seal ID token</i> files for two-factor authorization of sealing should reside. The default path is dynamic with respect to the device which is used for sealing.
consentOutOfBand	
Boolean	Default: false Setting this property to "true" results in sending <code>HttpRedirectStage.outOfBand=true</code> , which forces the creation of a new browser page for the consent and a polling behavior in your application (if you use the standard <code>csconversation.js</code> ). You should set this to "true" whenever you explicitly set a window name.
consentWindowName	
String	The name of the window to create for the sign-me consent. This results in sending <code>HttpRedirectStage.name=&lt;my name&gt;</code> . The client should create a new window for the redirected content.
consentWindowSpecs	
String	The "specs" to be used in the <code>window.open()</code> call.
registrationOutOfBand	
Boolean	Default: \${consentOutOfBand} Setting this property to "true" results in sending <code>HttpRedirectStage.outOfBand=true</code> , which forces the creation of a new browser page for the registration and a polling behavior in your application (if you use the standard <code>csconversation.js</code> ). You should set this to "true" whenever you explicitly set a window name.
registrationWindowName	
String	Default: \${consentWindowName} The name of the window to create for the sign-me registration. This results in sending <code>HttpRedirectStage.name=&lt;my name&gt;</code> .

	The client should create a new window for the redirected content.
<b>registrationWindowSpecs</b>	
String	Default: \${consentWindowSpecs} The "specs" to be used in the window.open() call.
<b>verificationOutOfBand</b>	
Boolean	Default: \${consentOutOfBand} Setting this property to "true" results in sending HttpRedirectStage.outOfBand=true, which forces the creation of a new browser page for the verification and a polling behavior in your application (if you use the standard csconversation.js). You should set this to "true" whenever you explicitly set a window name.
<b>verificationWindowName</b>	
String	Default: \${consentWindowName} The name of the window to create for the sign-me verification. This results in sending HttpRedirectStage.name=<my name>. The client should create a new window for the redirected content.
<b>verificationWindowSpecs</b>	
String	Default: \${consentWindowSpecs} The "specs" to be used in the window.open() call.
<b>enrollment.enabled</b>	
Boolean	Flag if gears should trigger the automatic enrollment when the signing entity is unknown to the sign-me backend. Default true
<b>enrollment.checkConfirmationEmail</b>	
Boolean	Flag if the enrollment process requires the confirmation of the email address before a signature can take place. Default false
<b>enrollment.identityVerificationType</b>	
String	The type of verification process to apply in the enrollment process. This is one of <ul style="list-style-type: none"> <li>• VIDEO_4EYE</li> <li>• WEB_FT_IDENT_4EYE</li> <li>• POS_FT_IDENT_4EYE</li> <li>• E_ID</li> <li>• FT_E_ID (Default)</li> </ul> If you are using automatic enrollment, you must discuss with your D-Trust representative which verification type must be used for your account.
<b>queryStringSignature</b>	
String	An optional query suffix for the signature process URL. This allows to fine tune the sign-me UI behavior.  The flags are undocumented and unsupported. Remember to escape "&" when used in XML.  "LO=M" Layout support for mobile "USERNAME=x" Preset the username with "x"  Example "LO=M&USERNAME=?{username}"
<b>queryStringRegistration</b>	
String	An optional query suffix for the registration process URL. This allows to fine tune the sign-me UI behavior.  The flags are undocumented and unsupported. Remember to escape "&" when used in XML.

	"LO=M" Layout support for mobile "EMRO=true" Switch the email field to read only  Example "LO=M&EMRO=true"
queryStringVerification	
String	An optional query suffix for the verification process URL. This allows to fine tune the sign-me UI behavior.  The flags are undocumented and unsupported. Remember to escape "&" when used in XML.  "LO=M" Layout support for mobile "USERNAME=x" Preset the username with "x"  Example "LO=M&USERNAME=?{username}"

#### 14.7.3.3 Default device

The system comes with a preconfigured sign-me device "default", which can be addressed as **default@signMe**.

This is a standard suitable for most use cases.

The default device properties are configured using a set of Spring properties that must be provided in your gears.properties file. Most of these information stems from the credentials you receive after getting partner status.

##### Property definitions:

signme.service.address	
URL required	Define "service.address"
signme.service.username	
String required	Define "service.username"
signme.service.password	
String required	Define "service.password"
signme.partner.username	
String required	Define "partner.username"
signme.partner.password	
String required	Define "partner.password"
signme.partner.role	
String required	Define "partner.role"
signme.authorization.tokenPath	
String	Define "authorization.tokenPath"  Default: \${cloudsuite.config.shared}/devices/signMe/default /authorization-tokens
signme.consent.outOfBand	
Boolean	Define "consentOutOfBand"



	Default: false
signme.consent.windowName	
String	Define "consentWindowName"
	Default: ""
signme.consent.windowSpecs	
String	Define "consentWindowSpecs"
	Default: ""
signme.registration.outOfBand	
Boolean	Define "registrationOutOfBand"
	Default: \${signme.consent.outOfBand:false}
signme.registration.windowName	
String	Define "registrationWindowName"
	Default: \${signme.consent.windowName:}
signme.registration.windowSpecs	
String	Define "registrationWindowSpecs"
	Default: \${signme.consent.windowSpecs:}
signme.verification.outOfBand	
Boolean	Define "verificationOutOfBand"
	Default: \${signme.consent.outOfBand:false}
signme.verification.windowName	
String	Define "verificationWindowName"
	Default: \${signme.consent.windowName:}
signme.verification.windowSpecs	
String	Define "verificationWindowSpecs"
	Default: \${signme.consent.windowSpecs:}
signme.enrollment.enabled	
Boolean	Define "enrollment.enabled"
	Default: false
signme.enrollment.checkConfirmationEmail	
Boolean	Define " enrollment.checkConfirmationEmail"
	Default: false
signme.enrollment.identityVerificationType	
String	Define " enrollment.identityVerificationType"
	Default: "FT_E_ID"
signme.queryStringSignature	
String	Define "queryStringSignature"
	Default: "LO=M&USERNAME=?{username}"
signme.queryStringRegistration	
String	Define "queryStringRegistration"
	Default: "LO=M&EMRO=true"
signme.queryStringVerification	
String	Define "queryStringVerification"
	Default: "LO=M&USERNAME=?{username}"

A properties file could look like the following:

---

### 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=k84jdg74hwbs9tjn4g3zsdjdjnkdocubzvfdzvbzd
signme.partner.role=PARTNER
```

---

#### 14.7.3.4 Custom device

A custom device can be created using the device provider (e.g. **deviceProvider.signMe**) as a factory.

The factory method is "createInstance", all support properties are enumerated in a map argument.

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

---

### Spring XML fragment

---

```
<bean factory-bean="deviceProvider.signMe" factory-method="createInstance">
  <constructor-arg>
    <map>
      <entry key="key" value="value" />
    </map>
  </constructor-arg>
</bean>
```

---

##### 14.7.3.4.1 Example

Create a device "mydevice@signMe" with another partner account.

## Spring XML fragment

```

<bean factory-bean="deviceProvider.signMe" factory-method="createInstance">
  <constructor-arg>
    <map>
      <entry>
        key="id"
        value="mydevice" />
      <entry>
        key="service.address"
        value="https://cloud-ref-sp.sign-me.de:443/api/v2" />
      <entry>
        key="service.username"
        value="serviceuser" />
      <entry>
        key="service.password"
        value="d83jd9fnnc3fj3i4kd" />
      <entry>
        key="partner.username"
        value="2ndUser" />
      <entry>
        key="partner.password"
        value="53udicf4c3zsu" />
      <entry>
        key="partner.role"
        value="PARTNER" />
    </map>
  </constructor-arg>
</bean>

```

## 14.7.4 Usage

### Factory

For signature and seal creation, use the **de.intarsys.security.device.signme.processor.SignMeDigestSignerFactory**.

The arguments we could use for personal signature creation are

### Arguments

device	
String Optional	The id of the device that will create the signer application. If no device is provided, the first configured sign-me device is used.
username	
String Required	The id of the principal that owns the key that will be used for signing.
principalFilter	
String Optional	An expression that will select the correct token from the set of tokens assigned to the principal. Depending on the quality of the required signature you should select one of <ul style="list-style-type: none"> <li>principal.hasCapability('SIGNATURE_BASIC')</li> <li>principal.hasCapability('SIGNATURE_ADVANCED')</li> <li>principal.hasCapability('SIGNATURE_QUALIFIED')</li> </ul>

## 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.device.signme.processor.SignMeDigestSignerFactory",
          "args": {
            "device": "default@signMe",
            "username": "foobar",
            "principalFilter": "principal.hasCapability('SIGNATURE_ADVANCED')"
          }
        }
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.txt",
      "content": "QUJDLi4u"
    }
  ]
}

```

The arguments we could use for seal creation are

## Arguments

device	
String Optional	The id of the device that will create the signer application. If no device is provided, the first configured sign-me device is used.
authorization.tokenId	
String Required	The name of the <i>Qualified Seal ID token</i> file used for sealing authorization.
authorization.tokenPassword	
String Required	The <i>PIN</i> of the token file used for sealing authorization.
authorization.organizationName	
String Required	The name of the <i>sealing organization</i> .
principalFilter	
String Required	An expression that will select the correct token from the set of tokens assigned to the principal. For sealing you should use <ul style="list-style-type: none"> <li>principal.hasCapability('SEALING_QUALIFIED')</li> </ul>

## 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.device.signme.processor.SignMeDigestSignerFactory",
          "args": {
            "device": "default@signMe",
            "principalFilter": "principal.hasCapability('SEALING_QUALIFIED')",
            "authorization": {
              "tokenId": "TestOrg_CSM9832.p12",
              "tokenPassword": "U4KMTL0AJ1",
              "orgnanizationName": "TestOrg",
            }
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "testfile.pdf",
    "content": "QUJDLi4us"
  }
]
}
```

### 14.7.5 Signer Configuration

To further abstract from a concrete device and concrete arguments, remember you can invent a "signer configuration" (see 9.11.1.1). Please be sure to use a unique id for your configuration.

The gears environment comes with **predefined** signer configurations for the standard tokens that can be assigned to a principal, being

- **BASIC**
- **ADVANCED**
- **QUALIFIED**

for personal signing and

- **QUALIFIED\_SEAL**

for seal creation.

The respective configurations are:

- **signMeBasic**
- **signMeAdvanced**
- **signMeQualified**
- **signMeQualifiedSeal**

#### 14.7.5.1 Personal signature creation

Personal signing can be easily achieved by applying one of the signer configurations

- **signMeBasic**
- **signMeAdvanced**
- **signMeQualified**

The following is the example for the "BASIC" configuration, the "ADVANCED" one is alike:

## Spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="signMeBasic"/>
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
value="de.intarsys.security.processor.signature.DocumentSignerFactory"/>
      <entry key="documentSigner.args.digestSigner.factory"
value="de.intarsys.security.device.signme.processor.SignMeDigestSignerFactory"/>
      <entry key="documentSigner.args.digestSigner.args.device" value="default@signMe"/>
      <entry key="documentSigner.args.digestSigner.args.username"
value="\${signme.signer.username:}" />
      <entry key="documentSigner.args.digestSigner.args.principalFilter"
value="principal.hasCapability('SIGNATURE_BASIC')"/>
      <entry key="documentSigner.args.signatureLabel" value="\${signme.signer.text:}" />
    </map>
  </property>
</bean>
```

Instead of the "usage" example you can leverage this configuration:

## service call

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

{
  "configuration": "signMeBasic",
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}
```

Below, the "QUALIFIED" configuration by default creates an LT-level signature, including a timestamp and validation data. This is strongly recommended, as this signature call to the sign-me service may result in a signature applying a short-lived certificate:

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="signMeQualified" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
value="de.intarsys.security.processor.signature.DocumentSignerFactory" />
      <entry key="documentSigner.args.conformanceLevel" value="LT" />
      <entry key="documentSigner.args.digestSigner.factory"
value="de.intarsys.security.device.signme.processor.SignMeDigestSignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device" value="default@signMe"
/>
      <entry key="documentSigner.args.digestSigner.args.username"
value="\${signme.signer.username:}" />
      <entry key="documentSigner.args.digestSigner.args.principalFilter"
value="principal.hasCapability('SIGNATURE_QUALIFIED')"/>
      <entry key="documentSigner.args.signatureLabel" value="\${signme.signer.text:}" />
    </map>
  </property>
</bean>
```

These default configurations can be customized through the use of the following custom property definitions:

signme.signer.username	
String	A default value for <b>documentSigner.args.digestSigner.args.username</b> .

	<p>Instead of always sending this value directly in the arguments to your signer/create call, you can define a static or dynamic value here.</p> <p>A good value is</p> <p><b><code>{principal.user.claims.signMeUsername:!principal.user.name}</code></b></p> <p>which uses the current principal user.</p>
<b>signme.signer.text</b>	
<b>String</b>	<p>A default value for <b>documentSigner.args.signatureLabel</b>.</p> <p>Instead of always sending this value directly in the arguments to your signer/create call, you can define a static or dynamic value here.</p>

A properties file could look like the following:

### Spring properties

```
signme.signer.username=?{principal.user.name}
signme.signer.text=Signed by sign-me?{entity.nl}?{digestSigner.subject.cn}
```

#### 14.7.5.2 Seal creation

Sealing can be easily achieved by applying the signer configuration

- `signMeQualifiedSeal`

For the sealing process, remote sealing is performed by means of electronic seal hosted at the TSP. Application of the electronic seal is authorized by the partner organization by means of two-factor authorization, which involves challenge signing using soft token, an encrypted file in PKCS #12 format. To access the token file we need the tokenId, which is the name of the file, a tokenPassword which is a PIN delivered by SMS and the organizationName which was specified in the application for seal.

The "QUALIFIED\_SEAL" configuration creates an LT-level signature by default, including a timestamp and validation data. In contrast to the signer configurations for personal signing, it replaces the user name by the seal authorization data:

```
<bean
  class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="signMeQualifiedSeal" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
        value="de.intarsys.security.processor.signature.DocumentSignerFactory" />
      <entry key="documentSigner.args.conformanceLevel" value="LT" />
      <entry key="documentSigner.args.digestSigner.factory"
        value="de.intarsys.security.device.signme.processor.SignMeDigestSignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device" value="default@signMe" />
      <entry key="documentSigner.args.digestSigner.args.principalFilter"
        value="principal.hasCapability('SEALING_QUALIFIED')" />
      <entry key="documentSigner.args.digestSigner.args.authorization.tokenId"
        value="{signme.sealer.authorization.tokenId}" />
      <entry key="documentSigner.args.digestSigner.args.authorization.tokenPassword"
        value="{signme.sealer.authorization.tokenPassword}" />
      <entry key="documentSigner.args.digestSigner.args.authorization.organizationName"
        value="{signme.sealer.authorization.organizationName}" />
      <entry key="documentSigner.args.signatureLabel" value="{signme.sealer.text}" />
    </map>
  </property>
</bean>
```

This default configuration can be customized through the use of the following custom property definitions:

signme.sealer.authorization.tokenId	
String	<p>A default value for <b>documentSigner.args.digestSigner.args.authorization.tokenId</b>.</p> <p>Instead of always sending this value directly in the arguments to your signer/create call, you can define a static or dynamic value here.</p> <p>A good value is</p> <p><b>?{principal.client.claims.tokenId:!""}</b></p>
signme.sealer.authorization.tokenPassword	
String	<p>A default value for <b>documentSigner.args.digestSigner.args.authorization.tokenPassword</b>.</p> <p>Instead of always sending this value directly in the arguments to your signer/create call, you can define a static or dynamic value here.</p>
signme.sealer.authorization.organizationName	
String	<p>A default value for <b>documentSigner.args.digestSigner.args.authorization.organizationName</b>.</p> <p>Instead of always sending this value directly in the arguments to your signer/create call, you can define a static or dynamic value here.</p> <p>A good value</p> <p><b>?{principal.client.claims.organizationName:!""}</b></p>
signme.sealer.text	
String	<p>A default value for <b>documentSigner.args.signatureLabel</b>.</p> <p>Instead of always sending this value directly in the arguments to your signer/create call, you can define a static or dynamic value here.</p>

A properties file could look like the following:

### Spring properties

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

## 14.7.6 Monitoring

This device supports the standard device properties and notifications.

Health check is done by default by requesting “getVersion” from the API in a concurrent health monitor. You can overwrite the properties like

Example:

```
<entry key="healthMonitor">
  <map>
    <entry key="delay" value="42" />
  </map>
</entry>
```



### 14.7.7 Observations

To support client specific accounting, this device emits observations that allow to observe the process created on the TSP backend:

source	device.<provider>.<device>
	The identification of the device that originates the observation.  Example "device.signme.default"
code	sign.signme.verificationProcess.created
	This event is issued after a signme verification process has been created
verificationProcessId	
	The signme id of the verification process
*	
	All other properties available when observing a security application

source	device.<provider>.<device>
	The identification of the device that originates the observation.  Example "device.signme.default"
code	sign.signme.verificationProcess.fail
	Issued after a "failure" callback from the signme verification process.  As this is an error condition, you <b>cannot</b> be sure if the corresponding process was billed at the TSP.
verificationProcessId	
	The signme id of the verification process
*	
	All other properties available when observing a security application

source	device.<provider>.<device>
	The identification of the device that originates the observation.  Example "device.signme.default"
code	sign.signme.verificationProcess.cancel
	Issued after a callback from the signme verification process that was caused by a cancellation.
verificationProcessId	
	The signme id of the verification process.
*	
	All other properties available when observing a security application

source	device.<provider>.<device>
	The identification of the device that originates the observation.  Example "device.signme.default"
code	sign.signme.verificationProcess.ok
	Issued after an "ok" callback from the signme verification process.
verificationProcessId	
	The signme id of the verification process created.

	At least at this moment you can be sure that the corresponding process was billed at the TSP.
*	
	All other properties available when observing a security application

source	device.<provider>.<device>
	The identification of the device that originates the observation.  Example "device.signme.default"
code	sign.signme.signatureProcess.created
	This event is issued immediately after a signme signature process has been created
signatureProcessId	
	The signme id of the signature process
*	
	All other properties available when observing a security application

source	device.<provider>.<device>
	The identification of the device that originates the observation.  Example "device.signme.default"
code	sign.signme.signatureProcess.fail
	Issued after a "failure" callback from the signme backend.  As this is an error condition, you <b>cannot</b> be sure if the corresponding process was billed at the TSP.
signatureProcessId	
	The signme id of the signature process
*	
	All other properties available when observing a security application

source	device.<provider>.<device>
	The identification of the device that originates the observation.  Example "device.signme.default"
code	sign.signme.signatureProcess.cancel
	Issued after a callback from the signme backend that was caused by a cancellation.
signatureProcessId	
	The signme id of the signature process.
*	
	All other properties available when observing a security application

Source	device.<provider>.<device>
	The identification of the device that originates the observation.  Example "device.signme.default"
Code	sign.signme.signatureProcess.ok
	Issued after an "ok" callback from the signme backend.
signatureProcessId	

	<p>The signme id of the signature process created.</p> <p>At least at this moment you can be sure that the corresponding process was billed at the TSP.</p>
*	
	All other properties available when observing a security application

## 14.8 TSA Device

### 14.8.1 Overview

The TSA device is a built-in generator for RFC3161-style timestamps.

The device applies a hash function to some data-to-be-timestamped, documents the system time in a timestamp info data structure and signs all this using a preconfigured signature device.

### 14.8.2 Device Configuration

#### 14.8.2.1 Device provider

A device is created by the respective device provider. The demo device provider is registered with the Spring bean id **deviceProvider.tsa**.

The id of the device provider is **tsa**.

A concrete TSA device can be addressed as **<device id>@tsa**.

There is no configuration you can apply to the device provider itself.

#### 14.8.2.2 Device properties

id	
string required	The id of the new device. Only ASCII-alphanumeric, '_' and '-' is allowed.
digestSigner	
Instance spec	The definition of a signer application used to sign the generated timestamp info. The signer is typically denoted by a factory reference and corresponding arguments. See section 14.1.4 for details on how to specify a signer application.
digester	
String optional	An optional reference to the hash algorithm to be applied to the generated timestamp info. The following algorithms are supported. <ul style="list-style-type: none"> <li>• SHA-1</li> <li>• SHA-256</li> <li>• SHA-384</li> <li>• SHA-512</li> <li>• RIPEMD160</li> <li>• RIPEMD256</li> </ul> The default is "SHA-256".

#### 14.8.2.3 Default device

The system comes with a preconfigured device "default", which can be addressed as **default@tsa**.

#### 14.8.2.4 Custom device

A custom device can be created using the device provider (e.g. **deviceProvider.tsa**) as a factory.

The factory method is "createInstance", all supported bean definitions are enumerated in a map argument.

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

## Spring XML fragment

```
<bean factory-bean="deviceProvider.tsa" factory-method="createInstance">
  <constructor-arg>
    <map>
      <entry key="id" value="mytsa" />
      <entry key="digester" value="SHA-256" />
      <entry key="digestSigner.factory"
value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="digestSigner.args.device" value="tsa@demo" />
    </map>
  </constructor-arg>
</bean>
```

## 14.8.2.4.1 Example

Create a device "mytsa@tsa".

## Spring XML fragment

```
<bean factory-bean="deviceProvider.tsa" factory-method="createInstance">
  <constructor-arg>
    <map>
      <entry key="id" value="mytsa" />
      <entry key="digester" value="SHA-256" />
      <entry key="digestSigner.factory"
value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="digestSigner.args.device" value="tsa@demo" />
    </map>
  </constructor-arg>
</bean>
```

## 14.8.3 Signer Usage

## Context

You can pass a tsa device as an argument to most document signer implementations.

## Arguments

timestampDevice	
string optional	The id of the timestamp device that will generate a signature timestamp to be embedded. For the preconfigured tsa demo device, you should refer to "tsa@demo".

## 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"
          }
        },
        "timestampDevice": "default@tsa"
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.txt",
      "content": "QUJDLi4u"
    }]
  }
}
```

#### 14.8.4 Signer Configuration integration

The timestamp device can also be statically integrated in the context of a signer configuration using the document signer argument **timestampDevice**. See section 14.3.5 for a corresponding example.

#### 14.8.5 Timestamper Usage

## Example

## service call

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

{
  "args": {
    "documentTimestamper": {
      "factory": "de.intarsys.security.processor.timestamp.DocumentTimestampCreatorFactory",
      "args": {
        "timestampDevice": "default@tsa",
      }
    },
    "documents": [{
      "type": "d",
      "name": "test.pdf",
      "content": "<Base64-encoded PDF document>"
    }]
  }
}
```

#### 14.8.6 Timestamper Configuration

The timestamper allows the creation of a document timestamp.

---

**Spring XML fragment**

---

```
class="de.intarsys.cloudsuite.gears.core.service.timestamp.impl.FlowTimestampConfiguration">
  <property name="id" value="aisTimestamp"/>
  <property name="definitions">
    <map>
      <entry key="documentTimestamp.factory"
        value="de.intarsys.security.processor.timestamp.DocumentTimestampCreatorFactory"/>
      <entry key="documentTimestamp.args.timestampDevice"
        value="default@tsa"/>
    </map>
  </property>
</bean>
```

### 14.8.7 Monitoring

This device supports the standard device properties and notifications.

It does currently not support background health check

### 14.8.8 Observations

This device supports the standard observations.

## 14.9 UPReg Device

### 14.9.1 Overview

The UPReg device allows you – in accordance with the “Verordnung über die Erstellung elektronischer öffentlicher Urkunden und elektronischer Beglaubigungen” (EÖBV) to obtain confirmations of admission (Zulassungsbestätigungen) from the Swiss Register der Urkundspersonen (UPReg) and apply them to documents with a qualified signature.

To use this device

- you (the operator) must register as a provider and
- any user requesting attestation must be registered as a signatory

with the UPReg.

### 14.9.2 Device Configuration

#### 14.9.2.1 Device provider

The UPReg device provider has the ID **upreg** and is registered as a Spring bean with the ID **deviceProvider.upreg**. A concrete device can be addressed as **<device id>@upreg**. The device provider cannot be configured.

#### 14.9.2.2 Device properties

An UPReg device can be configured with these properties:

id	
String required	The id of the new device. Only ASCII-alphanumeric, '_' and '-' is allowed.
serviceUrl	
URL optional	Default: https://www.upreg.ch  The root URL of the UPReg service to be used.
providerId	
String optional	Default: none, for which UPReg assumes a Cygillum client.  The ID of the calling system, for which an authentication callback URL has been registered with UPReg.  The registered callback URL must match external URL of the device's authentication callback service (for example, https://somehost.example.com/cloudsuite-gears/core/ <b>api/upreg/authenticationCompleted</b> , where the suffix in bold is given, but the prefix is variable and depends on your gears setup).
callbackPort	
Integer optional	Default: -1  HTTP port for the authentication callback. If no port is specified or the specified port 0 or negative, then UPReg assumes port 443.
httpConfiguration.sslContext	
SSLContext required	The SSL context to use for the mutual-TLS-authenticated communication with the UPReg service.
httpConfiguration.hostnameVerifier	
HostnameVerifier optional	The implementation of <i>javax.net.ssl.HostnameVerifier</i> to use for authentication of the TLS communication partner.
httpConfiguration.connectTimeout	
Integer optional	Default: 0



	The connection timeout in milliseconds for outgoing connections. 0 indicates infinite.
<b>httpConfiguration.readTimeout</b>	
Integer optional	Default: 0  The connection timeout in milliseconds for outgoing connections. 0 indicates infinite.
<b>httpConfiguration.useProxy</b>	
Boolean optional	Default: false  If true, a HTTP proxy specified by the standard system properties (http.proxyHost, http.proxyPort, http.proxyUser, and http.proxyPassword) is used to connect to the UPRReg service.

#### 14.9.2.3 Default device

The system comes with a preconfigured UPRReg device, which can be addressed as **default@upreg**. This is a standard suitable for most use cases.

The default device properties are configured using the following set of Spring properties that must be provided in your gears.properties file:

<b>upreg.serviceUrl</b>	
URL optional	Define serviceUrl
<b>upreg.callbackPort</b>	
Integer optional	Define callbackPort
<b>upreg.providerId</b>	
String optional	Define providerId

A properties file could look like the following:

Spring properties	
<pre>upreg.serviceUrl=https://www.upreg.ch upreg.callbackPort=9000 upreg.providerId=example-provider</pre>	

### 14.9.3 Usage

#### Factory

To apply confirmations of admission with the UPRReg service, you have to use a UPRReg document signer together with a UPRReg digest signer. Both signers can only be used in tandem and only on PDF/A documents that have been signed and conform to the EÖBV.

You create a UPRReg signer with the generic **de.intarsys.security.app.signature.SignerFactory** and configure it with the following arguments:

#### Arguments

<b>device</b>	
String required	The UPRReg device that will create the signer application. For the preconfigured UPRReg device, refer to "default@upreg".

canton	
String required	The abbreviation of the canton, in which the notary is registered (for example, "BE" for Bern).
domain	
String required	The abbreviation of the domain, in which the notary is registered (for example, "ehra" for Handelsregisterbehörden).

In addition, you have to explicitly specify the UPReg document signer factory or else gears will use the default document signer for PDF documents.

### 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.upreg.UPRegDocumentSigner",
      "args": {
        "digestSigner": {
          "factory": "de.intarsys.security.app.signature.SignerFactory",
          "args": {
            "device": "default@upreg",
            "canton": "BE",
            "domain": "ehra"
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<Base64-encoded PDF document>"
  }]
}
```

## 14.9.4 Signer Configuration

To further abstract from a concrete device and concrete arguments, remember you can define a "signer configuration" (see 9.11.1.1). Please be sure to use a unique id for your configuration.

The gears environment comes with a predefined signer configuration named **upreg**, which uses the **default@upreg** device and looks like this:

## Spring XML fragment

```

<bean
  class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="upreg"/>
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
        value="de.intarsys.security.document.upreg.UPRegDocumentSignerFactory"/>
      <entry key="documentSigner.args.digester"
        value="\${upreg.hashAlgorithm:SHA-256}"/>

      <entry key="documentSigner.args.digestSigner.factory"
        value="de.intarsys.security.app.signature.SignerFactory"/>
      <entry key="documentSigner.args.digestSigner.args.device"
        value="default@upreg"/>
      <entry key="documentSigner.args.digestSigner.args.canton"
        value="\${upreg.canton:}"/>
      <entry key="documentSigner.args.digestSigner.args.domain"
        value="\${upreg.domain:}"/>
    </map>
  </property>
</bean>

```

This predefined configuration can be customized through the use of the following custom property definitions

upreg.hashAlgorithm	
String optional	Default: SHA-256  Hash algorithm for the document hash to be signed. UPReg supports the following SHA-256, SHA-384, SHA-512, SHA3-256, SHA3-384, and SHA3-512.
upreg.canton	
String required	The abbreviation of the canton in which the notary is registered (for example, "BE" for Bern).
upreg.domain	
String required	The abbreviation of the domain, in which the notary is registered (for example, "ehra" for Handelsregisterbehörden).

For example:

## Spring properties

```

upreg.canton=BE
upreg.domain=ehra

```

Instead of explicitly defining all details of the UPReg document signer as in the example in 14.9.3, you can leverage the predefined configuration:

---

**service call**

---

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

```
{
  "configuration": "upreg",
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<Base64-encoded PDF document>"
  }]
}
```

### 14.9.5 Monitoring

This device supports the standard device properties and notifications.

It does currently not support background health check

### 14.9.6 Observations

This device supports the standard observations.

## 14.10 Azure Device

### 14.10.1 Overview

The Azure device integrates certificates into your signing processes which are stored and managed in Microsoft® Azure Key Vault.

Using these certificates requires that

- you have an Azure Key Vault instance up and running,
- Azure permissions set up in a way that the gears application can access the key vault and
- you have generated or imported at least one certificate in Azure Key Vault which is meant to be used in your signing processes.

The device supports the use of

- RSA keys and certificates in conjunction with PSS or PKCS#1v1.5 padding and hash algorithms SHA-256, SHA-384 or SHA-512 and
- EC keys based on the curves
  - P-256 in conjunction with hash algorithm SHA-256,
  - P-384 in conjunction with hash algorithm SHA-384 or
  - P-521 in conjunction with hash algorithm SHA-512.

Once these requirements are met, you can go on and configure the gears Azure device and its use as described in the following sections.

### 14.10.2 Device Configuration

#### 14.10.2.1 Device provider

The Azure device provider has the ID **azure** and is registered as a Spring bean with the ID **deviceProvider.azure**. A concrete device can be addressed as **<device id>@azure**. The device provider cannot be configured.

#### 14.10.2.2 Device properties

An Azure device can be configured with these properties:

id	
String required	The id of the new device. Only ASCII-alphanumeric, '_' and '-' is allowed.
keyVaultUrl	
URL required	The URL of the key vault to access, for example https://my-key-vault.vault.azure.net/ .
credential	
Instance spec optional	Default: null, for which the a DefaultAzureCredential is used.  Instance spec defining how to obtain a credential for accessing the key vault. See section 14.10.2.3 for information on credential definition.

#### 14.10.2.3 Credential definition

In order to access the Azure Key Vault, gears has to authenticate with the service and be granted access. There are two means of gaining this access grant:

- Authentication using the DefaultAzureCredential
- Authentication using an app registration

The following sections provide information on how to apply these authentication mechanisms.

### 14.10.2.3.1 Authentication using the DefaultAzureCredential

Authenticates using environment variables or the shared token cache. It tries to create a valid credential in the following order:

- EnvironmentCredential
- ManagedIdentityCredential
- SharedTokenCacheCredential
- IntelliJCredential
- VisualStudioCodeCredential
- AzureCliCredential
- AzurePowerShellCredential
- Fails if none of the credentials above could be created.

See <https://learn.microsoft.com/en-us/java/api/com.azure.identity.defaultazurecredential> for further details on the underlying authentication methods.

This is the default authentication mechanism if no credential is defined for the device.

### 14.10.2.3.2 Authentication using an app registration

You can register your application (i.e. gears) in Azure Active Directory and assign a client secret. After setting up usage permissions within your key vault for this application, you can access the key vault from gears using the following data:

- your Azure tenant ID
- your registered application's client ID
- your registered application's client secret

#### 14.10.2.3.2.1 Credential factory

The credential factory has the ID

**de.intarsys.security.device.azure.credential.ClientSecretCredentialFactory.**

This is the default credential factory if an instance spec without factory ID is defined for the device.

#### 14.10.2.3.2.2 Credential arguments

The factory creates the credential based on the following arguments:

tenantId	
String required	The Azure tenant ID.
clientId	
String required	The client ID obtained during app registration.
clientSecret	
String required	The client secret assigned to the registered app.

### 14.10.2.4 Default device

The system comes with a preconfigured Azure device, which can be addressed as **default@azure**. This is a standard suitable for most use cases. It assumes that you have a valid app registration and authenticate using client ID and secret as described in section 14.10.2.3.2.

The default device properties are configured using the following set of Spring properties that must be provided in your gears.properties file:

azure.keyVaultUrl	
URL	Define keyVaultUrl.

optional	
azure.credential.tenantId	
String optional	Define credential.args.tenantId.
azure.credential.clientId	
String optional	Define credential.args.clientId.
azure.credential.clientSecret	
String optional	Define credential.args.clientSecret.

A properties file could look like the following:

### Spring properties

```
azure.keyVaultUrl=https://my-keys.vault.azure.net/
azure.credential.tenantId=2c5f744a-f1f1-4023-9efa-9f555250b46f
azure.credential.clientId=0c0330c8-d173-422e-8a39-21203d9476a2
azure.credential.clientSecret=${secret.plain#amRrZGozOTcwb2pkc2YwMzg0MDkzMnJqZndwb2ktw6Qrd2pvd3JqZnc=}
```

## 14.10.3 Usage

### Factory

For signature creation, use the generic **de.intarsys.security.app.signature.SignerFactory**.

### Arguments

device	
String required	The id of the device that will create the signer application. For the preconfigured Azure device, you should refer to "default@azure".
principalFilter	
String required	An expression that will select the correct certificate from the set of certificates available in the key vault. Assuming that you know the certificate name as registered in the key vault, you should use an expression of following pattern: principal.certificateName == 'my-certificate-name'
rsaSignatureScheme	
String optional	Default: PSS  The RSA signature scheme to use in case of RSA signatures. Possible values are: <ul style="list-style-type: none"> <li>PSS</li> <li>PKCSv1_5</li> </ul>

## 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@azure",
            "principalFilter": "principal.certificateName == 'my-certificate-name'",
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}
```

## 14.10.4 Signer Configuration

To further abstract from a concrete device and concrete arguments, remember you can define a "signer configuration" (see 9.11.1.1). Please be sure to use a unique id for your configuration.

The gears environment comes with a predefined signer configuration named **azure**, which uses the **default@azure** device and looks like this:

## Spring XML fragment

```
<bean
  class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="azure" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
        value="de.intarsys.security.processor.signature.DocumentSignerFactory" />
      <entry key="documentSigner.args.digestSigner.factory"
        value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device"
        value="default@azure" />
      <entry key="documentSigner.args.digestSigner.args.principalFilter"
        value="principal.certificateName == '${azure.certificateName}'" />
      <entry key="documentSigner.args.digestSigner.args.rsaSignatureScheme"
        value="${azure.rsaSignatureScheme:PSS}" />
    </map>
  </property>
</bean>
```

This predefined configuration can be customized through the use of the following custom property definitions

azure.certificateName	
String required	The name of the certificate to use, as registered in the Azure Key Vault.
azure.rsaSignatureScheme	
String optional	Default: PSS



	<p>The RSA signature scheme to use in case of RSA signatures. Possible values are:</p> <ul style="list-style-type: none"> <li>• PSS</li> <li>• PKCSv1_5</li> </ul>
--	--

For example:

### Spring properties

```
azure.certificateName=demo
```

Instead of explicitly defining all details of the Azure signer as in the example in 14.10.3, you can leverage the predefined configuration:

### service call

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

{
  "configuration": "azure",
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<Base64-encoded PDF document>"
  }]
}
```

## 14.10.5 Monitoring

This device supports the standard device properties and notifications.

It does currently not support background health check

## 14.10.6 Observations

This device supports the standard observations.

## 14.11 proNEXT Device

### 14.11.1 Overview

The proNEXT device allows remote signing using procilon's proNEXT Secure Framework components, typically comprising

- an identity provider (IDP),
- a key manager (KM) service and
- a server-signing application (SSA).

Using these peripheral services requires that you have the following information ready:

- The base URLs of all services.
- The authentication data (e.g. a shared secret) to authenticate with the IDP.

The device supports the use of

- RSA keys and certificates in conjunction with PSS padding and hash algorithms SHA-256, SHA-384 or SHA-512.

Once these requirements are met, you can go on and configure the gears proNEXT device and its use as described in the following sections.

### 14.11.2 Device Configuration

#### 14.11.2.1 Device provider

The proNEXT device provider has the ID **pronext** and is registered as a Spring bean with the ID **deviceProvider.pronext**. A concrete device can be addressed as **<device id>@pronext**. The device provider cannot be configured.

#### 14.11.2.2 Device properties

A proNEXT device can be configured with these properties:

id	
String required	The id of the new device. Only ASCII-alphanumeric, '_' and '-' is allowed.
authenticator	
Object required	An instance of <code>de.intarsys.security.device.pronext.client.IProNEXTAuthenticator</code> , handling the acquisition of an ID token.
idpUrl	
URL required	The base URL of the identity provider used to fetch authorization tokens, for example <code>https://pronext.myserver.com/PKIManagementService</code> .
keyManagerUrl	
URL required	The base URL of the key manager, for example <code>https://pronext.myserver.com/KeyManager</code> .
ssaUrl	
URL required	The base URL of the server-signing application, for example <code>https://pronext.myserver.com/ServerSigningApplication</code> .
healthMonitor.delay	
integer optional	Number of seconds between two health checks of the CSC service. The default is 300, which amounts to 5 minutes. A value less than 1 disables health monitoring.
credential	

### 14.11.2.3 Authenticator definition

In order to obtain an authorization token for the proNEXT components, gears has to process a set of authentication data provided within a signing request, typically involving the IDP.

Currently there is one authentication mechanism supported:

- Authentication using a signed JWT token.

The following sections provide information on how to apply these authentication mechanisms.

#### 14.11.2.3.1 Authentication using a signed JWT token

Authenticates by

- taking a user ID,
- wrapping it as “sub” claim in a JWT,
- securing the JWT using an HMAC with shared secret and
- sending the resulting token as bearer token to the IDP’s token authentication endpoint.

To activate this feature you can leverage the **de.intarsys.security.device.pronext.client.TokenAuthenticator**.

#### Spring XML fragment

```
<bean id="pronextAuthenticatorToken"
class="de.intarsys.security.device.pronext.client.TokenAuthenticator">
  <constructor-arg value="https://
pronext.myserver.com/PKIManagementService"/>
  <property name="issuer" value="gears" />
  <property name="sharedSecret"
value="\${secret.plain#QVByZXR0eVNlY3JldFNlY3JldFN0cmZ1dWNoSXNmb25nRW5v
dWdoRm9ySFMyNTY}" />
</bean>
```

This definition creates a token authenticator which

- additionally includes an issuer claim with value “gears”,
- uses the shared secret behind `\${secret.plain#QVByZXR0eVNlY3JldFNlY3JldFN0cmZ1dWNoSXNmb25nRW5vdWdoRm9ySFMyNTY}` for JWS derivation and
- addresses a token authentication endpoint at `https://pronext.myserver.com/PKIManagementService/api/auth/token`.

### 14.11.2.4 Default device

The system comes with a preconfigured Azure device, which can be addressed as **default@pronext**. This is a standard suitable for most use cases. It assumes that you authenticate using the token authenticator as described in section 14.11.2.3.1.

The default device properties are configured using the following set of Spring properties that must be provided in your gears.properties file:

pronext.idpUrl	
URL required	Define <i>idpUrl</i> .
pronext.keyManagerUrl	
URL required	Define <i>keyManagerUrl</i> .
pronext.ssaUrl	
URL	Define <i>ssaUrl</i> .

required	
pronext.idp.tokenauth.issuer	
String optional	Default: gears Define the token authenticator's <i>issuer</i> field.
pronext.idp.tokenauth.secret	
String required	Define the token authenticator's <i>sharedSecret</i> field.

A properties file could look like the following:

### Spring properties

```
pronext.idpUrl=https://pronext.myserver.com/PKIManagementService
pronext.keyManagerUrl=https://pronext.myserver.com/KeyManager
pronext.ssaUrl=https://pronext.myserver.com/ServerSigningApplication

pronext.idp.tokenauth.issuer=gears-test
pronext.idp.tokenauth.secret=${secret.plain#QVByZXR0eVNlY3JldFNlY3JldFN0cmZ1doZWNoSXNMb25nRW5vdWdoRm9ySFMyNTY}
```

## 14.11.3 Usage

### Factory

For signature creation, use the generic **de.intarsys.security.app.signature.SignerFactory**.

### Arguments

device	
String required	The id of the device that will create the signer application. For the preconfigured proNEXT device, you should refer to "default@pronext".
authenticationData.userName	
String required	The ID of the user as registered with the proNEXT services.
principalFilter	
String optional	An expression that will select the correct certificate from the set of certificates available in the key manager.

## 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@pronext",
            "authenticationData": {
              "userName": "test-user"
            }
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}

```

## 14.11.4 Signer Configuration

To further abstract from a concrete device and concrete arguments, remember you can define a "signer configuration" (see 9.11.1.1). Please be sure to use a unique id for your configuration.

The gears environment comes with a predefined signer configuration named **pronext**, which uses the **default@pronext** device and looks like this:

## Spring XML fragment

```

<bean
  class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="pronext" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
        value="de.intarsys.security.processor.signature.DocumentSignerFactory" />
      <entry key="documentSigner.args.digestSigner.factory"
        value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device"
        value="default@pronext" />
      <entry key="documentSigner.args.digestSigner.args.authenticationData.userName"
        value="{pronext.signer.userName}" />
    </map>
  </property>
</bean>

```

This predefined configuration can be customized through the use of the following custom property definitions

pronext.signer.userName	
String required	The ID of the user as registered with the proNEXT services.

For example:

### Spring properties

```
pronext.signer.userName=?{principal.user.claims.pronextUsername:!principal.user.name}
```

Instead of explicitly defining all details of the Azure signer as in the example in 14.10.3, you can leverage the predefined configuration:

### service call

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

{
  "configuration": "pronext",
  "options": {
    "principal": {
      "name": "test-user"
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<Base64-encoded PDF document>"
  }
]
```

## 14.11.5 Monitoring

This device supports the standard device properties and notifications.

It does currently not support background health check

## 14.11.6 Observations

This device supports the following observations, which can be processed by Micrometer and exposed via Prometheus (see chapter 16.4).

The observations expand on the standard observations described in chapter 14.1.7.2.

source	device.<provider>.<device>
	<p>The identification of the device that originates the observation.</p> <p>Example</p> <p>"device.proNext.default"</p>
code	
	<p>A property which may resolve to a NLS specific message text.</p> <p>start:</p> <ul style="list-style-type: none"> <li>• "authenticate.authenticateToken.start"</li> <li>• "authenticate.authenticateMtls.start"</li> <li>• "authenticate.refreshToken.start"</li> <li>• "authenticate.listKeys.start"</li> <li>• "sign.augmentIdpToken.start"</li> <li>• "sign.remoteSign.start"</li> </ul> <p>ok:</p> <ul style="list-style-type: none"> <li>• "authenticate.authenticateToken.ok"</li> <li>• "authenticate.authenticateMtls.ok"</li> <li>• "authenticate.refreshToken.ok"</li> <li>• "authenticate.listKeys.ok"</li> </ul>

	<ul style="list-style-type: none"> <li>• "sign.augmentIdpToken.ok"</li> <li>• "sign.remoteSign.ok"</li> </ul> fail: <ul style="list-style-type: none"> <li>• "authenticate.authenticateToken.fail"</li> <li>• "authenticate.authenticateMtls.fail"</li> <li>• "authenticate.refreshToken.fail"</li> <li>• "authenticate.listKeys.fail"</li> <li>• "sign.augmentIdpToken.fail"</li> <li>• "sign.remoteSign.fail"</li> </ul>
operation	
	A property which indicates the kind of security application <ul style="list-style-type: none"> <li>• "sign"</li> <li>• "authenticate"</li> </ul>
subcode	
	A property which indicates the state or outcome of the operation <ul style="list-style-type: none"> <li>• "start"</li> <li>• "ok"</li> <li>• "fail"</li> </ul>
app	
	The security application reference. You can collect further information from this object.

#### 14.11.6.1 Observations for external service calls

During a signature creation process, the proNEXT signer steps through multiple external service calls. Each of the following observations marks a specific moment just before or after an external service call is made. The “.start” suffix indicates the time right before the service call occurs, while “.ok” and “.fail” indicate what happens immediately after the call—whether it succeeded or failed. The table below links each observation code to its corresponding external service endpoint.

Observation code	External service endpoint
authenticate.authenticateToken	/api/auth/jwt
authenticate.authenticateMtls	/api/auth/tls
authenticate.refreshToken	/api/id-token/refresh
authenticate.listKeys	/api/resources/certificates
sign.augmentIdpToken	/api/id-token/sad
sign.remoteSign	/signing

If micrometer is configured (see chapter 16.4) these observations are processed into micrometer timers which can be used for monitoring via e.g. prometheus and grafana.

Here are a few examples for micrometer timers which can result from these observations:

- {\_\_name\_\_="device\_pronext\_default", outcome="ok", instance="integration.intarsys.de:80", job="cloudsuite-core-1", operation="authenticate", step="authenticateToken"}
- {\_\_name\_\_="device\_pronext\_default", outcome="ok", instance="integration.intarsys.de:80", job="cloudsuite-core-1", operation="sign", step="remoteSign"}

- {\_\_name\_\_="device\_pronext\_default", outcome="fail", instance="integration.intarsys.de:80", job="cloudsuite-core-1", operation="sign", step="augmentIdpToken "}



## 14.12 CSC device

### 14.12.1 Overview

The CSC device allows to sign hash values with remote signing services that conform to the Cloud Signature Consortium (CSC) standard version 1.x or 2.x.

The device supports:

- Service and credential authorization with OAuth 2.0
- signatures/signHash with selection of credentials by ID or certificate filter (CSC 1.x or 2.x)
- signatures/signDoc with signature qualifier (CSC 2.x)
- Multi-signing (sign multiple hash values in a single request)

### 14.12.2 Device configuration

#### 14.12.2.1 Device provider

The CSC device provider has the ID **csc** and is registered as a Spring bean with the ID **deviceProvider.csc**. A concrete device can be addressed as **<device id>@csc**. The device provider cannot be configured.

#### 14.12.2.2 Device properties

id	
string required	The device's ID. Only ASCII-alphanumeric characters, '_', and '-' are allowed.
apiBaseUri	
URI required	The base URI of the remote service API. For example, <b>https://example.org/csc/v1/</b> .
oauth2Issuer	
URI optional	The issuer URI of the OAuth 2.0 authorization server supported by the remote service for service and credential authorization. Overrides the oauth2Issuer property returned by the remote service's info endpoint.
oauth2BaseUri	
URI optional	The base URI of the OAuth 2.0 authorization server endpoint supported by the remote service for service and credential authorization. Overrides the oauth2 property returned by the remote service's info endpoint.
authorizationManager	
AuthorizationManager conditional	<p>An authorization manager that handles the service and credential authorization for the device. The value can be either an instance or instance spec of <b>de.intarsys.security.device.csc.auth.AuthorizationManager</b> (see section 14.12.2.2.1 for details).</p> <p>Alternatively, the authorization manager can be specified per signer. If specified per device, the authorization manager will be shared by signers and allows them to reuse service access tokens.</p>
httpConfiguration.sslContext	
SSLContext optional	<p>An SSL context that controls the protocol for TLS connections. For example, the context can specify a client TLS certificate or trusted certificates for server authentication.</p> <p>The value of this property can be either:</p>

	<ul style="list-style-type: none"> <li>an instance of <b>javax.net.ssl.SSLContext</b></li> <li>an instance of <b>de.intarsys.tools.ssl.ISslContextProvider</b> (for example, a <b>de.intarsys.tools.ssl.ConfigurableSslContextProvider</b>)</li> </ul> <p>Default: none</p>
<b>httpConfiguration.hostnameVerifier</b>	
HostnameVerifier optional	<p>An instance of <b>javax.net.ssl.HostnameVerifier</b> to verify the server's identity.</p> <p>Default: none</p>
<b>httpConfiguration.connectTimeout</b>	
integer optional	<p>Timeout in milliseconds for connections to the server. 0 represents infinity.</p> <p>Default: 0</p>
<b>httpConfiguration.readTimeout</b>	
integer optional	<p>Timeout in milliseconds to read a response from the server. 0 represents infinity.</p> <p>Default: 0</p>
<b>httpConfiguration.useProxy</b>	
boolean optional	<p>If true, the HTTP proxy specified by the standard system properties <b>http.proxyHost</b>, <b>http.proxyPort</b>, <b>http.proxyUser</b>, and <b>http.proxyPassword</b> is used to connect to the remote service API.</p>
<b>healthMonitor.delay</b>	
integer optional	<p>Number of seconds between two health checks of the CSC service. The default is 300, which amounts to 5 minutes. A value less than 1 disables health monitoring.</p>

#### 14.12.2.2.1 Authorization Managers

An authorization manager for the CSC remote service that uses OAuth 2.0 can be created with the **de.intarsys.security.device.csc.auth.AuthorizationManagerFactory**.

The OAuth 2.0 authorization manager created by the factory caches and reuses service access tokens. If an access token has expired and the remote service also provided a refresh token, then the authorization manager first tries to refresh the access token before it requests a new one from scratch.

The factory is configured with the following properties:

<b>clientId</b>	
String required	The client ID to use for authorization requests.
<b>clientSecret</b>	
String required	The client secret to use for authorization requests.
<b>accountTokenSupplier</b>	
AccountTokenSupplier optional	An optional supplier of account tokens for authorization requests. The value can be either an instance or instance spec of <b>de.intarsys.security.device.csc.auth.AccountTokenSupplier</b> (see 14.12.2.2.2 for details).
<b>accessTokenLifetimeMargin</b>	
Duration optional	A duration before the actual expiration time in which service access tokens are deemed already expired. Durations can be specified in ISO 8601 format (for example, PT1M30S for 1 minute and 30 seconds) or in

	the simpler format “<value> <unit>” (for example, 90s), where unit can be ns, ms, s, m, h, or d.  Default: 1 minute
callbackUri	
URI optional	The manager’s callback URI for redirections by the remote service during OAuth 2.0 authentication flows. Usually, this URI can be detected automatically by the manager.  Default: auto-detect

#### 14.12.2.2 Account Token Suppliers

Access to a remote service’s authorization server may be restricted and require an account token. Account tokens are secure tokens based on an account ID uniquely assigned to the signing user or the user’s application account (see CSC API specification 1.0.4.0 [7] for details).

Account token suppliers can be created with the **de.intarsys.security.device.csc.auth.AccountTokenSupplierFactory**, which supports the following configuration properties:

clientId	
String required	The unique client ID previously assigned to the signature application by the remote service.
clientSecret	
String required	The client secret previously assigned to the signature application by the remote service.
issuer	
String required	The token’s issuer, for example, the commercial name of the signature application.
subject	
String Required	The client-defined account ID that allows the remote service to identify the account or user initiating the authorization transaction.
tokenLifetime	
Duration Required	Lifetime of account tokens before they expire. Durations can be specified in ISO 8601 format (for example, PT1H30M for 1 hour and 30 minutes) or in the simpler format “<value> <unit>” (for example, 90m), where unit can be ns, ms, s, m, h, or d.  Default: 1 hour

### 14.12.3 Usage

#### Factory

To create a CSC signer, use the generic **de.intarsys.security.app.signature.SignerFactory** and configure it with the following arguments:

#### Arguments

Device	
String Required	The id of the device that will create the signer application.
Method	
signHash   signDoc optional	The method to use to create the signature. The default method is <b>signHash</b> , which is supported by CSC 1.x and 2.x. <b>signDoc</b> requires a remote service that implements CSC 2.x.

authorizationManager	
AuthorizationManager conditional	An authorization manager that handles the service and credential authorization for the signer. If not specified, the device's authorization manager is used (see device properties). An authorization manager (either per device or per signer) is required.

Note: CSC signers do not impose any restrictions or preferences on the hash algorithm. If you are signing a document, you may need to explicitly configure a digester using the **documentSigner.args.digester** argument. Otherwise, the default SHA-256 algorithm will be used.

#### Additional arguments for signHash method

userId	
String conditional	The ID previously assigned by the remote service to the user. This argument shall be used, if and only if the remote service uses one of the auth types <b>external</b> , <b>TLS</b> , or <b>oauth2client</b> .
credentialId	
Object optional	The ID of a credential to use for the signature creation.
certificateFilter	
Object optional	Any object that can be converted to a <b>de.intarsys.security.certificate.filter.IX509CertificateFilter</b> , which is then used to select the credential on the remote service to create the signature.

Note: **credentialId** and **certificateFilter** are mutually exclusive. If neither one is given, the signer will use any credential returned by the remote service's credentials/list endpoint.

#### Additional arguments for signDoc method

signatureQualifier	
Object required	Identifier of the signature type to be created, e.g. "eu_eidas_qes" to denote a Qualified Electronic Signature according to eIDAS.
signatureAlgorithm	
String required	The signature algorithm (name or OID) to use, e.g. "RSAEncryption" or 1.2.840.113549.1.1.1 for RSA.

### 14.12.4 Monitoring

This device supports the standard device properties and notifications.

### 14.12.5 Observations

This device supports the standard observations.

## 14.13 A-Trust device

### 14.13.1 Overview

The A-Trust device is a pre-defined CSC device (see 14.12) with the ID **atrust@csc** and is defined as follows:

#### Spring XML fragment

```
<bean class="de.intarsys.tools.factory.InstanceSpecInstaller">
  <property name="factory" ref="deviceProvider.csc"/>
  <property name="args">
    <map>
      <entry key="id" value="atrust"/>
      <entry key="apiBaseUri" value="\${atrust.baseUri:}"/>
      <entry key="oauth2BaseUri" value="\${atrust.oauth2.baseUri:}"/>
    </map>
  </property>
</bean>
```

### 14.13.2 Device configuration

The A-Trust device can be configured with the following Spring application properties:

atrust.baseUri	
String required	The base URI of A-Trust's CSC service API.
atrust.oauth2.baseUri	
String required	The base URI of A-Trust's OAuth 2.0 endpoints (oauth2/authorize and oauth2/token).  Note: This URI is auto-detected from the metadata provided by the info endpoint. However, A-Trust's service (at least in their test environment) returns an incorrect URI with an extra trailing ".../oauth2/". To fix this, you need to explicitly configure the correct URI to override the faulty metadata.
atrust.oauth2.callbackUri	
String optional	The redirection URI for OAuth 2.0 callbacks. Usually, this URI can be auto-detected by the device itself.
atrust.clientId	
String required	The client ID for authentication requests.
atrust.clientSecret	
String required	The client secret for authentication requests.
atrust.subscriberId	
String required	The subscriber ID assigned by A-Trust to the user or the user's signing application.
atrust.healthMonitor.delay	
integer optional	Number of seconds between two health checks of the A-Trust service. The default is 300, which amounts to 5 minutes. A value less than 1 disables health monitoring.

### 14.13.3 Usage

To create a signer for the pre-defined A-Trust device, use the generic **de.intarsys.security.app.signature.SignerFactory** and pass **atrust@csc** as **device** argument.

#### 14.13.4 Signer configuration

The gears environment comes with a pre-defined signer configuration named **atrust**, which uses the **atrust@csc** device and a timestamp device to create LT-conforming signatures. The configuration is defined as follows:

## Spring XML fragment

```

<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="atrust"/>
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
        value="de.intarsys.security.processor.signature.DocumentSignerFactory"/>
      <entry key="documentSigner.args.digester"
        value="\${atrust.digester:SHA-256}"/>

      <entry key="documentSigner.args.digestSigner.factory"
        value="de.intarsys.security.app.signature.SignerFactory"/>
      <entry key="documentSigner.args.digestSigner.args.device"
        value="atrust@csc"/>

      <entry key="documentSigner.args.digestSigner.args.authorizationManager.factory"
        value="de.intarsys.security.device.csc.auth.AuthorizationManagerFactory"/>
      <entry
key="documentSigner.args.digestSigner.args.authorizationManager.args.clientId"
        value="\${atrust.clientId:UNDEFINED}"/>
      <entry
key="documentSigner.args.digestSigner.args.authorizationManager.args.clientSecret"
        value="\${atrust.clientSecret:UNDEFINED}"/>
      <entry
key="documentSigner.args.digestSigner.args.authorizationManager.args.callbackUri"
        value="\${atrust.oauth2.callbackUri:}"/>

      <entry
key="documentSigner.args.digestSigner.args.authorizationManager.args.accountTokenSupplie
r.factory"
        value="de.intarsys.security.device.csc.auth.AccountTokenSupplierFactory"/>
      <entry
key="documentSigner.args.digestSigner.args.authorizationManager.args.accountTokenSupplie
r.args.subject"
        value="\${atrust.subscriberId:}"/>
      <entry
key="documentSigner.args.digestSigner.args.authorizationManager.args.accountTokenSupplie
r.args.issuer"
        value="\${atrust.clientId:UNDEFINED}"/>
      <entry
key="documentSigner.args.digestSigner.args.authorizationManager.args.accountTokenSupplie
r.args.clientId"
        value="\${atrust.clientId:UNDEFINED}"/>
      <entry
key="documentSigner.args.digestSigner.args.authorizationManager.args.accountTokenSupplie
r.args.clientSecret"
        value="\${atrust.clientSecret:UNDEFINED}"/>

      <entry key="documentSigner.args.conformanceLevel" value="LT"/>
      <entry key="documentSigner.args.timestampDevice" value="atrust@httpTimestamp"/>
    </map>
  </property>
</bean>

<bean factory-bean="deviceProvider.httpTimestamp" factory-method="createInstance">
  <constructor-arg>
    <map>
      <entry key="id" value="atrust"/>
      <entry key="url" value="\${atrust.tss.url:}"/>
      <entry key="user" value="\${atrust.tss.user:}"/>
      <entry key="password" value="\${atrust.tss.password:}"/>
    </map>
  </constructor-arg>
</bean>

```

The timestamp device for this configuration can be customized with the following Spring application properties:

atrust.tss.url	
string required	The URL of the timestamp service.
atrust.tss.user	

string optional	The username for the timestamp service.
atrust.tss.password	
string optional	The password for the timestamp service.

For example:

### Spring properties

```
atrust.tss.url=https://example.org/tss
atrust.tss.user=nobody
atrust.tss.password=nobody's password
```

The pre-defined A-Trust signer configuration can be used as follows:

### service call

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

{
  "configuration": "atrust",
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<Base64-encoded PDF document>"
  }]
}
```

## 14.13.5 Monitoring

This device supports the same device properties and notifications as all CSC devices (see 14.12.4).

It does not support background health check.

## 14.13.6 Observations

This device supports the same observations as all CSC devices (see 14.12.5).



## 14.14 gears Device

### 14.14.1 Overview

The gears device allows delegation of signature creation to a different gears instance. This is a particularly interesting setup whenever you process documents which must not leave a specific environment while you want to leverage the signing devices maintained by a central gears instance.

In order to set up access to the delegate gears instance, you should have the following information ready:

- The base URL of the delegate gears core application.
- The authentication data (e.g. username and password) to authenticate with the delegate gears instance, if required by the latter.

The device's signer implementation processes a digest value which is locally computed by the document signer at work. It sends this hash to the remote instance along with further request data (requested signer configuration, arguments etc.) and receives a CMS-formatted signature result. In consequence, it is suited for the creation of PAdES and CAdES signatures, but not for XAdES signatures.

Be aware that the remote gears instance does not expose certificate details to the client. Thus local processing (e.g. PDF signature appearance creation) cannot be based on certificate information.

The signer supports any combination of hash algorithms, signature algorithms and padding schemes supported by the CMS (PKCS#7) document signer and the effective signature creation device.

Once these requirements are met, you can go on and configure the gears device and its use as described in the following sections.

### 14.14.2 Device Configuration

#### 14.14.2.1 Device provider

The gears device provider has the ID **pronext** and is registered as a Spring bean with the ID **deviceProvider.gears**. A concrete device can be addressed as **<device id>@gears**. The device provider cannot be configured.

#### 14.14.2.2 Device properties

A proNEXT device can be configured with these properties:

id	
String required	The id of the new device. Only ASCII-alphanumeric, '_' and '-' is allowed.
baseUrl	
URL required	The base URL of the delegate gears core application, for example <a href="https://cloudsuite.intarsys.de/cloudsuite-gears/core">https://cloudsuite.intarsys.de/cloudsuite-gears/core</a> .

#### 14.14.2.3 Default device

The system comes with a preconfigured gears device, which can be addressed as **default@gears**. This is a standard suitable for most use cases. It supports HTTP Basic authentication with the remote gears instance, applying statically configured credentials.

The default device properties are configured using the following set of Spring properties that must be provided in your gears.properties file:

gears.baseUrl
---------------

URL required	Define <i>baseUrl</i> .
gears.remote.authorization.user	
String optional	Define a user name to be used in HTTP Basic authentication.
gears.remote.authorization.password	
Secret optional	Define a password to be used in HTTP Basic authentication.

A properties file could look like the following:

### Spring properties

```
gears.baseUrl=https://cloudsuite.intarsys.de/cloudsuite-gears/core
gears.remote.authorization.user=user1
gears.remote.authorization.password=${secret.plain#Zm9v}
```

## 14.14.3 Usage

### Factory

For signature creation, use the generic **de.intarsys.security.app.signature.SignerFactory**.

### Arguments

device	
String required	The id of the device that will create the signer application. For the preconfigured gears device, you should refer to "default@gears".
remote.configuration	
String optional	The name of the signer application to apply remotely. If skipped, the delegate instance's default signer configuration is used.
remote.args	
Object optional	The args to be passed to the delegate instance.
remote.options	
Object optional	The options to be passed to the delegate instance.
remote.variables	
Object optional	The variables to be passed to the delegate instance.
remote.label	
String optional	The label to be passed to the delegate instance.

## 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@gears",
            "remote": {
              "configuration": "demoPlain"
            }
          }
        }
      }
    }
  },
  "documents": [{
    "type": "d",
    "name": "test.txt",
    "content": "QUJDLi4u"
  }]
}

```

## 14.14.4 Signer Configuration

To further abstract from a concrete device and concrete arguments, remember you can define a "signer configuration" (see 9.11.1.1). Please be sure to use a unique id for your configuration.

The gears environment comes with a predefined signer configuration named **gears**, which uses the **default@gears** device and looks like this:

## Spring XML fragment

```

<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="gears" />
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
value="de.intarsys.security.processor.signature.DocumentSignerFactory"/>

      <entry key="documentSigner.args.digestSigner.factory"
value="de.intarsys.security.app.signature.SignerFactory" />
      <entry key="documentSigner.args.digestSigner.args.device" value="default@gears" />
      <entry key="documentSigner.args.digestSigner.args.remote.options"
value="#{flow.options}" />
      <entry key="documentSigner.args.digestSigner.args.remote.variables"
value="#{flow.variables}" />
      <entry key="documentSigner.args.digestSigner.args.remote.configuration"
value="#{gears.remote.configuration:}" />
    </map>
  </property>
</bean>

```

This predefined configuration can be customized through the use of the following custom property definitions

gears.remote.configuration	
String optional	The name of the signer application to apply remotely. If skipped, the delegate instance's default signer configuration is used.

For example:

#### Spring properties

```
gears.remote.configuration=demoPlain
```

Instead of explicitly defining all details of the Azure signer as in the example in 14.10.3, you can leverage the predefined configuration:

#### service call

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

{
  "configuration": "gears",
  "documents": [{
    "type": "d",
    "name": "test.pdf",
    "content": "<Base64-encoded PDF document>"
  }]
}
```

### 14.14.5 Monitoring

This device supports the standard device properties and notifications.

Health check is done by default by polling the spring actuator health endpoint in a concurrent health monitor. You can overwrite the properties if required

Example:

```
<entry key="healthMonitor">
  <map>
    <entry key="delay" value="42" />

    <entry key="jsonPtr" value="/components/deviceProviders/components/signMe/details/devic
es/default" />
  </map>
</entry>
```

### 14.14.6 Observations

This device supports the standard observations.

## 14.15 PKCS #11 device

### 14.15.1 Overview

PKCS #11 devices allow to create signatures with cryptographic devices that support the PKCS #11 API (also called “Cryptoki”). The devices support signatures with RSA, RSA-PSS, and elliptic curves and operate on the following assumptions:

- The device only needs read access.
- Tokens contain both, private key and the corresponding certificates.
- Private keys and certificates are stored as private objects on the token and can only be accessed in a user session after logging in with the correct user PIN.
- Private keys and certificates are matched by their CKA\_ID attribute. A matching pair must share the same unique CKA\_ID.

### 14.15.2 Device configuration

#### 14.15.2.1 Device provider

The PKCS #11 device provider has the ID **pkcs11** and is registered as a Spring bean with the ID **deviceProvider.pkcs11**. A concrete device can be addressed as **<device id>@pkcs11**. The device provider cannot be configured.

#### 14.15.2.2 Device properties

Id	
string required	The device's ID. Only ASCII-alphanumeric characters, '_', and '-' are allowed.
module	
string required	The name of the PKCS #11 module to be used to access the cryptographic device. See section 14.15.2.3 on how to register PKCS #11 modules.
slot	
integer conditional	ID of the slot that contains the token to be used with this device. Alternatively, the slot can be selected by the label of its token (see property tokenLabel).
tokenLabel	
string conditional	The label of the token to be used with this device. Alternatively, a slot can be selected by its ID.
userPin	
string required	The PIN required to log in to a user session.

#### 14.15.2.3 PKCS #11 modules

The native PKCS #11 modules are registered in Gears with the following configuration properties:

**pkcs11.modules.<module name>=<name or absolute path of the library>**

Note that both, module name and library need to be unique. You cannot register multiple modules with same name or library path.

### 14.15.3 Usage

#### Factory

To create a PKCS #11 signer, use the generic **de.intarsys.security.app.signature.SignerFactory** and configure it with the following arguments:

#### Arguments

device	
String required	The id of the device that will create the signer application.
signerIdentifier	
IX509CertificateFilter optional	Identifies the pair of private key and certificate for the signature. By default, the first private key with a certificate found on the token is used.
rsaSignatureEncoding	
enum (RSA   RSAPSS) optional	For signatures with an RSA private key, this property specifies the signature encoding, which is either RSA (default) or RSA-PSS.

### 14.15.4 Signer configuration

The gears environment includes a pre-defined signer configuration named **pkcs11**, which is defined as follows:

#### Spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.signer.impl.FlowSignerConfiguration">
  <property name="id" value="pkcs11"/>
  <property name="definitions">
    <map>
      <entry key="documentSigner.factory"
        value="de.intarsys.security.processor.signature.DocumentSignerFactory"/>
      <entry key="documentSigner.args.digestSigner.factory"
        value="de.intarsys.security.app.signature.SignerFactory"/>

      <!-- Module and slot are given by the device. -->
      <entry key="documentSigner.args.digestSigner.args.device"
        value="{pkcs11.device}"/>

      <!--
        The signer certificate (and thus the private key) can be selected with
        signerIdentifier. By any key found in the device's slot is selected.
      -->
      <entry key="documentSigner.args.digestSigner.args.signerIdentifier"
        value="{pkcs11.signerIdentifier}" />
    </map>
  </property>
</bean>
```

This configuration can be customized with the following Spring application properties:

pkcs11.device	
string required	The ID of the PKCS #11 device for the signature creation.
pkcs11.signerIdentifier	
string optional	The signer identifier to select the private key on the device's token for the signature creation.

### 14.15.5 Monitoring

This device supports the standard device properties and notifications.  
It does currently not support background health check

### 14.15.6 Observations

This device supports the standard observations.

# 15. Integration

## 15.1 Overview

An application provides a lot of non-functional information that may be required for planning, monitoring and accounting its operation by completely independent and very client specific processes and applications.

We try to adopt a universal design for

- generating
- collecting
- processing
- serialization

this non-functional data based on very flexible concept to support integration: "observations".

In our terminology an **observation** is an "event" that originates at a **source**, is identified by a **code** and carries along observation specific **properties**. This **observation** is broadcast to a bus where interested **observers** can subscribe and consume the observation.

This is a well-known system design, leading to flexible components, loose coupling and lightweight integration scenarios.

## 15.2 Model

### 15.2.1 Observation

Observations are created as "meta information" while processing requests/events or housekeeping procedures.

Observations have at least a **source**, **code** (which may resolve to a NLS specific message text) and **created** property, along with other generic name/value **properties**.

Examples for such meta information:

Request processing properties

- duration
- "size"
- "cost"

Processing context properties

- user
- tenant
- date/time
- application defined tagging

Processing course

- input



- decisions (e.g. consent or denial)
- output
- outcome

This data is not part of the gears "business requirements", but is needed out of band for other operational processes like accounting or auditing.

To provide this information to 3<sup>rd</sup> party and keep these requirements separate from gears, an observation is created and published to the observation runtime where registered observers are triggered.

Basic observation properties

source	
string	The source of the observation, e.g. the application or a specific device
code	
string	The specific technical code for this observation in the source, e.g. "started" or "item.created".
text	
string	An optional clear text internationalized text associated with the code
created	
integer	A timestamp

### 15.2.2 Observer

An observation is observed by an observer. There's no requirement on what and when the observer does with the information. The observer does not feedback reply into the observation process.

The observer registers itself for an observation source. When an observation from a registered source is encountered, it is forwarded to the observer.

An observer has some typical tasks to perform on the observation properties (not all of them are required for all observers):

View creation

- selecting and enhancing the context information

Serialization

- log file
- database
- monitoring facility (MBean, Actuator)
- push request (webhook)

Integrity

- add integrity information for inter & intra observation validation (audit log)

Confidentiality

- observation encryption

### 15.2.3 Filter

An observer may filter the observations it receives.

Any observation may be accepted or rejected based on matching observations properties and context information against a regex pattern.

## 15.2.4 View

After an observation is received, an observer may be interested in creating its individual view on the observation properties and other contextual properties.

These are examples of views that we may want to create

Observation "request finished"

- created
- duration
- bytes read
- bytes sent
- endpoint
- conversation/session context

Observation "signature created"

- tenant
- user
- no. of documents

Creating a view will be configurable for the observer based on

- selecting properties from the observation
- selecting properties from execution context (string expression evaluation)
- filtering/transforming/aggregating properties (string manipulation, concatenation, other functions)

It is an important feature that besides the observation properties a view can add other contextual information via the string expansion mechanics.

## 15.3 Implementation

The implementation is based on an industry proven component that quite closely matches the goals outlined above: the logging framework "logback".

"logback" is specialized in high volume processing of logging events that carry along meta information about the processing. These events can be very flexibly mapped to so called "appenders" that handle the tasks of an observer.

## 15.4 Observer definition

An observer and the associated logback appender is defined using a spring bean of type

```
de.intarsys.tools.observation.logback.LogbackObserver
```

This bean defines all properties required to instrument logback and the intended observer behavior.

A LogbackObserver has the following properties:

source	
string	A (partially) defined source path. All observations that start with this path are forwarded to the observer.
filter	
Filter	An optional filter definition.
view	
View	An optional view definition.
appender	

logback appender	<p>A definition for a logback appender.</p> <p>All properties defined by the view are forwarded to the appender, in the order of definition. If no view is defined, all properties of the observations are used by default.</p> <p>You can define here either a literal observer or a name of an existing observer of the current logback configuration. Be aware that in this case you can only use an appender that is already attached to a logger in the current logback configuration. The appender must be either attached to the root logger <b>or</b> you reference the appender in a path notation with the logger as a leading segment.</p> <p>Example:</p> <ul style="list-style-type: none"> <li>• "FILE" Use the appender "FILE" attached to the root logger</li> <li>• "foobar/AUDIT" Use the appender "AUDIT" attached to the logger "foobar"</li> </ul>
------------------	---

Example observer definition for any device originated observation with a literal logback appender:

#### Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device" />
  <property name="appender">
    <bean class="ch.qos.logback.core.FileAppender">
      <property name="name" value="FILE" />
      <property name="file" value="${cloudsuite.log.dir}/device.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} [%.-1p] [%c] %msg%n" />
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

Example observer definition for any device originated observation with a reference to an existing appender:

#### Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device" />
  <property name="appender" value="mylogger/FILE"/>
</bean>
```

## 15.5 Filter definition

A filter accepts or rejects a single observation before it is forwarded to the appender.

The filter is defined using a bean of type

```
de.intarsys.tools.observation.impl.ObservationFilter
```

The "filter" property of the observer definition is polymorphic and accepts

- An ObservationFilter
- A list of predicates

- A single predicate

These predicates are available

- Accept  
Accept an observation based on matching an expression
- Reject  
Reject an observation based on matching an expression

The filter is defined as a logical "and" of all defined predicates.

### 15.5.1 Accept

All observations where the evaluated "expr" matches the given regex pattern are accepted.

---

#### Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="license"/>
  <property name="filter">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AcceptPredicate">
        <property name="expr" value="{observation.code}"/>
        <property name="pattern" value="withdraw"/>
      </bean>
    </list>
  </property>
  ...
</bean>
```

Here all observations with a "code" of "withdraw" are accepted.

### 15.5.2 Reject

All observations where the evaluated "expr" matches the given regex pattern are rejected.

---

#### Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="license"/>
  <property name="filter">
    <bean class="de.intarsys.tools.observation.impl.RejectPredicate">
      <property name="expr" value="{observation.code}"/>
      <property name="pattern" value="withdraw"/>
    </bean>
  </property>
  ...
</bean>
```

Here all observations with a "code" of "withdraw" are rejected.

## 15.6 View definition

A view defines the data that is made available from the observation event and context to the appender.

If no view is defined, by default all properties of the observation are forwarded to the observer.

A view is defined using a bean of type

---

```
de.intarsys.tools.observation.impl.ObservationView
```

---

The "view" property of the observer definition is polymorphic and accepts

- An ObservationView
- A list of statements
- A single statement

We have actually three primitives

- Include  
Include properties from the observation using a regex pattern
- Exclude  
Exclude properties from the observation using a regex pattern
- Add Property  
Add a named property by evaluating a string

These primitives are additive.

## 15.6.1 Include

All observation properties that match a given regex pattern are included in the view

---

### Spring XML fragment

---

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="license"/>
  <property name="view">
    <bean class="de.intarsys.tools.observation.impl.IncludeStatement">
      <property name="pattern" value="c..e"/>
    </bean>
  </property>
  ...
</bean>
```

---

Add all properties that match "c..e" (which is only "code").

## 15.6.2 Exclude

All observation properties that match a given regex pattern are excluded from the view

---

### Spring XML fragment

---

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="license"/>
  <property name="view">
    <bean class="de.intarsys.tools.observation.impl.ExcludeStatement">
      <property name="pattern" value="c..e"/>
    </bean>
  </property>
  ...
</bean>
```

---

Add all properties that match "c..e" are excluded, which means normally all observation properties except "code" are used.

## 15.6.3 Add property

Add a dedicated name/value pair. The value definition is expanded.

You have access to the standard string expansion namespaces like "**flow.\***" as well as the observation properties via the "**observation**" namespace.

Attention:

Be sure to use the "{}" notation for deferred string expansion.

Example

---

### Spring XML fragment

---

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="license"/>
  <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="fromObservation"/>
        <property name="value" value="{observation.code}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="fromFlow"/>
        <property name="value" value="{flow.variables.xy}"/>
      </bean>
    </list>
  </property>
  ...
</bean>
```

This example defines three properties

- static  
With the literal string "some value"
- fromObservation  
Holds the value "code" from the current observation
- fromFlow  
Holds the value of the flow variable "xy"

## 15.6.4 Complete observer example

This is a complete observer definition:

## Spring XML fragment

```

<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device"/>
  <property name="filter">
    <bean class="de.intarsys.tools.observation.impl.RejectPredicate">
      <property name="expr" value="{observation.code}"/>
      <property name="pattern" value="withdraw"/>
    </bean>
  </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="fromObservation"/>
        <property name="value" value="{observation.code}"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="fromFlow"/>
        <property name="value" value="{flow.variables.xy}"/>
      </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}/device.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} [%.-1p] [%c]
%msg%n"/>
        </bean>
      </property>
    </bean>
  </property>
</bean>

```

## 15.7 Appender definition

### 15.7.1 Plain logging

You can attach any of the well-known logback appenders to integrate in your own system scenario.

Examples of such appenders are

- File logging
- Syslog
- Logstash

Example configuration

## Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="" />
  <property name="appender">
    <bean class="ch.qos.logback.core.FileAppender">
      <property name="name" value="FILE" />
      <property name="file" value="\${cloudsuite.log.dir}/observation.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} [%.-1p] [%c] %msg%n" />
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

This configuration will log any observation in gears to a file "observation.log". Note that the logback DSL cannot be used here!

You must have intimate knowledge of logback data structures to create such a configuration. Please note that not all appenders may work with this approach because of logback internals.

This will result in a "observation.log" file in the log file directory of gears after starting that reads:

```
[18:37:24.003] [I] [observation.application] {started}
```

## 15.7.2 Pattern conversion for args

Logback does not have support for directly addressing the arguments of a log event – something that is important for the use case we are working with here.

That's why we have added two conversion tokens to the PatternLayout, "arg" and "args", to access the arguments.

### 15.7.2.1 args

The "args" token simply adds a ";" separated list of all arguments.

Pattern example:

```
<property name="pattern" value="%d{HH:mm:ss.SSS} [%msg] [%args] %n" />
```

This will result in something similar to

```
[18:41:06.044] [hello, world] [created=1619109666043;source=license;code=loaded]
```

### 15.7.2.2 arg

The "arg" token needs an option that designates the argument to be replaced. The option may be the index or name of the required argument.

Pattern example:

```
<property name="pattern" value="%d{HH:mm:ss.SSS} [%msg] %arg{source} %arg{code} %n" />
```

This will result in something similar to



```
[18:41:06.044] [hello, world] license withdraw
```

### 15.7.3 Other appenders

You can find a description of the existing logback appender implementations here

```
http://logback.qos.ch/manual/appenders.html
```

Please note that the logback DSL cannot be used here!

You must have intimate knowledge of logback data structures to create such a configuration. Please note that not all appenders may work with this approach because of logback internals.

### 15.7.4 Logstash support

"logstash" (see <https://github.com/logstash/logstash-logback-encoder>) provides sophisticated support for using JSON formatted log output. This framework is included by default in gears to ease integration.

This enables you to add a logstash encoder in your appender definition out of the box.

This definition will create a JSON log with the logstash default.

#### Spring XML fragment

```
<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="net.logstash.logback.encoder.LogstashEncoder">
    </bean>
  </property>
</bean>
```

You can customize the output in a very detailed way, see the documentation linked above for more.

To have an easy integration with the observation component, a special "provider" implementation is included that acts on the observation properties the same way as logstash on its StructuredValue.

The

```
de.intarsys.tools.logging.logback.logstash.ArgumentsJsonProvider
```

will add all properties to the JSON created. You can use this provider according to the logstash documentation.

To make life a little bit simpler, a predefined encoder comes with gears that adds this provider by default:

```
de.intarsys.tools.logging.logback.logstash.DefaultJsonEncoder
```

This definition will create a JSON log with all observation properties as fields.

## Spring XML fragment

```
<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>
```

## Example output

```
{
  "source": "device.demo.default",
  "timestamp": 1621711667993,
  "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":
"K3lyVTU2dnByRk1KSE41T0VHL2taTk0vbXhhRUM3UklGaWhFNGx5QU83dz0="
      }
    }
  ]
}
```

## 15.7.5 Webhook

This appender is specific to gears and relies on the webhook concept that is described in detail in another chapter.

To attach a webhook to an observer you either reference its "id" or create a literal webhook in the property element.

## Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source" value="device"/>
  <property name="appender">
    <bean class="de.intarsys.tools.webhook.logback.WebhookAppender">
      <property name="webhook" value="testhook"/>
    </bean>
  </property>
</bean>
```

This bean will ensure that each time an observation is made by a device, the "testhook" webhook is called.

The webhook payload is created as a map of all observation properties available. If required, you can create a view on the observation to build a payload matching your demands.

## 15.8 Example

There is an example configuration in the gears "example" folder (gears observations) that defines the above-mentioned observation primitives.

## 15.9 Observations

### 15.9.1 Overview

This chapter describes the common observations that are supported generally by gears so far.

For some system components like devices there may be special additional observations to be made. You can find their description in the respective chapters.

### 15.9.2 application

source	application
	Observations for the application process
code	started
	This is issued when the application is up and running
-	
	no other properties provided

### 15.9.3 license

source	license
	Observations for the license component
code	loaded
	This is issued to indicate that a license was loaded
locator	
	The location from where this license was loaded
product	
	The product for this license
validFrom	
	The start date of license validity Format mm/dd/yy
validTo	
	The end date of license validity Format mm/dd/yy
state	
	The license state na   test   invalid   valid
properties	
	A string with the serialized properties attached to the license

source	license
	Observations for the license component
code	withdraw
	This is issued to indicate that a license account has been touched
amount	
	The amount that is withdrawn from the license account
product	
	The associated product
property	
	The associated property
limit	

	The limit for this account -1 for no limit
balance	
	The balance (remaining amount) for this account -1 for unlimited
nextReset	
	For "repeating licenses" the next instant when this account is reset.

## 15.10 Webhook

### 15.10.1 Overview

A "webhook" is a loosely specified term for integrating applications via HTTP based calls.

Typical applications are for example a GIT server that supports "calling out" to 3<sup>rd</sup> party systems upon certain events like a commit or (webhook client) or a collaboration application like "slack" that can triggered by simple HTTP calls to broadcast a message (webhook server).

In <https://glaforge.appspot.com/article/implementing-webhooks-not-as-trivial-as-it-may-seem> you can have a good overview of the techniques required to use webhooks.

### 15.10.2 Webhook stub

The webhook stub is your gears hosted template for executing an outgoing webhook call.

A webhook can be defined literally within a containing definition (like a webhook appender) or as a standalone template that can be looked up in a registry.

#### 15.10.2.1 Type

The bean type is

```
de.intarsys.tools.webhook.impl.StandardWebhookStub
```

#### 15.10.2.2 Properties

id	
string	An id that may be used to look up the webhook in a registry.  Any stub with an id is registered in the webhook registry.
endpointUrl	
string	The complete URL for the endpoint to be called.  Example: <code>https://myhost/webhook</code>
async	
Boolean	Flag if this webhook is executed asynchronous. This means that the caller is not waiting for the request to succeed or fail  Default: false
connectTimeout	
integer	The maximum time in milliseconds to wait for a successful connect to the webhook server.  -1 means no change to the system default

	0 means infinity Default: -1
<b>readTimeout</b>	
integer	The maximum time in milliseconds to wait for data from the webhook server to arrive.  -1 means no change to the system default 0 means infinity Default: -1
<b>followRedirects</b>	
boolean	Flag if the caller follows redirects returned by the webhook server.  Default: false
<b>method</b>	
string	The HTTP method to be used for contacting the webhook server.  Must be one of GET   POST Default: GET
<b>swallowExceptions</b>	
Boolean	A flag if exceptions are swallowed or propagated to the calling code.  Default: true
<b>sslContextProvider</b>	
ISslContextProvider	A factory object for the SSL context to be used for webhook connections.  Default: The global gears SSL context provider.

Example bean configuration

#### Spring XML fragment

```
<bean class="de.intarsys.tools.webhook.impl.StandardWebhookStub">
  <property name="id" value="testhook" />
  <property name="endpointUrl" value="http://www.google.com/observe" />
</bean>
```

### 15.10.3 Execution semantics

The intention of a webhook is event propagation, not (bidirectional) communication.

As a result, the execution semantics are normally

- Synchronous (to ensure execution order)  
This can be changed using the "async" property
- Results are ignored
- Errors are logged (not propagated)  
This can be changed using the "swallowExceptions" property

#### 15.10.3.1 Webhook arguments

The arguments to a webhook execution are collected in an event object. The properties of this object depend on the execution context.

### 15.10.3.2 HTTP GET

All arguments to the standard webhook stub are serialized as query parameters and sent along with the GET.

#### Example

With a webhook stub defined as outlined above a call with this object

```
{  
  "foo": "bar"  
}
```

will result in a HTTP call like this

```
GET http://www.google.com/observe?foo=bar
```

### 15.10.3.3 HTTP POST

All arguments to the standard webhook stub are serialized as a JSON object and sent along with the POST.

#### Example

With a webhook stub defined as outlined above a call with this object

```
{  
  "foo": "bar"  
}
```

will result in a HTTP call like this

```
POST http://www.google.com/observe  
Accept: */*  
Content-Type: application/json  
{ "foo": "bar" }
```

## 15.10.4 SSL/TLS

Its best practice to always use TLS for production webhooks.

This is a simple way to improve integrity and confidentiality of your communication backbone.

The webhook client uses by default the standard SSL environment of gears.

If required, you can customize the SSL context for every webhook stub using an `ISslContextProvider` (see `ISslContextProvider`).

This is an example that will support client authenticated TLS

## Spring XML fragment

```
<bean class="de.intarsys.tools.webhook.impl.StandardWebhookStub">
  <property name="method" value="POST"/>
  <property name="endpointUrl" value="https://xxxx.x.pipedream.net/" />
  <property name="sslContextProvider">
    <bean class="de.intarsys.tools.ssl.ConfigurableSslContextProvider">
      <property name="keyStoreName"
value="\${cloudsuite.config.shared}/webhook.jks"/>
      <property name="keyStoreType" value="JKS"/>
      <property name="keyStorePassword" value="client"/>
      <property name="keyPassword" value="client"/>
    </bean>
  </property>
</bean>
```

### 15.10.5 Authentication

Authentication is an important concept for implementing a webhook based infrastructure.

While TLS ensures integrity and confidentiality, you must be sure that the events that you will feed into some inhouse business process are coming from a trusted source.

When working in a data center without external access to the webhook servers you can omit this topic. In any externally reachable scenario, you should put one of these strategies in place.

Be aware that on the client side an authentication failure is not checked (see execution semantics).

#### 15.10.5.1 Basic auth

You can use basic authentication for your webhook service by configuring the authenticator like this

## Spring XML fragment

```
<bean class="de.intarsys.tools.webhook.impl.StandardWebhookStub">
  <property name="id" value="testhook" />
  <property name="endpointUrl" value="http://www.google.com/observe" />
  <property name="authenticator">
    <bean class=" de.intarsys.tools.webhook.authenticator.BasicAuthAuthenticator">
      <property name="user" value="gnu"/>
      <property name="password" value="gnat"/>
    </bean>
  </property>
</bean>
```

The resulting request will carry an "Authentication: Bearer" header.

#### Example GET

```
GET http://localhost:56161/test/webhook/get?foo=bar
Authorization: Basic Z251OmduYXQ=
```

#### Example POST

```
POST http://localhost:56161/test/webhook/post
Authorization: Basic Z251OmduYXQ=
Content-Type: application/json

{"foo":"bar"}
```

#### 15.10.5.2 Api token

You can use a static API token like this:

---

**Spring XML fragment**

---

```
<bean class="de.intarsys.tools.webhook.impl.StandardWebhookStub">
  <property name="id" value="testhook" />
  <property name="endpointUrl" value="http://www.google.com/observe" />
  <property name="authenticator">
    <bean class="de.intarsys.tools.webhook.authenticator.ApiTokenAuthenticator">
      <property name="token" value="plain#d3Vyc3RzYWxhdA"/>
    </bean>
  </property>
</bean>
```

---

The resulting request will carry an "Authentication: Bearer" header.

**Example GET**

---

```
GET http://localhost:49943/test/webhook/get?foo=bar
Authorization: Bearer wurstsalat
```

---

**Example POST**

---

```
POST http://localhost:49943/test/webhook/post
Authorization: Bearer wurstsalat
Content-Type: application/json

{"foo":"bar"}
```

---

### 15.10.5.3 TLS client authentication

You can use the SSL/TLS configuration described in the chapter before to set up an authenticating client.



# 16. Monitoring

## 16.1 Overview

"Monitoring" is the process of publishing internal state to external processes.

There are numerous tools and standards to support this process. Three of the most prevalent standards, "Java Management Extensions" (JMX), "Spring Actuator" (V 2.1.x) and micrometer are built-in in gears.

## 16.2 JMX

JMX monitoring is part of the Java runtime. Gears injects the monitoring objects, named "MBeans", simply into the existing management infrastructure.

You should refer to 3<sup>rd</sup> party documentation like

<https://www.oracle.com/technical-resources/articles/javase/jmx.html> , **Getting Started with Java Management Extensions (JMX)**

for more details.

### 16.2.1 Installation

There is no installation step required for JMX support.

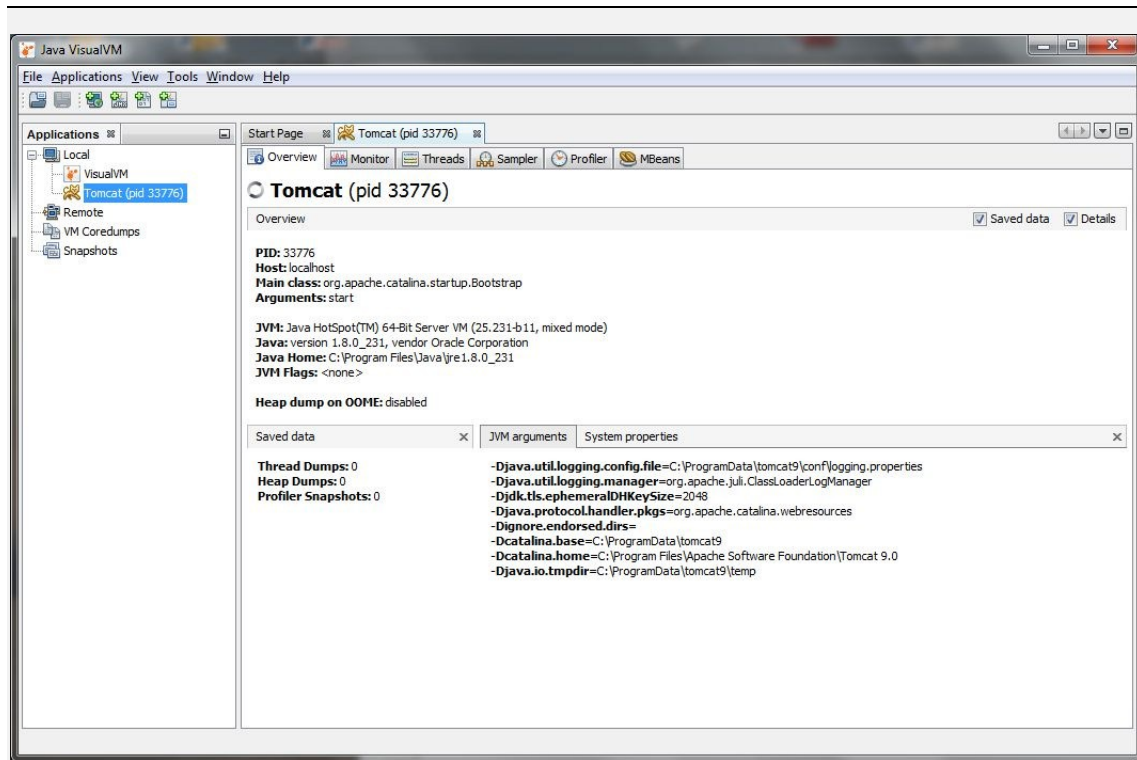
### 16.2.2 Configuration

There is no configuration step required for JMX support

### 16.2.3 Client

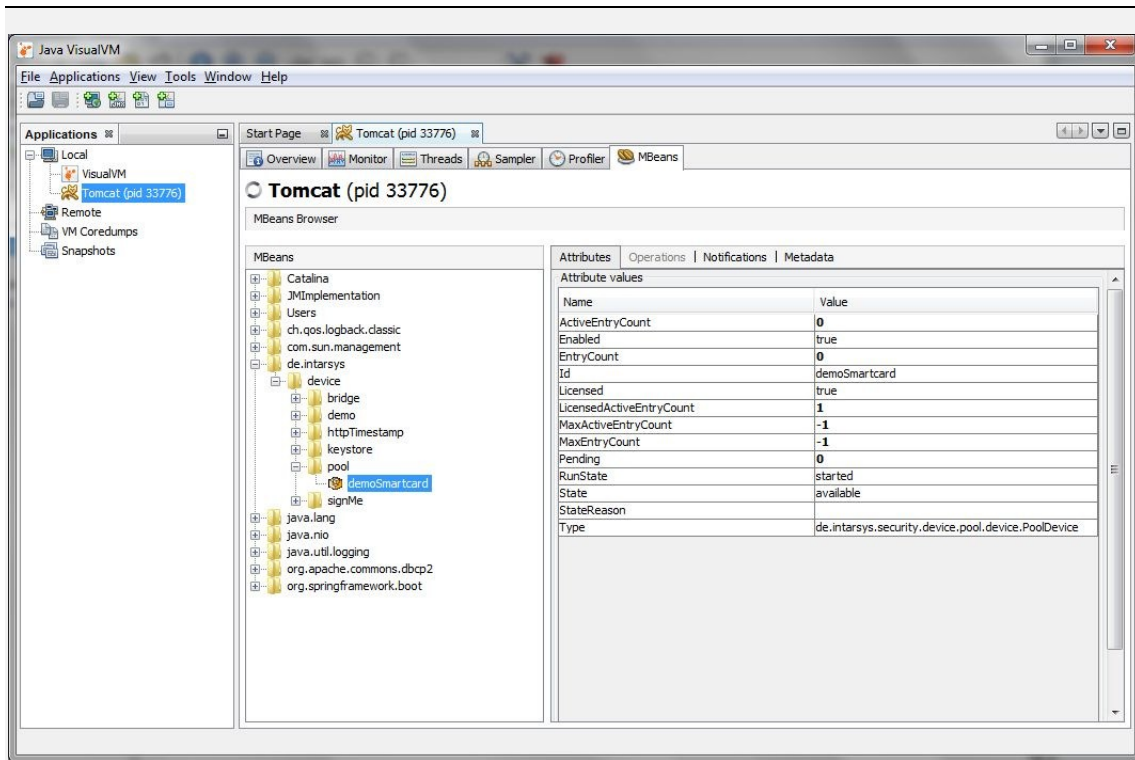
There are numerous client products for dealing with JMX, both open source and commercial.

For a quick look at the JMX server state you can leverage the standard tool "visualvm". After starting and connecting to the gears process (normally the process of the enclosing servlet container instance), you will see a window like this.



The last tab will bring you to the JMX registry. Here you can navigate down the path to the MBeans and see the published properties. If this tab is not visible, you may need to install the "MBeans" plugin from the "Tools->Plugins" menu.

On the sub-tab "Notifications" you can start a subscription to the notifications provided by the MBean.



## 16.2.4 MBean Deployment

An MBean is deployed into the JX runtime using a "domain" and some properties, resulting in an "ObjectName".

The domain used by gears is "de.intarsys".

The path to the MBean is constructed from the properties defined for the MBean.

## 16.2.5 gears MBeans

### 16.2.5.1 device

For every device, a MBean is deployed in the JMX runtime.

The JMX ObjectName properties of a device MBean are:

- type = "device"
- device-provider = <device provider id>
- device = <device id>

The MBean structure is described in the "Monitoring" chapter of the respective device documentation. The same information is made available for other monitoring protocols, e.g. Spring Actuator.

### 16.2.5.2 License

For every valid license (i.e. issued by intarsys and in its validity period), a MBean is deployed.

The JMX ObjectName properties of a device MBean are:

- type = "License"

The fields available for a "License" are

Owner	
String	The owner field of the license
product	

String	The product that is referenced by the license
productVersion	
String	The version of the product
Properties	
List of String	The properties that are attached to the license
ValidFrom	
String	The start of license validity
ValidTo	
String	The end of license validity
valid	
String	Flag indicating if this license is considered valid

The same information is made available for other monitoring protocols, e.g. Spring Actuator.

### 16.2.5.3 LicenseAccount

For every license account that is required by the application (i.e. a counter that is used for counting usage), a MBean is deployed.

The JMX ObjectName properties of a device MBean are:

- type = "LicenseAccount"

The fields available for a "LicenseAccount" are

Balance	
Int	The current balance of the account
Label	
String	A descriptive name for the account
Limit	
Int	The maximum balance for the account
NextReset	
Date	If applicable, the next time the account is reset to its limit
Product	
String	The product containing the account
Property	
String	The account id (the license property that is required)
Unit	
String	The unit for the account limit/balance

The same information is made available for other monitoring protocols, e.g. Spring Actuator.

## 16.3 Spring actuator

You should have a good understanding of the Spring Actuator framework to use this feature correctly.

You find an in-depth description at <https://docs.spring.io/spring-boot/docs/2.1.1.RELEASE/reference/html/production-ready.html> and <https://docs.spring.io/spring-boot/docs/2.1.1.RELEASE/actuator-api/html/>.

### 16.3.1 Installation

The Spring Actuator monitoring feature is provided by installing (or removing) the "intarsys-cloudsuite-gears-module-manage.jar" to/from the web application libraries.

If this jar file is available, the dispatcher servlet responsible for handling the actuator specific requests is deployed.

The request path for actuator requests is

**`http://<host>/<gears context>/manage/*`**

### 16.3.2 Configuration

The installation provides the capability to eventually serve actuator services. To get them deployed and enabled, you have to add some property definitions.

These are the **only** settings that gears injects by default into the Spring configuration.

#### Spring properties

```
# see https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-monitoring.html
management.endpoints.web.base-path=/manage

# see https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html
management.endpoint.health.show-details=ALWAYS
```

To enable Spring actuator for your specific use, follow the instructions in the Spring documentation. Basically, you can apply any Spring specific configuration tweak here. Be sure to always consider the security aspects of your configuration.

### 16.3.3 Endpoints

#### 16.3.3.1 info

Request

**`http://<host>/<gears context>/manage/info`**

Response (with the **demo** profile activated)

```
{
  "gears": {
    "message": "Sign Live! cloudsuite gears"
  }
}
```

#### 16.3.3.2 health

The health endpoint provides application status information. Besides the standard Spring boot health indicators, gears adds more indicators to signal application state.

The additional gears endpoints are documented below.

Request

**`http://<host>/<gears context>/manage/health`**

Response

```

{
  "status": "UP",
  "components": {
    "deviceProviders": {
      "status": "UP",
      "components": {
        "demo": {
          "status": "UP",
          "details": {
            "devices": {
              "default": {
                "status": "UP",
                "details": {
                  "id": "default",
                  "type": "de.intarsys.security.device.demo.device.DemoDevice",
                  "state": "available",
                  "stateReason": null,
                  "licensed": true,
                  "enabled": true
                }
              }
            },
            "id": "demo",
            "type": "de.intarsys.security.device.demo.device.DemoDeviceProvider",
            "enabled": true
          }
        },
        ...
        "pool": {
          "status": "UP",
          "details": {
            "devices": {
              "demoSmartcard": {
                "status": "UP",
                "details": {
                  "id": "demoSmartcard",
                  "type": "de.intarsys.security.device.pool.device.PoolDevice",
                  "state": "available",
                  "stateReason": null,
                  "licensed": true,
                  "enabled": true,
                  "runState": "started",
                  "activeEntryCount": 0,
                  "entryCount": 0,
                  "licensedActiveEntryCount": 3,
                  "maxActiveEntryCount": -1,
                  "maxEntryCount": -1,
                  "pending": 0
                }
              }
            },
            "id": "pool",
            "type": "de.intarsys.security.device.pool.device.PoolDeviceProvider",
            "enabled": true
          }
        }
      }
    },
    "db": {
      "status": "UP",
      "details": {
        "database": "H2",
        "hello": 1
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 510770802688,
        "free": 242670993408,
        "threshold": 10485760
      }
    }
  }
}

```

Notice:

It is a Spring feature to drill into the first two hierarchy levels of the health tree by adding the respective property name from the details. This feature works only if the hierarchy is built using "composite health indicators". As device providers are built using this technique, you can drill down like

---

```
http://<host>/<gears context>/manage/health/deviceProviders/pool
```

---

to select for example the health of a single device provider.

## 16.3.4 gears Health objects

### 16.3.4.1 deviceProviders

The main health detail entry point is "deviceProviders", containing a collection of all device providers installed.

#### Status

The status is derived as follows:

"UP"

- when all children are up

"DOWN"

- otherwise

#### Details

<device provider id>	
device provider health indicator	A health indicator for the device provider

#### Example

```
"deviceProviders": {
  "status": "UP",
  "components": {
    "deviceProvider1": {
    },
    "deviceProvider2": {
    },
    ...
  }
}
```

### 16.3.4.2 deviceProvider

This indicator holds status information for a single device provider, references in the "deviceProviders" health object.

#### Status

The status is derived as follows:

"UP"

- when !enabled
- **or** when all children are up

"DOWN"

- otherwise

#### Details

id	
string	The configured id for this device provider
type	
string	The implementation type of the device provider
enabled	
Boolean	flag if this device provider is enabled in the configuration. A device provider is enabled by default, see device configuration for more details
devices	
collection of device health indicator	A collection of all devices for this device provider.

#### Example

```
"deviceProviders": {
  "status": "UP",
  "components": {
    "deviceProvider1": {
      "status": "UP",
      "details": {
        "devices": {
          "device1": {
            ...
          }
        },
        "id": "demo",
        "type": "de.intarsys.security.device.demo.device.DemoDeviceProvider",
        "enabled": true
      }
    },
    ...
  },
  ...
}
```

#### 16.3.4.3 device

This indicator holds status information for a single device, referenced in the "deviceProvider.devices" health object.

The structure is described in the "Monitoring" chapter of the respective device documentation. The same information is made available for other monitoring protocols, e.g. JMX.

#### Status

The status is derived as follows:

"UP"

- when !enabled
- **or**
  - licensed && state=="available"
  - **and** (for pools)
    - runState != "stopped"
    - entryCount > 0
    - licensedActiveEntryCount != 0

"DOWN"

- otherwise



## Example for a demo device

```

"deviceProviders": {
  "status": "UP",
  "components": {
    "deviceProvider1": {
      "status": "UP",
      "details": {
        "devices": {
          "default": {
            "status": "UP",
            "details": {
              "id": "default",
              "type": "de.intarsys.security.device.demo.device.DemoDevice",
              "state": "available",
              "stateReason": null,
              "licensed": true,
              "enabled": true
            }
          }
        },
        "id": "demo",
        "type": "de.intarsys.security.device.demo.device.DemoDeviceProvider",
        "enabled": true
      }
    },
    ...
  }
}

```

## Example for a pool device

```

"deviceProviders": {
  "status": "UP",
  "components": {
    "pool": {
      "status": "UP",
      "details": {
        "devices": {
          "demoSmartcard": {
            "status": "UP",
            "details": {
              "id": "demoSmartcard",
              "type": "de.intarsys.security.device.pool.device.PoolDevice",
              "state": "available",
              "stateReason": null,
              "licensed": true,
              "enabled": true,
              "runState": "started",
              "activeEntryCount": 0,
              "entryCount": 1,
              "licensedActiveEntryCount": 3,
              "maxActiveEntryCount": -1,
              "maxEntryCount": -1,
              "pending": 0
            }
          }
        },
        "id": "pool",
        "type": "de.intarsys.security.device.pool.device.PoolDeviceProvider",
        "enabled": true
      }
    },
    ...
  }
}

```

## 16.3.4.4 licensing

The main health detail entry point is "licensing", containing information about licenses and products.

## Status

The status is derived as follows:

"UP"

- when there is a valid license for the main product

"DOWN"

- otherwise

## Details

licenses	
A list of license objects	An enumeration of all installed licenses and their state
products	
A list of product objects	An enumeration of all available products and optional accounts

## Example

```
"licensing": {
  "status": "UP",
  "details": {
    "licenses": [{
      "status": "UP",
      "details": {
        "owner": "intarsys.de",
        "valid": true,
        "productId": "de.intarsys.cloudsuite.product.gears",
        "productVersion": "8",
        "validFrom": "11/13/2023",
        "validTo": "10/20/2123",
        ...
      }
    }],
    ...
  },
  ...
}
```

## 16.3.4.5 license

This indicator holds status information for a single installed license

## Status

The status is derived as follows:

"UP"

- when the issuer is correctly authenticated
- and the validity period is not expired

"DOWN"

- otherwise

## Details

Owner	
string	
productId	
string	

productVersion	
string	
Properties	
List of property objects	A list of "property" objects
validFrom	
string	The beginning of the validity period
validTo	
string	The end of the validity period
Valid	
boolean	Flag If this license is considered valid

#### Example

```
{
  "status": "UP",
  "details": {
    "properties": [{
      "name": "id",
      "value": "de.intarsys.cloudsuite.product.gears"
    }, {
      "name": "account_automation_cli",
      "value": "-1:day"
    }, {
      "name": "account_automation_batch",
      "value": "-1:day"
    }
  ],
  "owner": "intarsys.de",
  "valid": true,
  "productId": "de.intarsys.cloudsuite.product.gears",
  "productVersion": "8",
  "validFrom": "11/13/2023",
  "validTo": "10/20/2123"
}
```

#### 16.3.4.6 property

This indicator holds status information for a license property.

##### Details

name	
string	The name of the property
Value	
string	The value of the property

#### 16.3.4.7 product

This indicator holds status information for an application component that is separately licensed as a product.

##### Status

The status is derived as follows:

"UP"

- when there is a valid license for the product (or it is free)

"DOWN"

- otherwise

## Details

Id	
string	The id of the product
Version	
string	The version of the product
Label	
string	A descriptive label for the product
Main	
Boolean	Falg if this represents the main product (the application)
State	
string	State of the product (according to its licenses) One of na   valid   invalid
Accounts	
A list of account objects	
Licenses	
A list of license objects	See above.

## Example

```

"products": [{
  "status": "UP",
  "details": {
    "state": "valid",
    "id": "de.intarsys.cloudsuite.product.gears",
    "version": "8.11.0",
    "label": "Sign Live! cloud suite gears",
    "main": true,
    "accounts": [],
    "licenses": [{
      "properties": [{
        "name": "id",
        "value": "de.intarsys.cloudsuite.product.gears"
      },
      ...
    ],
    "owner": "intarsys.de",
    "valid": true,
    "productId": "de.intarsys.cloudsuite.product.gears",
    "productVersion": "8",
    "validFrom": "11/13/2023",
    "validTo": "10/20/2123"
  ]
},
...
]

```

## 16.3.4.8 account

This indicator holds status information for a product account. An account holds information about the current and maximum usage of a dedicated property of a product.

## Status

The status is derived as follows:

"UP"

- when there is a valid license for the product (or it is free)

"DOWN"

- otherwise

#### Details

Property	
string	The licensed property
Label	
string	A descriptive name for the account
Limit	
Int	The maximum usage limit
Balance	
Int	The current balance
nextReset	
Timestamp	If applicable the time of the next reset of the account to its limit
Product	
String	The associated product
Unit	
String	The unit of the balance/limit

#### Example

```
"accounts": [{
  "property": "de.intarsys.security.app.sign.account",
  "label": "Signaturerstellung",
  "limit": 10,
  "balance": 0,
  "nextReset": "2023-11-15T09:12:26.741+01:00[Europe/Berlin]",
  "product": "de.intarsys.security.device.gears",
  "unit": "minute"
}]
```

## 16.4 Micrometer, Prometheus and Grafana

This chapter explains the configurations and methods for monitoring the gears application using Spring Actuator, Micrometer, Prometheus and Grafana. These components work together to measure, collect, export, store and visualize application metrics.

### 16.4.1 Components

- Micrometer: Metrics collection facade.
- Spring Actuator: Framework for exposing operational information.
- Prometheus: Time-series database for storing metrics.
- Grafana: Visualization and dashboarding tool.

#### 16.4.1.1 Micrometer

Function: Vendor-neutral facade for metrics collection.

Capabilities: Collects metrics from the application and its components (e.g., JVM, HTTP requests).

Output: Formats metrics for Prometheus.

#### 16.4.1.2 Spring Actuator

Function: Exposes operational data about the running application.

Endpoint: `/manage/prometheus` for exposing metrics in Prometheus format.

Integration: Works with Micrometer to expose collected metrics.

### 16.4.1.3 Prometheus

Function: Scrapes and stores metrics as time-series data.

Endpoint: Scrapes metrics from Actuator's `/manage/prometheus`.

Query Language: PromQL for data aggregation and analysis.

Scraping Interval: Configured to scrape metrics at regular intervals (e.g., every 15 seconds).

### 16.4.1.4 Grafana

Function: Connects to Prometheus to visualize metrics.

Capabilities: Provides dashboards and alerting based on metric thresholds.

Features: Supports templating for dynamic dashboards.

## 16.4.2 Metric Flow Example

Metric Generation: Spring Boot application generates metrics (e.g., HTTP request count, response times).

Metric Collection: Micrometer captures and maintains these metrics in memory.

Metric Exposure: Spring Actuator exposes metrics via `/manage/prometheus` in Prometheus format.

Metric Scraping: Prometheus scrapes the endpoint and stores metrics in its time-series database.

Metric Visualization: Grafana queries Prometheus and displays metrics in customizable dashboards.

## 16.4.3 Metrics

In the gears application, metrics are used for monitoring and understanding system performance and behavior. Micrometer is used to expose standard metrics, providing insights into various aspects such as JVM performance, HTTP request handling and system health. Additionally, gears collects and exposes custom metrics tailored to specific operational needs. These custom metrics offer deeper visibility into application-specific events and operations, enabling more granular monitoring and analysis.

The custom metrics in the gears application are exclusively timers that measure how long certain processes take. The following sections will detail these metrics and utilization.

### 16.4.3.1 Prometheus Metrics Naming Convention

Prometheus metrics follow a specific naming convention to ensure clarity and consistency. The typical structure includes:

- Metric Name: Describes the metric being collected.
- Labels (Tags): Provide additional context about the metric.

#### 16.4.3.1.1 Metric Name

The metric follows these guidelines:

- Lowercase: Metric names are typically lowercase.
- Underscores: Use underscores to separate words.
- Suffixes: `_seconds` for timers
  - `_sum`: Represents the total time recorded of one metric.

- `_count`: Represents the number of occurrences.
- `_max`: Represents the maximum time recorded in an interval (see chapter 16.4.4.2.2 “distributionStatisticExpiry”).

### 16.4.3.1.2 Labels (Tags)

Labels add dimensions to metrics, allowing for more detailed analysis. Common labels include:

- `instance`: The instance from which the metric is collected.
- `job`: The job or service generating the metric.
- `operation`: The specific operation being measured.
- `code`: Status code or result of the operation.
- `configuration`: Configuration context for the metric.

### 16.4.3.2 Metric ID Construction

Each metric is identified by a `metricId`, which is constructed from the metric name and associated tags. This structure allows for precise identification and querying of metrics.

Below are examples of custom metrics in the gears application, illustrating the naming convention and tag combinations:

```
{__name__="service_viewer_seconds_max", outcome="ok",
configuration="demoSignature", instance="integration.intarsys.de:80", job="cloudsuite-
core-1", operation="flow", tenant="tenant"}
```

```
{__name__="service_signer_seconds_max", outcome="ok", configuration="demoBridge",
instance="integration.intarsys.de:80", job="cloudsuite-core-3", operation="create",
tenant="tenant"}
```

```
{__name__="device_demo_default_seconds_max", outcome="ok",
instance="integration.intarsys.de:80", job="cloudsuite-core-1", operation="sign",
tenant="tenant"}
```

In the table below the structure of the metric names and tags is further visualized:

Availability	general	only for services	only for devices
Metric name part 1		service	device
Metric name part 2		service name <ul style="list-style-type: none"> <li>• signer</li> <li>• viewer</li> </ul>	device provider <ul style="list-style-type: none"> <li>• demo</li> <li>• signme</li> <li>• ais</li> <li>• ...</li> </ul>
Metric name part 3			device name <ul style="list-style-type: none"> <li>• default</li> <li>• tsa</li> <li>• ...</li> </ul>
Metric name part 4	seconds		
Metric name part 5	suffix <ul style="list-style-type: none"> <li>• count</li> <li>• sum</li> <li>• max</li> </ul>		
Labels	operation	<ul style="list-style-type: none"> <li>• create</li> <li>• flow</li> </ul>	<ul style="list-style-type: none"> <li>• sign</li> <li>• authenticate</li> </ul>

			step • ... (defined per device)
	outcome • ok • fail • cancel		
		Configuration • aisStatic • signMeQualified • demoPlain • ...	
	AddPropertyStatement • tenant • ...		
	instance • ...		
	job • ...		

#### 16.4.3.2.1 Metric explanation

Some of the metric labels require further context:

- operation:
  - “flow” is the entire process of e.g. a sign process. This includes also time spent during asynchronous sign processes.
  - “create” only measures the time of a http request. In synchronous cases this e.g. includes the time spent signing and processing the document. In asynchronous cases only the time until a processing stage is returned is measured.
  - “sign” measures the time spent by the signature device.
- code:
  - “ok”: a process was successful
  - “fail”: an error occurred or a process was cancelled
  - “cancel”: a process was cancelled
- AddPropertyStatement:
  - This is a configurable property (see chapter 16.4.4.2.2)

### 16.4.4 Configuration

#### 16.4.4.1 Overview

Micrometer is enabled by default and values like JvmMemoryMetrics and ProcessorMetrics are being collected (see micrometer documentation). When the "demo" profile is active, custom metrics (e.g. a timer like service\_signer\_seconds) are also captured. If the "demo" profile is inactive, additional configuration is required to capture these custom metrics. An example configuration called “spring-gears-micrometer-monitoring.xml” can be found in the gears application bundle under /examples/monitoring/observations spring-beans.

Prometheus and Grafana are external applications that can be configured and run in various environments, such as Docker containers. The setup and configuration of these tools are flexible and can be tailored by the integrator to fit their specific deployment needs.



## 16.4.4.2 Micrometer Configuration

### 16.4.4.2.1 Observation metrics collection

The collection of custom metrics makes use of the observation system (see chapter 15).

This following example configuration enables the application to capture and process custom metrics from specified sources, apply filters, and use Micrometer for metrics collection. This configuration is also included in the demo profile.

#### Spring XML fragment

```
<bean class="de.intarsys.tools.observation.logback.LogbackObserver">
  <property name="source">
    <list>
      <value>device</value>
      <value>service</value>
    </list>
  </property>
  <property name="view">
    <list>
      <bean class="de.intarsys.tools.observation.impl.IncludeStatement">
        <property name="pattern" value=".*"/>
      </bean>
      <bean class="de.intarsys.tools.observation.impl.AddPropertyStatement">
        <property name="name" value="tenant"/>
        <property name="value" value="{principal.tenant.name}"/>
      </bean>
    </list>
  </property>
  <property name="filter">
    <list>
      <bean class="de.intarsys.tools.observation.impl.AcceptPredicate">
        <property name="expr" value="{observation.code}"/>
        <property name="pattern"
value="(sign|authenticate|create|flow)\. .*"/>
      </bean>
    </list>
  </property>
  <property name="appender">
    <bean class="de.intarsys.tools.observation.micrometer.MicrometerAppender">
      <property name="maxTimerAge" value="30m" />
      <property name="distributionStatisticExpiry" value="10s" />
      <property name="distributionStatisticBufferLength" value="2" />
      <property name="cleanupThreshold" value="1000" />
    </bean>
  </property>
</bean>
```

### 16.4.4.2.2 Configuration Details

#### LogbackObserver Bean

- The LogbackObserver bean is configured to observe specific sources and apply various processing steps to the captured metrics.

#### Source Property

- Purpose: Specifies the sources to be observed.
- Values: device, service.

#### View Property

- Purpose: Defines how the observed data is processed (see also chapter 15.6).
- IncludeStatement: Includes all observations matching the pattern .\*.
- AddPropertyStatement: Adds a property named tenant with a value derived from the principal's tenant name.

#### Filter Property

- Purpose: Applies filters to the observed data.
- AcceptPredicate: Accepts observations where the observation.code matches the pattern (sign|create|flow)\..\*.

#### Appender Property

- Purpose: Configures the Micrometer appender for metrics collection.
- MicrometerAppender:
  - maxTimerAge
    - Purpose: Defines the maximum time a timer can exist in the internal map before being considered expired and removed during cleanup.
    - Default Value: 30 minutes
    - Usage: This property prevents memory leaks by ensuring that timers that never received a finish event (due to errors or unexpected application behaviour) are eventually cleaned up.
    - Impact: A smaller value reduces memory usage but might remove legitimate timers that have long operations. A larger value allows for longer-running operations but might increase memory usage.
  - cleanupThreshold
    - Default Value: 1000
    - Usage: When the number of active timers exceeds this threshold, the method is called to remove expired timers.
    - Impact: A lower threshold causes more frequent cleanups, potentially reducing memory usage but adding more processing overhead. A higher threshold reduces cleanup frequency but might allow more expired timers to remain in memory.
  - distributionStatisticExpiry
    - Default Value: 10 seconds
    - Info from micrometer docs: Statistics emanating from a timer like max, percentiles, and histogram counts decay over time to give greater weight to recent samples. Samples are accumulated to such statistics in ring buffers which rotate after this expiry, with a buffer length of distributionStatisticBufferLength, hence complete expiry happens after this expiry \* buffer length.
  - distributionStatisticBufferLength
    - Default Value: 2
    - Usage: see distributionStatisticExpiry.

### 16.4.4.2.3 Metrics endpoint

As explained in chapter 16.3.2 the endpoint: /manage/prometheus is configured for exposing metrics in Prometheus format.

### 16.4.4.3 Prometheus and Grafana Configuration in Docker Environment

Prometheus and Grafana are both external applications that can be deployed and configured in various ways. We provide configuration examples for docker containers in the gears application bundle under /examples/monitoring:

- docker-compose.yml:
  - Purpose: Defines the services, networks, and volumes for running Prometheus and Grafana in Docker containers.
  - Contents: Specifies the Prometheus and Grafana services, including their images, ports, and volume mappings.

- `gears core dashboard.json`:
  - Purpose: Provides a pre-configured Grafana dashboard for visualizing metrics from the gears application.
  - Contents: JSON configuration for Grafana dashboards, including panels, queries, and layout settings.
- `grafana.yml`:
  - Purpose: Configures Grafana settings, such as data sources and dashboards.
  - Contents: YAML configuration for setting up Grafana, including the connection to Prometheus as a data source.
- `prometheus.yml`:
  - Purpose: Configures Prometheus to scrape metrics from the gears application.
  - Contents: YAML configuration for Prometheus, including scrape intervals, targets, and job definitions.
- `prometheus-integration.yml`:
  - Contents: YAML configuration for extended Prometheus settings, with additional scrape targets.

An additional Grafana dashboard that provides useful information from the JVM can be imported via the Grafana UI: Home -> Dashboards -> Import dashboard -> dashboard ID: 4701

# 17. NLS support

National language support is a cross-cutting concern that touches many aspects of gears, so we have a dedicated chapter that summarizes the aspects.

## 17.1 Language selection

The current language for an interaction is derived as follows (in ascending order)

- The language of the server VM
- The selected browser language
- The language associated with the flow via the option "lang"
- A user defined language that is requested via an actual query parameter "lang" for the request that is currently processed.

## 17.2 Language option

You can select a preferred language for a flow via the "lang" option. For more details see chapter "Language".

## 17.3 Language static expansion

gears supports Spring string expansion for NLS resolving.

This expansion, using the "\${}" notation in a Spring configuration file is done at system startup! If you need more dynamic resolving, you should use the NLS string expansion defined in a chapter below.

## 17.4 Language dynamic expansion

In your service calls (e.g., when defining widgets) or configuration files (take care of the workaround for spring specific files, see chapter 20) you can use the "nlsmg" namespace to access locale specific resources using string expansion.

This expansion is done using the language context for the request.

The complete syntax is

```
nlsmg:<package>.<resource>#<code>
```

where "package" is a path name according to the Java package naming convention (e.g. de.intarsys.gars.core.demo.ui), "resource" is the name of the resource file without extension (e.g. messages) and "code" is the key in the properties file (e.g. MyButton.label).

Example

```
${nlsmsg.de.intarsys.gears.core.demo.ui.messages#MyButton.label}
```

This will look up the resource "messages\_<lang>.properties" (respective "message.properties" if the "lang" code is not available) and resolve the property "MyButton.label".

If the resource or the property cannot be found, the message code itself is returned.

To force the language selection to happen during runtime and access NLS messages via **Spring config files** use `?{}`

```
?{nlsmsg.de.intarsys.gears.core.demo.ui.messages#MyButton.label}
```

Remember: when accessing NLS files from a service request you must use `${}`.

## 17.5 Additional resource path

Normally resources are looked up in the JAR files of the runtime. To ease management of NLS files, for resolving an additional directory is added to the lookup.

The directory is denoted by "cloudsuite.i18n.dir" which defaults to "`${cloudsuite.config.shared}/i18n`".

In this directory the resources are looked up according to Java package rules. The resources are **not** expected in JAR files but are looked up in plain directory/file format.

You can set the directory by adding the property to your configuration.

```
cloudsuite.i18n.dir=/opt/test/messages
```

You can use message definitions in this directory to **overwrite** existing message. If a bundle is found both in the additional resource path and builtin, the additional bundle takes precedence. As both resource files are merged, you can create a partial redefinition, too.

Example

To create a new resource for the message from the example above

```
${nlsmsg.de.intarsys.gears.core.demo.ui.messages#MyButton.label}
```

you create a directory hierarchy within the NLS directory

```
de
  intarsys
    gears
      core
        demo
          ui
```

In the last directory you can now create the resource file, e.g. "messages\_de.properties" for "de" or "messages.properties" for the default. Similarly, the icon descriptors can be configured in "icons.properties".

Within the file you define the translation for the key

```
MyButton.label=Tolle Beschriftung
```

## 17.6 String expansion within NLS files

As stated before, a detailed explanation for string expansion can be found in chapter 20. However if you wish to do string expansion within NLS (messages.properties) files the following should be considered.

### 17.6.1 Eliminate [] brackets

NLS files have their own logic for variable replacement. To escape the mechanism, we must use the following syntax: `'${variable}'`. Otherwise `{}` would be surrounded by `[]`.

Example:

With `foo.bar` containing **buttonDescription**

```
example MyButton.label=${foo.bar}
```

evaluates to

```
→ example [buttonDescription]
```

To get rid of the brackets `[]`, we need to adjust it as follows:

```
example MyButton.label='${foo.bar}'
```

evaluates the desired result

```
→ example buttonDescription
```

### 17.6.2 Spring string expansion in NLS files

When it comes to string expansion within NLS files, generally the same rules apply as described in chapter 20.

During application startup Spring does string expansion in Spring config files (gears.properties, .xml). The application will finish Spring template processing by replacing all occurrences of `"?{"` with `"${"`.

NLS files (messages.properties) are not Spring config files. Therefore the `"?{"` to `"${"` replacement does not occur. If, however, a variable in a Spring config file references a variable in a NLS file via `"${"`, Spring will do string expansion for that NLS variable during application startup. See the example in chapter 20.5.6.

# 18. Reference

## 18.1 Common data models

### 18.1.1 Configuration

#### 18.1.1.1 Overview

The configuration allows the definition of a flow execution context as a template.

The semantics of a configuration may depend on the concrete flow where it is used.

A configuration is passed as an argument in a flow create call. For greatest flexibility, it can be passed in the API calls

- by reference
- literal
- a list thereof

For details and examples see the OpenApi documentation.

#### 18.1.1.2 By reference

If a configuration is pre-defined, e.g., by providing a definition in a Spring XML file, it can be referenced simply by a string.

There may be other providers (from a database, ...) that allow resolving of configuration ids.

#### 18.1.1.3 Literal

The configuration object itself has the following properties:

id	
string	An optional unique id.  This is required when the configuration is intended to be looked up by reference.
label	
string	An optional human readable name
description	
string	An optional human readable description
variables	
object	Define key/value pairs that can be used for variable expansion throughout the flow.
arguments	
list of key/value pairs	Define key/value pairs that are applied in order of definition as defaults on the "args" structure that was send with the flow create request.
widgets	

list of WdigetSpec	Define a list of widgets that are available to an interactive UI (e.g., the viewer)
actions	
list of ActionSpec	Define a list of actions that are available to an interactive UI (e.g., the viewer).  The uniqueness of action id's is not enforced in this list. The last action definition with a specific id is the one to be registered in the action registry.
plugins	
list of PluginSpec	Define a list of plugins that are injected into the flow (either on client or server) as a means to customize or extend the default behavior.
settings	
object	

### 18.1.1.3.1 Spring XML examples

Spring configuration example, variables

#### Spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="myConfig"/>
  <property name="variables">
    <map>
      <entry key="foo.a" value="v-foo.a"/>
      <entry key="foo.b.a" value="v-foo.b.a"/>
    </map>
  </property>
</bean>
```

Spring configuration example, arguments

**Note** the property name “definitions” – this is for internal spring mapping reasons. Setting “arguments” is possible too but the syntax is (even more) awkward, so we skip this.

#### Spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="myConfig"/>
  <property name="definitions">
    <map>
      <entry key="def.foo.a" value="v-foo.a"/>
      <entry key="def.foo.b.a" value="v-foo.b.a"/>
    </map>
  </property>
</bean>
```

Spring configuration example, widgets



## Spring XML fragment

```

<bean id="flowViewerConfigurationPlain"
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="plain"/>
  <property name="widgets">
    <list>
      <w:widget parent="de.intarsys.widget.toolbar.additions" id="widget1"
label="Widget 1" icon="edit">
        <w:on event="select" do="Alert">
          <entry key="message" value="Hello 1"/>
        </w:on>
      </w:widget>
      <w:widget parent="de.intarsys.widget.toolbar.additions" id="widget2"
label="Widget 2" icon="edit">
        <w:on event="select" do="Alert">
          <entry key="message" value="Hello 2"/>
        </w:on>
      </w:widget>
    </list>
  </property>
</bean>

```

## Spring configuration example, actions

## Spring XML fragment

```

<bean
class="de.intarsys.cloudsuite.gears.core.service.viewer.impl.FlowViewerConfiguration">
  <property name="id" value="myConfig"/>
  <property name="actions">
    <list>
      <w:action factory="Alert" id="action1">
        <entry key="id" value="action1"/>
        <entry key="message" value="Hello 1"/>
      </w:action>
      <w:action factory="Alert" id="action2">
        <entry key="id" value="action2"/>
        <entry key="message" value="Hello 2"/>
      </w:action>
    </list>
  </property>
</bean>

```

## Spring configuration example, plugins

## Spring XML fragment

```

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

```

## 18.1.1.3.2 JSON examples

## Request parameter configuration example, variables

## JSON fragment

```
{
  "id": "myConfig",
  "label": "Meine Konfiguration",
  "variables": {
    "foo": "bar",
    "a": {
      "b": {
        "c": "x"
      }
    }
  }
}
```

request parameter configuration example, arguments.

**Note** Here too for reasons of simplicity “definitions” are used allowing for a more simple syntax.

## JSON fragment

```
{
  "id": "myConfig",
  "label": "Meine Konfiguration",
  "definitions": {
    "foo": "bar",
    "a.wombat": "x",
    "a.gnat": "y"
  }
}
```

request parameter configuration example, arguments. Here we present the “arguments” variation that is slightly less readable.

## JSON fragment

```
{
  "id": "myConfig",
  "label": "Meine Konfiguration",
  "arguments": [
    {
      "path": "foo",
      "value": "bar"
    },
    {
      "path": "a.wombat",
      "value": "x"
    },
    {
      "path": "a.gnat",
      "value": "y"
    }
  ]
}
```

request parameter configuration example, widgets

## JSON fragment

```
{
  "id": "myConfig",
  "label": "Meine Konfiguration",
  "widgets": [{
    "label": "Test 1",
    "parent": "de.intarsys.widget.toolbar.additions",
    "children": [{
      "label": "Subtest 1",
      "callbacks": {
        "select": {
          "factory": "Alert",
          "args": {
            "message": "subtest 1"
          }
        }
      }
    }, {
      "label": "Subtest 1",
      "callbacks": {
        "select": {
          "factory": "Alert",
          "args": {
            "message": "subtest 1"
          }
        }
      }
    }
  ]
}, {
  "label": "Test 2",
  "callbacks": {
    "select": {
      "factory": "Alert",
      "args": {
        "message": "test 2"
      }
    }
  }
}
]
```

request parameter configuration example, actions

## JSON fragment

```
{
  "id": "myConfig",
  "label": "Meine Konfiguration",
  "actions": [{
    "id": "action1",
    "factory": "Alert",
    "args": {
      "message": "action 1"
    }
  }, {
    "id": "action2",
    "factory": "Alert",
    "args": {
      "message": "action 2"
    }
  }
]
}
```

request parameter configuration example, plugins

## JSON fragment

```
{
  "id": "myConfig",
  "label": "Meine Konfiguration",
  "plugins": [{
    "factory": "de.intarsys.plugin.BlackeningEditor",
    "args": {
      "decorate": {
        "factory": "Ok",
        "action": {
          "factory": "BlackeningEditorSave",
          "args": {
            "saveAlways": false
          }
        }
      }
    }
  }
],
}
```

## 18.1.1.4 Array

A configuration may be passed as an array of configurations, too. This is relevant for service calls only, as you cannot define a configuration as a list in Spring XML currently.

The **effective configuration** is created by merging all configurations in the array. Configurations later in the array have higher precedence (they may overwrite definition from earlier configurations).

Example: Combine two configurations by reference

## JSON fragment

```
[
  "configA",
  "configB"
]
```

Example: Combine a literal configuration with a reference

## JSON fragment

```
[
  {
    "plugins": [
      {
        "factory": "de.intarsys.plugin.BlackeningEditor",
        "args": {}
      }
    ],
    "widgets": [
      {
        "label": "Start Signature",
        "parent": "de.intarsys.widget.toolbar.additions",
        "callbacks": {
          "select": {
            "ref": "newAction0"
          }
        }
      }
    ]
  },
  "configB"
]
```

Example: Combine two literal configurations

## JSON fragment

```
[
  {
    "plugins": [
      {
        "factory": "de.intarsys.plugin.BlackeningEditor",
        "args": {}
      }
    ],
    "widgets": [
      {
        "label": "A",
        "parent": "de.intarsys.widget.toolbar.additions",
        "callbacks": {
          "select": {
            "ref": "newActionA"
          }
        }
      }
    ]
  },
  {
    "widgets": [
      {
        "label": "B",
        "parent": "de.intarsys.widget.toolbar.additions",
        "callbacks": {
          "select": {
            "ref": "newActionB"
          }
        }
      }
    ]
  }
]
```

### 18.1.2 PluginSpec

A **PluginSpec** defines an extension to the standard behavior of a gears flow. Using plugins allows to constrain the basic application size and allows to support seamless future extensions.

#### properties

factory	
string	The type of plugin.  For a reference of available plugins see "Plugin reference"
args.*	
object	Arguments for the plugin.  The argument set depends on the plugin type.

#### 18.1.2.1 Spring XML examples

Plugins are defined using plain beans

## Spring XML fragment

```
<bean class="de.intarsys.cloudsuite.gears.plugin.PluginSpec">
  <property name="factory" value="de.intarsys.plugin.ComponentApi"/>
  <property name="args">
    <map>
      <entry key="protocol" value="windows"/>
    </map>
  </property>
</bean>
```

## 18.1.2.2 JSON examples

## Example

## JSON fragment

```
{
  "factory": "de.intarsys.plugin.ComponentApi",
  "args": {
    "protocol": "windows"
  }
}
```

## 18.1.3 ActionSpec

An **ActionSpec** defines a process that can be started by some trigger event. Using actions, you can for example customize the behavior of your viewer component or control gears behavior via calls from a containing web application (see ComponentApi, [3]).

All actions defined in the **actions** property of a **Configuration** are stored in a registry for later lookup via an action reference.

An action can also be used literally, e.g., within a widget definition. There is a special DSL supporting action definitions in Spring XML.

The available action types and the resulting behavior is documented in chapter "Action reference".

Wherever an action is supported, you can use either

- a literal action
- a reference to an action
- a list of actions

There are multiple flavors of literal definition style – which one you use depends on your preference and the complexity of the action.

## 18.1.3.1 "instance spec" style

factory	
string	The type of action
args.id	
string	An optional unique id for the action
	This is required for later lookup via a reference
args.label	
string	An optional label for the action
args.description	
string	An optional description for the action
args.icon	
string	An optional icon for the action
args.*	
object	Any additional argument that is valid for the respective argument type

## 18.1.3.2 "typed" style

type	
string	The type of action
id	
string	An optional unique id for the action
	This is required for later lookup via a reference
label	
string	An optional label for the action
description	
string	An optional description for the action
icon	
string	An optional icon for the action
*	
object	Any additional argument that is valid for the respective argument type

## 18.1.3.3 "ref" style

In addition, an action can be defined "by reference", for example from within a widget, referencing a predefined action from the action registry.

ref	
string	The id of an action from the action registry

## 18.1.3.4 Action block definition

An "action block" implements the try-catch-finally construct well known from programming languages. While internally it is implemented in the "Block" action (see reference), there are syntactical shortcuts to make a definition more readable.

## 18.1.3.5 Spring DSL

The chapter shows the specific DSL that is supported in Spring XML definition files.

The attributes and elements have the same behavior and meaning as defined in the reference above.

The DSL supports the "instance spec" and "ref" style.

## Elements

action	
	An action definition to be performed. You can add multiple action elements. All action elements must precede "catch" and "finally" elements.
catch	
	Contains an action that gets performed in case of an error. Only a single "catch" element is allowed. If you need to perform multiple actions, you can simply enclose the action elements as children of "catch".
finally	
	Contains an action that gets performed after processing all other actions. Only a single "finally" element is allowed. If you need to perform multiple actions, you can simply enclose the action elements as children of "finally".

*action*

## Attributes

factory	
string	The factory for an action. See above
ref	
string	The name of a registered action. See above

## Elements

entry	
element	action arguments

*entry*

## Attributes

key	
string required	The name of an action argument
value	
string	The value of an action argument. This property supports string expansion.

*catch*

## Attributes

-
---

## Elements

action	
	An action to be performed in case of an error

*finally*

## Attributes

-
---

## Elements

action	
	An action to be performed after execution.

## 18.1.3.6 Spring XML examples

For Spring XML definitions, you should use the widget-DSL. A "typed" style is not supported for this DSL.

Example instance-spec



## Spring XML fragment

```
<w:action factory="Alert">
  <entry key="id" value="action1"/>
  <entry key="message" value="Hello 1"/>
</w:action>
```

## Example ref in Spring XML

## Spring XML fragment

```
<w:action ref="myAction">
</w:action>
```

## Example block in Spring XML

## Spring XML fragment

```
<w:action factory="Confirm">
</w:action>
<w:catch factory="Alert">
</w:catch>
<w:finally factory="Alert">
</w:finally>
```

## Example block in Spring XML, you can add multiple "action statements" to a block.

## Spring XML fragment

```
<w:action factory="Confirm">
</w:action>
<w:action factory="Confirm">
</w:action>
<w:catch>
  <w:action factory="Alert">
  </w:action>
  <w:action factory="Alert">
  </w:action>
</w:catch>
<w:finally>
  <w:action factory="Alert">
  </w:action>
  <w:action factory="Alert">
  </w:action>
</w:finally>
```

## 18.1.3.7 JSON examples

## Example instance spec based definition

## JSON fragment

```
{
  "factory": "Alert",
  "args": {
    "id": "myid",
    "message": "Hello"
  }
}
```

## Example type based definition

## JSON fragment

```
{
  "type": "Alert",
  "id": "myid",
  "message": "Hello"
}
```

## Example ref-based action access

## JSON fragment

```
{
  "ref": "myAction"
}
```

## Example block action with error handling

## JSON fragment

```
{
  "try": [{
    "factory": "Confirm",
    "args": {
      "message": "Hi 1"
    }
  }, {
    "factory": "Confirm",
    "args": {
      "message": "Hi 2"
    }
  }
],
  "catch": {
    "factory": "Alert",
    "args": {
      "message": "catch"
    }
  },
  "finally": {
    "factory": "Alert",
    "args": {
      "message": "finally"
    }
  }
}
```

## 18.1.4 WidgetSpec

### 18.1.4.1 Properties

This structure defines appearance and behavior of a user interface element.

The semantics of a widget depend on its type. The description of supported widgets is in chapter "Widget reference".

## properties

id	
string	An optional locally unique id for the widget
	You can use this id when you navigate to a dedicated widget within it parent.
register	
string	An optional globally unique id for the widget.

	You can use this id when you need to reference a widget directly
parent	
string	The name of a registered parent widget (see "register" above)
type	
string	An optional type for the widget. If no type is given, it is derived from the widget parent.  For detailed information on the supported types see "Widget reference"
label	
string	An optional label to be used for the widget.
tip	
string	An optional short description to be used for the widget.
description	
string	An optional description to be used for the widget.
icon	
string	An optional icon to be used for the widget.
style	
string	An optional style definition to be used for the widget.
callbacks	
object	An object containing key / action bindings to define actions that should be triggered for the respective event from the widget.  The key represents the widget trigger name, the value is an action or action reference.  What triggers are available depends on the widget type. Normally a widget supports the <b>select</b> trigger to initiate its action.
properties	
object	An object containing key / value bindings.  What properties are supported depends on the widget type.

#### 18.1.4.2 Spring DSL

The chapter shows the specific DSL that is supported in Spring XML definition files.

The attributes and elements have the same behavior and meaning as defined in the reference above.

#### *widget*

##### Attributes

id	
string	see above
register	
string	see above
parent	
string	see above
type	
string	see above
icon	
string	see above
label	
string	see above
tip	
string	see above

description	
string	see above
style	
string	see above

## Elements

property	
	see <b>properties</b> above
on	
	see <b>callbacks</b> above

*property*

## Attributes

name	
string required	see above
value	
string	see above

*on*

## Attributes

event	
string	see above
action/do/ref	
string	<p>This starts a short form for an action where the "factory" (action or do) or "ref" is given as an attribute directly.</p> <p>The embedded w:action element can be omitted then.</p>

## Elements

action	
action	<p>A complete embedded action definition.</p> <p>See above</p>

## 18.1.4.3 Spring XML examples

For Spring XML definitions, you should use the widget-DSL.

button with an action, long form

## Spring XML fragment

```
<w:widget parent="de.intarsys.widget.toolbar.additions" id="widget1">
  <w:on event="select">
    <w:action factory="Alert">
      <entry key="message" value="Hello!" />
    </w:action>
  </w:on>
  <w:property name="dummy" value="dummy value" />
</w:widget>
```

button with an action, short form

## Spring XML fragment

```
<w:widget parent="de.intarsys.widget.toolbar.additions" id="widget1">
  <w:on event="select" do="Alert">
    <entry key="message" value="Hello!"/>
  </w:on>
  <w:property name="dummy" value="dummy value"/>
</w:widget>
```

widget definition with multiple actions and error handling

## Spring XML fragment

```
<w:widget parent="de.intarsys.widget.toolbar.additions" id="widget1">
  <w:on event="select">
    <w:action factory="Confirm">
      <entry key="message" value="Everything OK?"/>
    </w:action>
    <w:action factory="Alert">
      <entry key="message" value="Fine!"/>
    </w:action>
    <w:catch factory="Alert">
      <entry key="message" value="Why not?"/>
    </w:catch>
    <w:finally factory="Alert">
      <entry key="message" value="All over now"/>
    </w:catch>
  </w:on>
  <w:property name="dummy" value="dummy value"/>
</w:widget>
```

## 18.1.4.4 JSON examples

widget example with instance-spec action

## JSON fragment

```
{
  "label": "Achtung!",
  "callbacks":
  {
    "select":
    {
      "factory": "Alert",
      "args":
      {
        "message": "Hi"
      }
    }
  },
  "properties":
  {
    "dummy": "dummy value"
  }
}
```

widget example with typed action

## JSON fragment

```
{
  "label": "Achtung!",
  "callbacks": {
    "select": {
      {
        "type": "Alert",
        "message": "Hi"
      }
    },
    "properties": {
      {
        "dummy": "dummy value"
      }
    }
  }
}
```

widget example with action sequence and error handling

## JSON fragment

```
{
  "label": "Block test",
  "callbacks": {
    "select": {
      "try": [{
        {
          "factory": "Confirm",
          "args": {
            "message": "Hi 1"
          }
        }, {
          "factory": "Confirm",
          "args": {
            "message": "Hi 2"
          }
        }
      ],
      "catch": {
        "factory": "Alert",
        "args": {
          "message": "catch"
        }
      },
      "finally": {
        "factory": "Alert",
        "args": {
          "message": "finally"
        }
      }
    },
    "properties": {
      {
        "dummy": "dummy value"
      }
    }
  }
}
```

## 18.2 Principal related data models

### 18.2.1 GenericClaim

[\*de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim\*](#)

A **de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim** is a simple implementation that can be used directly to define literal claims about principal instances via Spring.

## Properties

key	
string required	The property name of the claim
value	
string required	The property value of the claim

## Spring configuration example

## Spring XML fragment

```
<bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
  <property name="name" value="foo" />
  <property name="claims">
    <list>
      <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
        <property name="key" value="claim1" />
        <property name="value" value="value1" />
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
        <property name="key" value="claim2" />
        <property name="value" value="value2" />
      </bean>
    </list>
  </property>
</bean>
```

## 18.2.2 GenericPrincipal

*de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal*

A **de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal** is a simple implementation that can be used directly to define literal principal instances via Spring.

## Properties

name	
string required	The name of the principal.
claims	
array of GenericClaim	Optional list of claims about the principal.

## Spring configuration example

## Spring XML fragment

```
<bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
  <property name="name" value="foo" />
  <property name="claims">
    <list>
      <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
        <property name="key" value="claim1" />
        <property name="value" value="value1" />
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
        <property name="key" value="claim2" />
        <property name="value" value="value2" />
      </bean>
    </list>
  </property>
</bean>
```

## 18.2.3 GenericUser

*de.intarsys.cloudsuite.gears.model.entity.user.GenericUser*

A **de.intarsys.cloudsuite.gears.model.entity.user.GenericUser** is a simple implementation that can be used directly to define literal entities via Spring.

## Properties

name	
string required	The name of the user.
password	
string optional	The cleartext password for the user. Using cleartext passwords is not recommended for production use, neither with in-memory, nor with database backed user representations.
passwordHash	
base64 string optional	The hashed password.
salt	
base64 string optional	The salt for the password hashing

## Spring configuration example

## Spring XML fragment

```
<bean class="de.intarsys.cloudsuite.gears.model.entity.user.GenericUser">
  <property name="name" value="foo" />
  <property name="password" value="bar" />
</bean>
```



## 18.2.4 JdbcPrincipalDao

*de.intarsys.cloudsuite.gears.model.entity.user.JdbcUserDao*

A JDBC based implementation for user entity lookup.

### Properties

dataSource	
Data Source required	The data source used to create a database connection
lookupSql	
String required	The SQL string used to select a single entity. The SQL must return the columns <ul style="list-style-type: none"> <li>• name</li> <li>• key</li> <li>• value</li> </ul>

### Spring configuration example

#### Spring XML fragment

```
<bean id="modelPrincipalDao"
class="de.intarsys.cloudsuite.gears.model.entity.principal.JdbcPrincipalDao">
  <property name="dataSource" ref="dataSource" />
  <property name="lookupSql" value="select Principal.name, Claim.key, Claim.value from
Principal inner join Claim on Principal.name = Claim.name where Principal.name=?" />
</bean>
```

## 18.2.5 JdbcUserDao

*de.intarsys.cloudsuite.gears.model.entity.user.JdbcUserDao*

A JDBC based implementation for user entity lookup.

### Properties

dataSource	
Data Source required	The data source used to create a database connection
lookupSql	
String required	The SQL string used to select a single entity. The SQL must return the columns <ul style="list-style-type: none"> <li>• name</li> <li>• salt (optional)</li> <li>• password (optional)</li> <li>• passwordHash (optional)</li> </ul>

## Spring configuration example

## Spring XML fragment

```
<bean id="modelUserDao"
class="de.intarsys.cloudsuite.gears.model.entity.user.JdbcUserDao">
  <property name="dataSource" ref="dataSource" />
  <property name="lookupSql" value="select name, salt, passwordHash, password from
BasicAuth where name=?" />
</bean>
```

## 18.2.6 PojoPrincipalDao

*de.intarsys.cloudsuite.gears.model.entity.principal.PojoPrincipalDao*

A simple implementation that can be used directly to define an in memory lookup table.

## Properties

items	
collection of model objects required	The list of model objects for the in-memory table.

## Spring configuration example

## Spring XML fragment

```

<bean id="modelPrincipalDao"
class="de.intarsys.cloudsuite.gears.model.persistence.pojo.PojoEntityDao">
  <property name="items">
    <list>
      <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
        <property name="name" value="foo" />
        <property name="claims">
          <list>
            <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
              <property name="key" value="claim1" />
              <property name="value" value="value1" />
            </bean>
            <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
              <property name="key" value="claim2" />
              <property name="value" value="value2" />
            </bean>
          </list>
        </property>
      </bean>
      <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
        <property name="name" value="E=support@intarsys.de,CN=gears demo client
ssl,O=intarsys GmbH" />
        <property name="claims">
          <list>
            <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
              <property name="key" value="claimX" />
              <property name="value" value="valueX" />
            </bean>
            <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericClaim">
              <property name="key" value="claimY" />
              <property name="value" value="valueY" />
            </bean>
          </list>
        </property>
      </bean>
    </list>
  </property>
</bean>

```

## 18.2.7 PojoUserDao

*de.intarsys.cloudsuite.gears.model.entity.user.PojoUserDao*

A simple implementation that can be used directly to define an in-memory lookup table.

## Properties

items	
collection of model objects required	The list of model objects for the in-memory table.

## Spring configuration example

## Spring XML fragment

```
<bean id="modelUserDao"
class="de.intarsys.cloudsuite.gears.model.entity.user.PojoUserDao">
  <property name="items">
    <list>
      <bean class="de.intarsys.cloudsuite.gears.model.entity.user.GenericUser">
        <property name="name" value="foo" />
        <property name="password" value="bar" />
      </bean>
    </list>
  </property>
</bean>
```

## 18.2.8 ExplicitPrincipalProvider

*de.intarsys.cloudsuite.gears.model.entity.principal.ExplicitPrincipalProvider*

The context principal is derived from the service arguments.

## Properties

role	
String required	One of the well-known role names (or a custom one).
principalDao	
DAO required	A DAO that can look up a principal for the "principal" value provided in the service call options.

## Spring configuration example

## Spring XML fragment

```
<bean id="principalProviderUser"
class="de.intarsys.cloudsuite.gears.model.entity.principal.ExplicitPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:User" />
  <property name="principalDao" ref="modelPrincipalDao"/>
</bean>
<!-- do not forget the additional required JAX-RS integration -->
<bean class="de.intarsys.spring.jaxrs.Registration">
  <property name="instance" ref="principalProviderUser" />
</bean>
```

## 18.2.9 StaticPrincipalProvider

*de.intarsys.cloudsuite.gears.model.entity.principal.StaticPrincipalProvider*

A static definition for the context principal.

## Properties

role	
String required	One of the well-known role names (or a custom one).
principal	
GenericPrincipal required	A fully configured instance of GenericPrincipal

## Spring configuration example

## Spring XML fragment

```
<bean id="principalProviderTenant"
class="de.intarsys.cloudsuite.gears.model.entity.principal.StaticPrincipalProvider">
  <property name="role" value="urn:intarsys:names:principal:1.0:role:Tenant" />
  <property name="principal">
    <bean class="de.intarsys.cloudsuite.gears.model.entity.principal.GenericPrincipal">
      <property name="name" value="tenant" />
    </bean>
  </property>
</bean>
```

## 18.2.10 SpringSecurityPrincipalProvider

*[de.intarsys.cloudsuite.gears.security.spring.SpringSecurityPrincipalProvider](#)*

The context principal is derived from the Spring security authentication.

## Properties

role	
String required	One of the well-known role names (or a custom one).
keyConverter	
IKeyConverter optional	An optional conversation function for the Spring "Authentication" object before looking up in the principal DAO
principalDao	
DAO required	A DAO that can look up a principal for the key derived from the Spring authentication data.

## Spring configuration example

## Spring XML fragment

```
<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="keyConverter">
    <bean class="de.intarsys.cloudsuite.gears.security.spring.\
AuthenticationToStringConverter"/>
  </property>
  <property name="principalDao" ref="modelPrincipalDao" />
</bean>
```

## 18.3 Crypto related data models

## 18.3.1 ISslContextProvider

*[de.intarsys.tools.ssl.ConfigurableSslContextProvider](#)*

Create an SSL context based on user defined properties.

## Properties

protocol	
string	The protocol for the SSL implementation.

	This should be one of  TLS   TLSv1   TLSv1.1   TLSv1.2  Default TLS
provider	
string	The name of a well known provider
keyManagerProvider	
IKeyManagerProvider	An optional reference to an IKeyManagerProvider.  This may no be used in conjunction with the direct assignment of keystore properties
keyPassword	
Secret	The password of the key from the keystore
keyStore	
KeyStore	The keystore to be used.
keyStoreName	
string	The file name of the keystore to be used
keyStorePassword	
string	The password for the keystore.
keyStoreType	
string	The type of the keystore implementation.
sslSessionTimeout	
integer	The timeout for cached SSL/TLS connections in seconds.  Default: 86400 (=24 hours)
trustManagerProvider	
ITrustManagerProvider	An optional reference to an ITrustManagerProvider.  This may not be used in conjunction with the direct assignment of truststore properties
trustStore	
KeyStore	The truststore to be used.
trustStoreName	
string	The file name of the truststore to be used
trustStorePassword	
string	The password for the truststore
truststoreStoreType	
string	The type of the truststore implementation.

### 18.3.2 IByteProvider

#### *de.intarsys.tools.crypto.bytes.RandomByteProvider*

Create random bytes from the Java secure random generator.

#### Properties

size	
Integer	The number of bytes created when calling "getBytes". Default: 16

#### *de.intarsys.tools.crypto.bytes.StaticByteProvider*

Create bytes from static input. This provider allows creating bytes from

- text data (UTF-8)
- hex data
- file content

The input can be defined in the configuration or using a system property.

The definition order is

- 3) text from system property
- 4) hex from system property
- 5) file from system property
- 6) text from configuration
- 7) hex from configuration
- 8) file from configuration

#### Properties

propertyText	
String	The name of a system property holding the text. The text is converted to bytes based on UTF-8
propertyHex	
String	The name of a system property holding hex encoded bytes.
propertyFile	
String	The name of a file holding the bytes.
text	
String	Plaintext, converted to bytes using UTF-8
hex	
String	Hex encoded bytes
file	
String	Name of a file holding the bytes

#### [de.intarsys.tools.crypto.bytes.PbkdfByteProvider](#)

Create bytes derived from a password. Internally the algorithm **PBKDF2WithHmacSHA1** is used.

#### Properties

passwordProvider	
IBYTEProvider Required	The password that is processed.
saltProvider	
IBYTEProvider Required	The salt used for processing. You should use a salt of at least 32 bytes.
iterations	
Integer	The number of iterations for the derivation algorithm. Default: 10000
size	
integer	The size of the generated key in bytes. Default: 16

#### [de.intarsys.tools.crypto.bytes.DerivedByteProvider](#)

Derive bytes from input bytes using a key derivation function.

## Properties

kdf	
IKeyDerivationFunction Required	The key derivation function used to derive the input.
inputProvider	
IByteProvider Required	The input to be derived.
size	
integer Optional	The number of bytes returned. The default is to return the same number of bytes as collected from the inputProvider.

## 18.3.3 ICipherFactory

[\*de.intarsys.tools.crypto.standard.JcaCipherFactory\*](#)

Create a JCA Cipher.

## Properties

encryptionAlgorithmTransformation	
string required	Supported & tested algorithms  Transformation AES/CTR/NoPadding      Default for content encryption. AESWrap      Used for key wrapping.
keySize	
integer	Key size of the algorithm.  Default depends on algorithm Algorithm      Size AES      16
blockSize	
integer	Size of a block for block ciphers.  Default depends on algorithm Algorithm      Size AES      16

[\*de.intarsys.tools.crypto.standard.DerivedKeyCipherFactory\*](#)

Create a cipher after deriving a new key. The cipher parameter is filtered by injecting the key material into the KDF and using the result as the key for the wrapped cipher factory.

The IV is unchanged.

## Properties

cipherFactory	
ICipherFactory required	The base ICipherFactory.



kdf	
IKeyDerivationFunction	The IKeyDerivationFunction used to compute the key for the base ICipherFactory

### *de.intarsys.tools.crypto.standard.StaticKeyCipherFactory*

Create a cipher after filtering the key. The cipher parameter key material is **ignored**. The key for the wrapped cipher factory is created from the "keyProvider".

The IV is unchanged.

#### Properties

cipherFactory	
ICipherFactory required	The base ICipherFactory.
keyProvider	
IByteProvider	The IByteProvider for the wrapped ICipherFactory.

### *de.intarsys.tools.crypto.standard.NullCipherFactory*

Create no-operation cipher instances.

## 18.3.4 IKeyDerivationFunction

### *de.intarsys.tools.crypto.kdf.HashKeyDerivationFunction*

Implementation of HKDF.

This should be used, together with a "StaticByteProvider" to setup a key hierarchy.

#### Properties

keyProvider	
IByteProvider required	The base key material that is processed.
saltProvider	
IByteProvider required	The salt bytes used for processing. HKDF requires at least 32 bytes of salt.

### *de.intarsys.tools.crypto.kdf.WithPrefixKeyDerivationFunction*

A "prefix" operation on a base key derivation function.

This should be used to setup a key hierarchy.

#### Properties

kdf	
IKeyDerivationFunction required	The base key derivation function to use.
prefixProvider	
IByteProvider required	These bytes are prefixed to the key derivation input in the form <prefix>"-<input> before calling the base key derivation function.

## 18.4 Plugin reference

Currently there are no plugins defined for the base product.

See [3] for an example.

## 18.5 Action reference

### 18.5.1 Basic actions

#### 18.5.1.1 SetValue

##### 18.5.1.1.1 Description

Set a value in the execution context for later use.

This allows for example to reuse a complex action definition with different arguments from different widgets by simply prepending different "SetValue" actions.

##### 18.5.1.1.2 Arguments

name	
string	The name of the context property
value	
any	The value of the context property

XML example

Spring XML fragment

```
<w:action factory="SetValue">
  <entry key="name" value="foo"/>
  <entry key="value" value="bar"/>
</w:action>
```

JSON example

JSON fragment

```
{
  "try": [{
    "factory": "SetValue",
    "args": {
      "name": "foo",
      "value": "bar"
    }
  }, {
    "factory": "Alert",
    "args": {
      "message?": "foo"
    }
  }
]
```

##### 18.5.1.1.3 Result

The result value is undefined.

#### 18.5.1.2 GetValue

##### 18.5.1.2.1 Description

Get a value from the execution context.

### 18.5.1.2.2 Arguments

name	
string	The name of the context property

XML example

#### Spring XML fragment

```
<w:action factory="GetValue">
  <entry key="name" value="foo"/>
</w:action>
```

JSON example

#### JSON fragment

```
{
  "try": [{
    "factory": "GetValue",
    "args": {
      "name": "foo"
    }
  }]
}
```

### 18.5.1.2.3 Result

The result value is the property value from the execution context.

### 18.5.1.3 Block

#### 18.5.1.3.1 Description

Execute a block of other actions, optionally with a "catch" and "finally" action.

This action is created internally when using the "w:action", "w:catch" and "w:finally" elements from the widget bean definition namespace or the "try/catch/finally" DSL from the JSON action definition.

#### 18.5.1.3.2 Arguments

try	
action	The action to be carried out
catch	
action	The action to be performed if executing "try" encounters an error
finally	
action	An action to be performed at the end of the block, regardless of error state

## XML example

## Spring XML fragment

```

<w:action factory="Confirm">
  <entry key="message" value="OK?"/>
</w:action>
<w:action factory="Alert">
  <entry key="message" value="OK!!"/>
</w:action>
<w:catch factory="Alert">
  <entry key="message" value="failed"/>
</w:catch>
<w:finally factory="Alert">
  <entry key="message" value="finally"/>
</w:finally>

```

## JSON example

## JSON fragment

```

{
  "try": [{
    "factory": "Confirm",
    "args": {
      "message": "OK?"
    }
  }, {
    "factory": "Alert",
    "args": {
      "message": "OK!!"
    }
  }
],
  "catch": {
    "factory": "Alert",
    "args": {
      "message": "catch"
    }
  },
  "finally": {
    "factory": "Alert",
    "args": {
      "message": "finally"
    }
  }
}

```

**18.5.1.3.3 Result**

The result of a block is the result of last action executed.

**18.5.1.4 Throw****18.5.1.4.1 Description**

Throw a JavaScript exception.

**18.5.1.4.2 Arguments**

code	
string	The error code
message	
string	The error message

## XML example

## Spring XML fragment

```
<w:action factory="Throw">
</w:action>
<w:catch factory="Alert">
  <entry key="message" value="failed"/>
</w:catch>
<w:finally factory="Alert">
  <entry key="message" value="finally"/>
</w:finally>
```

## JSON example

## JSON fragment

```
{
  "try": {
    "factory": "Throw"
  },
  "catch": {
    "factory": "Alert",
    "args": {
      "message": "catch"
    }
  },
  "finally": {
    "factory": "Alert",
    "args": {
      "message": "finally"
    }
  }
}
```

**18.5.1.4.3 Result**

-

**18.5.1.5 Log****18.5.1.5.1 Description**

Use the JavaScript "console" to log a message

**18.5.1.5.2 Arguments**

message	
string	The log message

## XML example

## Spring XML fragment

```
<w:action factory="Log">
  <entry key="message" value="log message"/>
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "Log",
  "args": {
    "message": "log message"
  }
}
```

**18.5.1.5.3 Result**

undefined

**18.5.1.6 Echo****18.5.1.6.1 Description**

Echo the spec and the execution args (for testing purposes), e.g., when using ComponentApi.

**18.5.1.6.2 Arguments**

*	
any	Any argument

## XML example

## Spring XML fragment

```
<w:action factory="Echo">
  <entry key="foo" value="bar"/>
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "Echo",
  "args": {
    "foo": "bar"
  }
}
```

**18.5.1.6.3 Result**

An object reflecting the "spec" and "args".

**18.5.2 "ui" support****18.5.2.1 Notice****18.5.2.1.1 Description**

Show a notification ("toast") to the user.

**18.5.2.1.2 Arguments**

type	
string	The type of notification to be used.

	info   warning   error Default is "info"
message	
string required	The message to display

## XML example

## Spring XML fragment

```
<w:action factory="Notice">
  <entry key="message" value="A notice"/>
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "Notice",
  "args": {
    "message": "A notice"
  }
}
```

**18.5.2.1.3 Result**

undefined

**18.5.2.2 Alert****18.5.2.2.1 Description**

Show the user a message, normally in a dialog. The user can only proceed.

**18.5.2.2.2 Arguments**

blocking	
boolean	Flag if this component should be executed blocking (i.e. the JavaScript interpreter holds execution until user feedback has happened using the browser dialogs) or non-blocking (using modal web forms).  Default: false
title	
string	An optional dialog title (this is not supported for browser dialogs)
message	
string	The alert message

## XML example

## Spring XML fragment

```
<w:action factory="Alert">
  <entry key="message" value="An alert"/>
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "Alert",
  "args": {
    "message": "An alert"
  }
}
```

**18.5.2.2.3 Result**

undefined

**18.5.2.3 Prompt****18.5.2.3.1 Description**

Prompt the user for some input that is used in later computation. The user can cancel or proceed.

**18.5.2.3.2 Arguments**

blocking	
boolean	Flag if this component should be executed blocking (i.e. the JavaScript interpreter holds execution until user feedback has happened using the browser dialogs) or non-blocking (using modal web forms).  Default: false
title	
string	An optional dialog title (this is not supported for browser dialogs)
label	
string	An optional label for the entry field (this is not supported for browser dialogs)
message	
string	The message to be displayed
multiline	
boolean	Flag if prompt is using a multiline widget (this is not supported for browser dialogs)
value	
string	The initial value for the prompted text (this is not supported for browser dialogs)

## XML example

## Spring XML fragment

```
<w:action factory="Prompt">
  <entry key="label" value="A prompt"/>
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "Prompt",
  "args": {
    "label": "A prompt"
  }
}
```

**18.5.2.3.3 Result**



undefined

## 18.5.2.4 Confirm

## 18.5.2.4.1 Description

Confirm execution of the current action. The user can cancel or proceed.

## 18.5.2.4.2 Arguments

blocking	
boolean	Flag if this component should be executed blocking (i.e. the JavaScript interpreter holds execution until user feedback has happened using the browser dialogs) or non-blocking (using modal web forms).  Default: false
title	
string	An optional dialog title (this is not supported for browser dialogs)
message	
string	The confirm message

XML example

## Spring XML fragment

```
<w:action factory="Confirm">
  <entry key="message" value="A confirmation"/>
</w:action>
```

JSON example

## JSON fragment

```
{
  "factory": "Confirm",
  "args": {
    "message": "A confirmation"
  }
}
```

## 18.5.2.4.3 Result

undefined

## 18.5.2.5 SetVisibility

## 18.5.2.5.1 Description

Set the widget visibility

## 18.5.2.5.2 Arguments

widget	
string	The name (register id or path) of the widget to manipulate
mode	
string	The working mode of the action <ul style="list-style-type: none"> <li>toggle (Default) Switch the visibility</li> <li>on Switch visibility on</li> </ul>

	<ul style="list-style-type: none"><li>• off Switch visibility off</li></ul>
--	---

XML example

Spring XML fragment

```
<w:action factory="SetVisibility">
  <entry key="widget" value="de.intarsys.widget.toolbar"/>
</w:action>
```

JSON example

JSON fragment

```
{
  "factory": "SetVisibility",
  "args": {
    "widget": "de.intarsys.widget.toolbar"
  }
}
```

### 18.5.2.5.3 Result

undefined

## 18.5.3 "app" actions

### 18.5.3.1 OpenSettings

#### 18.5.3.1.1 Description

Open the settings dialog.

#### 18.5.3.1.2 Arguments

-	

XML example

Spring XML fragment

```
<w:action factory="OpenSettings">
</w:action>
```

JSON example

JSON fragment

```
{
  "factory": "OpenSettings"
}
```

### 18.5.3.1.3 Result

undefined

## 18.5.4 "flow" actions

### 18.5.4.1 SelectPage

#### 18.5.4.1.1 Description

Select a page

#### 18.5.4.1.2 Arguments

index	
number	The 0-based page index to select. If index is < 0, the page is selected from the document end.

XML example

#### Spring XML fragment

```
<w:action factory="SelectPage">
  <entry key="index" value="2"/>
</w:action>
```

JSON example

#### JSON fragment

```
{
  "factory": "SelectPage",
  "args": {
    "index": "2"
  }
}
```

#### 18.5.4.1.3 Result

undefined

### 18.5.4.2 SelectPageNext

#### 18.5.4.2.1 Description

Select the next page.

#### 18.5.4.2.2 Arguments

-	

XML example

#### Spring XML fragment

```
<w:action factory="SelectPageNext">
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "SelectPageNext"
}
```

**18.5.4.2.3 Result**

undefined

**18.5.4.3 SelectPagePrevious****18.5.4.3.1 Description**

Select the previous page.

**18.5.4.3.2 Arguments**

-	

## XML example

## Spring XML fragment

```
<w:action factory="SelectPagePrevious">
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "SelectPagePrevious"
}
```

**18.5.4.3.3 Result**

undefined

**18.5.4.4 ZoomIn****18.5.4.4.1 Description**

Zoom into the page.

**18.5.4.4.2 Arguments**

-	

## XML example

---

**Spring XML fragment**

---

```
<w:action factory="ZoomIn">
</w:action>
```

---

JSON example

---

**JSON fragment**

---

```
{
  "factory": "ZoomIn"
}
```

---

**18.5.4.4.3 Result**

undefined

**18.5.4.5 ZoomOut****18.5.4.5.1 Description**

Zoom out of the page.

**18.5.4.5.2 Arguments**

-	

XML example

---

**Spring XML fragment**

---

```
<w:action factory="ZoomOut">
</w:action>
```

---

JSON example

---

**JSON fragment**

---

```
{
  "factory": "ZoomOut"
}
```

---

**18.5.4.5.3 Result**

undefined

**18.5.4.6 ZoomPageHeight****18.5.4.6.1 Description**

Set the zoom to fit the page height.

**18.5.4.6.2 Arguments**

-	

XML example

**Spring XML fragment**

```
<w:action factory="ZoomPageHeight">
</w:action>
```

**JSON example****JSON fragment**

```
{
  "factory": "ZoomPageHeight"
}
```

**18.5.4.6.3 Result**

undefined

**18.5.4.7 ZoomPageWidth****18.5.4.7.1 Description**

Set the zoom to fit the page width.

**18.5.4.7.2 Arguments**

-	

**XML example****Spring XML fragment**

```
<w:action factory="ZoomPageWidth">
</w:action>
```

**JSON example****JSON fragment**

```
{
  "factory": "ZoomPageWidth"
}
```

**18.5.4.7.3 Result**

undefined

**18.5.4.8 ZoomPage****18.5.4.8.1 Description**

Set the zoom to fit the page width &amp; height.

**18.5.4.8.2 Arguments**

-	

**XML example**

**Spring XML fragment**

```
<w:action factory="ZoomPage">
</w:action>
```

**JSON example****JSON fragment**

```
{
  "factory": "ZoomPage"
}
```

**18.5.4.8.3 Result**

undefined

**18.5.4.9 ZoomTo****18.5.4.9.1 Description**

Set the zoom to the defined scale

**18.5.4.9.2 Arguments**

scale	
number	The scale factor to use

**XML example****Spring XML fragment**

```
<w:action factory="ZoomTo">
  <entry key="scale" value="3.0"/>
</w:action>
```

**JSON example****JSON fragment**

```
{
  "factory": "ZoomTo",
  "args": {
    "scale": "3.0"
  }
}
```

**18.5.4.9.3 Result**

undefined

**18.5.4.10 Ok****18.5.4.10.1 Description**

Commit the current flow and return a "ResultStage" to the caller.

**18.5.4.10.2 Arguments**

-

XML example

Spring XML fragment

```
<w:action factory="Ok">
</w:action>
```

JSON example

JSON fragment

```
{
  "factory": "Ok"
}
```

### 18.5.4.10.3 Result

undefined

### 18.5.4.11 Cancel

#### 18.5.4.11.1 Description

Cancel the current flow and return a "CancelStage" to the caller.

#### 18.5.4.11.2 Arguments

-

XML example

Spring XML fragment

```
<w:action factory="Cancel">
</w:action>
```

JSON example

JSON fragment

```
{
  "factory": "Cancel"
}
```

### 18.5.4.11.3 Result

undefined

### 18.5.4.12 CancelChildren

#### 18.5.4.12.1 Description

Cancel all child operations.

#### 18.5.4.12.2 Arguments

-

XML example



---

**Spring XML fragment**

---

```
<w:action factory="CancelChildren">
</w:action>
```

---

JSON example

---

**JSON fragment**

---

```
{
  "factory": "CancelChildren"
}
```

---

**18.5.4.12.3 Result**

undefined

**18.5.4.13 Download****18.5.4.13.1 Description**

Send a download request to the current flow.

**18.5.4.13.2 Arguments**

--

XML example

---

**Spring XML fragment**

---

```
<w:action factory="Download">
</w:action>
```

---

JSON example

---

**JSON fragment**

---

```
{
  "factory": "Download"
}
```

---

**18.5.4.13.3 Result**

undefined

**18.5.4.14 Print****18.5.4.14.1 Description**

Send a print request to the current flow. The implementation tries its best to directly trigger the printing process, but the outcome depends on the document type, the operating system and browser.

**18.5.4.14.2 Arguments**

-
---

## XML example

## Spring XML fragment

```
<w:action factory="Print">
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "Print"
}
```

**18.5.4.14.3 Result**

undefined

## 18.5.4.15 CollectDepiction

**18.5.4.15.1 Description**

Collect a depiction for a signature from the user.

**18.5.4.15.2 Arguments**

-

## XML example

## Spring XML fragment

```
<w:action factory="CollectDepiction">
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "CollectDepiction",
  "args": {
  }
}
```

**18.5.4.15.3 Result**

depiction
The image that results from the user interaction

## 18.5.4.16 CollectField

**18.5.4.16.1 Description**

Collect a field definition from the user.

**18.5.4.16.2 Arguments**

help.message	
	An optional message that is shown on the overlay
editPermissions	
	<p>A permission string that defines the interactions that can be performed with the field definition interactively in the overlay.</p> <p>This is empty when no restrictions apply.</p> <p>To restrict interactions you can use a concatenation of these tags</p> <p>nw; provide a resize handle in the north-west corner</p> <p>nn; provide a resize handle on the north edge</p> <p>ne; provide a resize handle in the north-east corner</p> <p>ww; provide a resize handle on the west edge</p> <p>ee; provide a resize handle on the east edge</p> <p>sw; provide a resize handle in the south-west corner</p> <p>ss; provide a resize handle on the south edge</p> <p>se; provide a resize handle in the south-east corner</p> <p>mv; support dragging the rect</p> <p>Example</p> <p>ne;nw;se;sw;</p> <p>This example supports only handles on the four corners of the rect.</p>
field	
	An optional upfront definition of field dimensions.
field.x	
	The default x coordinate for the field
field.y	
	The default y coordinate for the field
field.width	
	The default width for the field
field.height	
	The default height for the field

## XML example

## Spring XML fragment

```
<w:action factory="CollectField">
  <entry key="width" value="100"/>
  <entry key="height" value="50"/>
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "CollectField",
  "args": {
    "width": 100,
    "height": 50
  }
}
```

This definition starts the interaction with a field of given size.

#### 18.5.4.16.3 Result

field	
	The field rectangle that is defined by the user.

#### 18.5.4.17 CollectInput

##### 18.5.4.17.1 Description

Collect a depiction for a signature from the user.

##### 18.5.4.17.2 Arguments

title	
string	An optional dialog title
message	
string	An optional message for the dialog
label	
string	An optional label for the entry field
multiline	
boolean	Flag if using a multiline widget.
value	
string	The initial value for the prompted text

XML example

## Spring XML fragment

```
<w:action factory="CollectInput">
  <entry key="message" value="Enter a reason"/>
</w:action>
```

JSON example

## JSON fragment

```
{
  "factory": "CollectInput",
  "args": {
    "message": "Enter a reason"
  }
}
```

#### 18.5.4.17.3 Result

input
-------

	The text entered by the user
--	------------------------------

#### 18.5.4.18 CollectEndorsement

##### 18.5.4.18.1 Description

This action allows to collect a detailed endorsement by going through every document page and requesting explicit acknowledgement of the page content.

##### 18.5.4.18.2 Arguments

x	
	The x coordinate of the annotation that will holds the indication of the user's acknowledgement.
y	
	The x coordinate of the annotation
width	
	The width of the annotation
height	
	The height of the annotation

##### XML example

##### Spring XML fragment

```
<w:action factory="CollectEndorsement">
  <entry key="x" value="10"/>
  <entry key="y" value="10"/>
  <entry key="width" value="50"/>
  <entry key="height" value="50"/>
</w:action>
```

##### JSON example

##### JSON fragment

```
{
  "factory": "CollectEndorsement",
  "args": {
    "x": 10,
    "y": 10,
    "width": 50,
    "height": 50
  }
}
```

Collect the endorsement and render annotations on every page in the lower left corner.

##### 18.5.4.18.3 Result

endorsement	
	The endorsement details

#### 18.5.4.19 Sign

##### 18.5.4.19.1 Description

Request a signature process for the current flow

#### 18.5.4.19.2 Arguments

requireField	
boolean	<p>Flag if the user is required to select a field for the signature.</p> <p>This is a shortcut for chaining a "CollectField" action (see above)</p> <p>Default false</p>
requireSignature	
boolean	<p>Flag if a handwritten "signature" is required.</p> <p>This is a shortcut for chaining a "CollectDepiction" action (see above)</p> <p>Default false</p>
requireInput	
boolean	<p>Flag if a text dialog is required. This text dialog will receive a custom text that is rendered in the final signature appearance.</p> <p>This is a shortcut for a "CollectInput" action (see above).</p> <p>Default false</p>
inputTitle	
string	An optional title for the input dialog.
field	
object	<p>An optional definition for the signature field position and size.</p> <pre>{   "x": 0,   "y": 0,   "width": 10,   "height": 10 }</pre>
pageRange	
object	The page for the signature. If no pageRange is defined, the current viewer page is used.
signerCreate.*	
object	<p>These arguments are directly forwarded to the signer/create gears service implementation.</p> <p>You can use the well-known arguments like</p> <ul style="list-style-type: none"> <li>• configuration</li> <li>• args</li> <li>• ...</li> </ul>

XML example

Spring XML fragment

```
<w:action factory="Sign">
  <entry key="requireField" value="true"/>
  <entry key="requireSignature" value="true"/>
  <entry key="signerCreate.configuration" value="mySignerConfiguration"/>
  <entry key="signerCreate.args" value="documentSigner.args.signatureLabel=Hi,
there"/>
</w:action>
```

JSON example

## JSON fragment

```
{
  "factory": "Sign",
  "args": {
    "requireField": true,
    "requireSignature": true,
    "signerCreate.configuration": "mySignerConfiguration",
    "signerCreate.args": "documentSigner.args.signatureLabel=Hi, there"
  }
}
```

**18.5.4.19.3 Result**

Undefined

## 18.5.4.20 AnnotationEdit

**18.5.4.20.1 Description**

Manipulate the current annotation.

**18.5.4.20.2 Arguments**

action	
string	The action to perform on the server side with the target annotation. Currently supported: <ul style="list-style-type: none"> <li>clear Clear all content for the signature annotation</li> </ul>
confirm	
string	An optional message to prompt before performing the action.

## XML example

## Spring XML fragment

```
<w:action factory="AnnotationEdit">
  <entry key="action" value="clear"/>
  <entry key="confirm" value="Do you really want to delete the signature?"/>
</w:action>
```

## JSON example

## JSON fragment

```
{
  "factory": "AnnotationEdit",
  "args": {
    "action": "clear",
    "confirm": "Do you really want to delete the signature?"
  }
}
```

**18.5.4.20.3 Result**

undefined

**18.6 Widget reference**

### 18.6.1 Overview

A widget type is an optional attribute. If absent, the default type is derived dependent on the context. E.g., when used in a toolbar, the default type is "button".

### 18.6.2 Group

A widget has the implicit type "Group" when it has no explicit type **and** it has children.

---

#### Spring XML fragment

```
<w:widget icon="search">
  <w:widget icon="search-plus">
    <w:on event="select" do="ZoomIn"/>
  </w:widget>
  <w:widget icon="search-minus">
    <w:on event="select" do="ZoomOut"/>
  </w:widget>
  <w:widget icon="arrows-v">
    <w:on event="select" do="ZoomPageHeight"/>
  </w:widget>
  <w:widget icon="arrows-h">
    <w:on event="select" do="ZoomPageWidth"/>
  </w:widget>
  <w:widget icon="arrows">
    <w:on event="select" do="ZoomPage"/>
  </w:widget>
</w:widget>
```

### 18.6.3 Button

A "Button" widget renders a label and an icon, depending on the css style and other properties.

---

#### Spring XML fragment

```
<w:widget parent="de.intarsys.widget.toolbar.additions" type="Button">
  <w:icon descriptor="far:smile" css="color: red;"/>
  <w:label message="Say hi" css="font-size: larger;"/>
  <w:on event="select" action="Alert">
    <entry key="message" value="hi"/>
  </w:on>
</w:widget>
```

### 18.6.4 Toggle button

A "Toggle" widget renders a label and an icon. The toggle button can represent two states, "pressed" and "not pressed", indicating an application state.



---

**Spring XML fragment**

---

```
<w:widget
  parent="de.intarsys.widget.toolbar.additions"
  type="Toggle"
  icon="search-plus"
  label="Toggle me"
>
  <w:on event="selectChecked" action="SetValue">
    <entry key="name" value="$page.myvar"/>
    <entry key="value" value="false"/>
  </w:on>
  <w:on event="selectUnchecked" action="SetValue">
    <entry key="name" value="$page.myvar"/>
    <entry key="value" value="true"/>
  </w:on>
  <w:on event="isChecked" action="GetValue">
    <entry key="name" value="$page.myvar"/>
  </w:on>
</w:widget>
```

---

### 18.6.5 Separator

A "Separator" can be used in a toolbar definition to render a vertical bar between items.

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.toolbar.additions" type="Separator">
</w:widget>
```

---

### 18.6.6 Label

A "Label" can be used to render a plain text label in the toolbar.

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.toolbar.additions" type="Label" label="test">
</w:widget>
```

---

### 18.6.7 Spacer

A "Spacer" can be used to create a forced space between items in a toolbar.

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.toolbar.additions" type="Spacer">
</w:widget>
```

---

### 18.6.8 de.intarsys.ui.control.PageSelector

A "de.intarsys.ui.control.PageSelector" can be used to render a page selection component in the toolbar.

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.toolbar.additions" type="PageSelector">
</w:widget>
```

---

### 18.6.9 Toolbar

This is a screen estate at the top of the main view that is used for actions. It is a child of "de.intarsys.widget.root".

It is automatically created in the default widget definition.

### 18.6.10 ControlOverlay

This component can be rendered as a child of "de.intarsys.widget.renderer.overlays"

This is an incubating feature, see [3].

#### Spring XML fragment

```
<w:widget parent="de.intarsys.widget.renderer.overlays" type="ControlOverlay">
  <w:widget id="toolbar">
    <w:widget icon="arrow-left">
      <w:on event="select" do="PagePrevious"/>
    </w:widget>
    <w:widget icon="arrow-right">
      <w:on event="select" do="PageNext"/>
    </w:widget>
  </w:widget>
</w:widget>
```

### 18.6.11 AnnotationsOverlay

This component can be rendered as a child of "de.intarsys.widget.renderer.overlays".

#### Spring XML fragment

```
<w:widget parent="de.intarsys.widget.renderer.overlays" type="AnnotationsOverlay">
  <w:widget id="canvas" type="switch">
    <w:widget type="default">
      <w:on event="select">
        <w:action factory="Notice">
          <entry key="message" value="nothing to see here..."/>
        </w:action>
      </w:on>
    </w:widget>
  </w:widget>
</w:widget>
```

### 18.6.12 SidebarContainer

This is a screen estate at the left of the main view that is used for detail information. It is a child of "de.intarsys.widget.root".

If not explicitly created by the configuration, a default instance is set up automatically. You can overwrite the default any time by defining the respective widget on your own.

Properties

collapsible	
Boolean	Flag if this sidebar instance should be collapsible

Example

To correctly set up the sidebar, you must mention all of the properties "parent", "register" and "id"! This configuration will create a sidebar that cannot be collapsed by the user.

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.root" register="de.intarsys.widget.sidebar"
id="sidebar">
  <w:property name="collapsible" value="false"/>
</w:widget>
```

---



---

**JSON fragment**

---

```
{
  "parent": "de.intarsys.widget.root",
  "register": "de.intarsys.widget.sidebar",
  "id": "sidebar",
  "properties": {
    "collapsible": false
  }
}
```

---

### 18.6.13 ThumbnailsSidebar

This component can be rendered as a child of "de.intarsys.widget.sidebar.components".

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.sidebar.components" id="thumbnails"
type="ThumbnailsSidebar">
</w:widget>
```

---

### 18.6.14 PropertiesSidebar

This component can be rendered as a child of "de.intarsys.widget.sidebar.components".

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.sidebar.components" id="properties"
type="PropertiesSidebar">
</w:widget>
```

---

### 18.6.15 SignaturesSidebar

This component can be rendered as a child of "de.intarsys.widget.sidebar.components".

---

**Spring XML fragment**

---

```
<w:widget parent="de.intarsys.widget.sidebar.components" id="signatures"
type="SignaturesSidebar">
</w:widget>
```

---

## 18.7 Well known widgets

This chapter enumerates all publicly available widget containers and widgets. Do not reference widgets in your extensions that are not listed here – they are subject to change without notice.

*de.intarsys.widget.renderer*

---

type	
	Container
context	
	-
description	
	A container for all renderer related widgets

### *de.intarsys.widget.renderer.overlays*

---

type	
	Container
context	
	-
description	
	The overlays to be rendered in the viewer

### *de.intarsys.widget.root*

---

type	
	Container
context	
	-
description	
	The root of all widgets

### *de.intarsys.widget.shortcuts*

---

type	
	Container
context	
	-
description	
	The shortcuts for the app

### *de.intarsys.widget.sidebar*

---

type	
	Container
context	
	-
description	
	A container for all sidebar related widgets

### *de.intarsys.widget.sidebar.components*

---

type	
	Container
context	
	-
description	
	The pluggable contents of the sidebar.

### *de.intarsys.widget.toolbar*

---

type	
	Container

context	
	-
description	
	A container for all toolbar widgets

### *de.intarsys.widget.toolbar.additions*

---

type	
	Container
context	
	-
description	
	The additions section of the toolbar (trailing part of toolbar left section).

### *de.intarsys.widget.toolbar.center*

---

type	
	Container
context	
	-
description	
	The center section of the toolbar.

### *de.intarsys.widget.toolbar.left*

---

type	
	Container
context	
	-
description	
	The left section of the toolbar

### *de.intarsys.widget.toolbar.right*

---

type	
	Container
context	
	-
description	
	The right section of the toolbar.

### *de.intarsys.widget.renderer.overlays/annotations*

---

type	
	AnnotationsOverlay
context	
	-
description	
	The default annotation overlay

### *de.intarsys.widget.renderer.overlays/control*

---

type	
	ControlOverlay
context	
	-

description	
	The default control overlay

### *de.intarsys.widget.sidebar.components/properties*

type	
	PropertiesSidebar
context	
	-
description	
	The default document properties component in the sidebar

### *de.intarsys.widget.sidebar.components/signatures*

type	
	SignaturesSidebar
context	
	-
description	
	The default document signatures component in the sidebar

### *de.intarsys.widget.sidebar.components/thumbnails*

type	
	ThumbnailsSidebar
context	
	-
description	
	The default document thumbnails component in the sidebar

### *\*/annotations*

type	
	Container
context	
	AnnotationsOverlay
description	
	Virtual annotations that are rendered in the annotation overlay

### *\*/canvas*

type	
	Container
context	
	AnnotationsOverlay
description	
	Description of the document annotations that are rendered in the annotation overlay

### *\*/popup*

type	
	Container
context	
	AnnotationsOverlay
description	
	Description of the popup for an annotation rendered in the annotation overlay

*\*/toolbar*

---

type	
	Container
context	
	ControlOverlay
description	
	The widgets to be rendered in the control overlay

## 19. Implementation hints

---

### 19.1 Chunked transfer

Depending on your client configuration and the size of the document list to transfer, you may experience memory problems on the client.

This may be caused by using default settings when calling the web API of Sign Live! cloud suite gears core. The standard Java HTTP connection classes for example will try to detect the content length for the HTTP header before sending a payload. If it is not set explicitly, all data will be written to a byte array upfront. Large documents will cause an `OutOfMemoryException` then.

Be sure to set a content length upfront or force the transfer to be executed using chunked encoding. Using a JAX-RS client this can be done like shown below

---

#### Java code fragment

```
protected Invocation.Builder createBuilder(String path) throws IOException {
    Invocation.Builder builder = getClient() //
        .property( //
            ClientProperties.REQUEST_ENTITY_PROCESSING, //
            RequestEntityProcessing.CHUNKED) //
        .target(getGearsCoreUrlServer()) //
        .path(CSURL_PATH_PREFIX) //
        .path(path) //
        .request();
    return builder;
}
```

---



## 20. String expansion

### 20.1 Disclaimer

Before diving into the details of string expansion, it is crucial to understand that two distinct mechanisms are at play within Sign Live! cloud suite gears:

- Spring's string expansion at startup and
- gears' runtime string expansion.

These two systems operate at different stages of the application lifecycle and can lead to unexpected behavior if not handled correctly. For more details see chapter 20.2.6.

Another important note is, that NLS files (messages.properties) have special characteristics. Therefore, string expansion behaves slightly differently (see chapter 17.6).

The following chapters describe in detail, how string expansion works for configuration files and other places which are available during runtime.

### 20.2 Basics

#### 20.2.1 Why string expansion

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

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

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

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

#### 20.2.2 Terminology

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

- Expression evaluation

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

- Template evaluation

While expression evaluation is quite useful by itself, it still can be improved. One problem you will encounter, is how to differentiate a plain piece of text from a variable to be

replaced, the other problem results from the need for more complex content that cannot be expressed using a single variable. Look for example at a directory name, that should be built using the temporary directory plus the name of the current user.

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

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

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

### 20.2.3 Syntax

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

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

The variable *foo* is embedded in the string.

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

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

will evaluate to

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

### 20.2.4 Constant text in expression

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

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

will evaluate to

```
hello, world.
```

### 20.2.5 Namespaces

gears' String expansion can resolve an extensive set of named values from various sources. The values are organized into hierarchical namespaces. Accordingly, a value name can have a prefix of one or more namespace name separated by dots “.”, which specifies the path to the value. For example, our information is organized using the prefixes

- **config** to define your own properties (see chapter 20.3.3) and make them accessible to the gears' runtime string expansion.
- **properties** for selecting among VM properties (chapter 20.3.11)
- **environment** for selecting execution context information like working directories (chapter 20.3.7) and many more.

## Example

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

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

You will find a complete description of the namespaces available in chapter 20.3.

**NOTE** Some namespaces grant access to sensitive data and are only available for string expansion of trusted data. This restriction can be selectively lifted in the configuration (see section 9.13).

## 20.2.6 Spring integration

The concept of string expansion is used throughout the Sign Live! product family. With gears we are facing the problem, that Spring uses the same escapes with their own string expansion implementation.

A problem arises when we want to use a template at runtime and configure it in a Spring XML or property file. Consider we want "signme.signer.username" to be replaced with the principal name at runtime. This example won't work:

```
signme.signer.username=${principal.user.name}
```

Spring will try to expand the value upon startup and fail, because it does not know the variable principal.user.name. To be able to forward string expansion to the runtime expansion, we have to escape the escape...

One solution consists in reverting to the Spring expression language – resulting in an unreadable and error prone construct like this:

```
signme.signer.username=#{'$' + '{principal.user.name}'}
```

To support a more readable alternative, the application will finish Spring template processing by replacing all occurrences of "{?" with "\${". This way you can (and should) write:

```
signme.signer.username=?{principal.user.name}
```

whenever you need your string expanded at runtime, not startup time.

Attention:

Replacing "{?" is done in spring configuration files only! Do not use this workaround in service arguments and other places which are only available during runtime.

In summary remember these key differences:

Use \${}:

- in Spring configuration files like .xml and .properties, when you want to use Spring's string expansion during startup.
- in service arguments or other places which are only available during runtime, this uses the gears' runtime string expansion. Only variables from available namespaces (chapter 20.3) can be accessed.

Use ?{}

- in Spring configuration files like .xml and .properties, when you want to use gears' runtime string expansion. E.g. when you want to access a variable that is not available during startup or when you want to use gears' formatting features

(see chapter 20.4). Only variables from available namespaces (chapter 20.3) can be accessed.

- In Spring configuration files to access NLS message files.

Use `${ }`

- inside NLS message files. NLS files (`messages.properties`) have special characteristics. Therefore, string expansion behaves slightly differently (see chapter 17.6).  
Important: NLS files are evaluated when used by another variable.

See the tutorial chapter 20.5 for a hands-on example.

## 20.3 Namespaces

### 20.3.1 Overview

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

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

### 20.3.2 app

#### 20.3.2.1 Name

app

#### 20.3.2.2 Description

Access to information about the application.

#### 20.3.2.3 Variables

Name	Description
<b>name</b>	The application's name.
<b>version</b>	The full version of the application.
<b>major</b>	The major version of the application (may be empty).
<b>minor</b>	The minor version of the application (may be empty).
<b>micro</b>	The micro version of the application (may be empty).

#### 20.3.2.4 Availability

This namespace is available after application startup.

### 20.3.3 config

#### 20.3.3.1 Name

config

#### 20.3.3.2 Description

This namespace should be used for defining new properties, which are made available to gears' string expansion.

#### 20.3.3.3 Variables

Name	Description
*	Any Spring property starting with "config."

#### 20.3.3.4 Availability

This namespace is available after application startup.

#### 20.3.3.5 Example

To understand the difference between `?{}` and `${}` see the chapters 20.2.6 and 20.5.  
Given an entry in the `gears.properties`

```
config.foo=bar
```

this expression

```
example ${config.foo}
or      ?{config.foo}
```

will evaluate to

```
example bar
```

### 20.3.4 counters

#### 20.3.4.1 Name

counters

#### 20.3.4.2 Description

Zero-based all-purpose counters.

#### 20.3.4.3 Variables

Name	Description
<b>&lt;name&gt;</b>	An arbitrarily named counter, for example, for creating unique ids. On the first request, the counter is initialized and returns 0. Each consecutive usage increments the counter and returns the new value. Counters are not persistent!

20.3.4.4 Availability

This namespace is generally available and usually used during runtime.

20.3.4.5 Examples

To understand the difference between `?{}` and `${}` see the chapters 20.2.6 and 20.5.

```
example ${counters.myCounter}  
or      ?{counters.myCounter}
```

will evaluate to

```
example 0
```

at the first usage. It evaluates to

```
example 1
```

at the second usage (and so on...).

20.3.5 digestSigner

20.3.5.1 Name

digestSigner

20.3.5.2 Description

Access detail information from the current signer process.

20.3.5.3 Variables

Name	Description
subject	An X500Name object that represents the subject
issuer	An X500Name object that represents the issuer
signatureEvidence	A SignatureEvidence object that comprises information for the signed content of this process

X500Name properties

Typical variables of an X500 name object are enumerated here. Be aware that not every certificate contains every possible property.

Name	Description
cn	The entity common name
givenname	The entity given name
surname	The entity surname
title	The entity title
emailaddress	The entity email address

<b>c</b>	The country
<b>o</b>	The organization
<b>ou</b>	The organizational unit

SignatureEvidence properties

Name	Description
<b>items</b>	A list of SignatureItemEvidence

SignatureItemEvidence properties

Name	Description
<b>label</b>	The name of the signed document
<b>digest</b>	The digest of the signed document

20.3.5.4 Availability

This namespace is available in the context of a signature process.

20.3.5.5 Example

To understand the difference between <code>?{}</code> and <code>\${}</code> see the chapters 20.2.6 and 20.5.example	Signed by <code>\${digestSigner.subject.cn}</code>
or	Signed by <code>?{digestSigner.subject.cn}</code>

will evaluate to something like

Signed by Alexander, the great
--------------------------------

20.3.6 entity

20.3.6.1 Name

entity

20.3.6.2 Description

Characters that are difficult to type or to use in some contexts.

20.3.6.3 Variables

Name	Description
<b>amp</b>	ampersand &
<b>backslash</b>	backslash \

<b>copy</b>	copyright © (Unicode U+00A9)
<b>cr</b>	carriage return (Unicode U+000D, \r in Java strings)
<b>gt</b>	greater than >
<b>lf</b>	line feed (Unicode U+000A, \n in Java strings)
<b>lt</b>	less than <
<b>nl</b>	line separator of the VM (usually \n on Unix and \r\n on Windows)
<b>quot</b>	double quote "
<b>squot</b>	single quote '
<b>slash</b>	slash /
<b>trade</b>	trademark ® (Unicode U+2122)

#### 20.3.6.4 Availability

This namespace is always available.

### 20.3.7 environment

#### 20.3.7.1 Name

environment

#### 20.3.7.2 Description

Access to the application's file environment. All paths are absolute.

#### 20.3.7.3 Variables

Name	Description
<b>basedir</b>	The application's base directory. Most operations will be performed relative to this directory.
<b>profiledir</b>	The directory for user-specific data.
<b>datadir</b>	The directory for application-private data.
<b>workingdir</b>	The application's working directory.
<b>tempdir</b>	The application's directory for temporary files.

#### 20.3.7.4 Availability

This namespace is available after application startup for trusted use cases.

### 20.3.8 flow

#### 20.3.8.1 Name

flow



### 20.3.8.2 Variables

Name	Description
<b>id</b>	The id of the flow (it is equivalent to the conversation id)
<b>variables.&lt;name&gt;</b>	Any variable that was assigned to the flow when created (via variables argument or configuration)

### 20.3.8.3 Description

Access flow context information

### 20.3.8.4 Availability

This namespace is available in the context of a flow service execution.

### 20.3.8.5 Examples

To understand the difference between `{}` and `${}`  see the chapters 20.2.6 and 20.5. With a message resource like the one added in chapter 17.5.

```
example ${flow.id}
or      ?{flow.id}
```

will evaluate to the conversation id for the flow

```
example faadc46e-c367-4963-b497-eb607b8d3f6a
```

```
example ${flow.variables.test}
or      ?{flow.variables.test}
```

will evaluate to the value that was set for the variables argument (or any variable assigned in a configuration) when the flow was created.

```
example bar
```

## 20.3.9 identifiers

### 20.3.9.1 Name

identifiers

### 20.3.9.2 Description

Generation of unique identifiers

### 20.3.9.3 Variables

Name	Description
<b>uuid</b>	Yields a new UUID on each request, which is represented as 32 hexadecimal digits, displayed in five groups separated by hyphens (for example, 123e4567-e89b-12d3-a456-426614174000) .

#### 20.3.9.4 Availability

This namespace is generally available.

### 20.3.10 nlsmsg

#### 20.3.10.1 Name

nlsmsg

#### 20.3.10.2 Description

Access a NLS message resource.

#### 20.3.10.3 Variables

Name	Description
*	A message resource path according to the definition in chapter 17.

#### 20.3.10.4 Availability

This namespace is available after application startup.

#### 20.3.10.5 Example

To understand the difference between `{}` and `${}`  see the chapters 20.2.6 and 20.5.  
With a message resource like the one added in chapter 17.5.

```
Hi ${nlsmsg.de.intarsys.gears.core.demo.ui.messages#MyButton.label}
or
Hi ?{nlsmsg.de.intarsys.gears.core.demo.ui.messages#MyButton.label}
```

will evaluate to

```
Hi Tolle Beschriftung
```

### 20.3.11 properties

#### 20.3.11.1 Name

properties

#### 20.3.11.2 Description

Access to properties in the application's Spring environment.

#### 20.3.11.3 Variables

Name	Description
*	Any property in the Spring environment.

#### 20.3.11.4 Availability

This namespace is available after application startup for trusted use cases.

### 20.3.12 system

#### 20.3.12.1 Name

system

#### 20.3.12.2 Description

Access some system information.

#### 20.3.12.3 Variables

Name	Description
<b>architecture</b>	Returns the architecture of the current platform, which is either “64-bit” or “32-bit”.
<b>getenv.&lt;name&gt;</b>	The value of the environment variable <name>.
<b>properties.&lt;name&gt;</b>	The value of the VM’s system property <name>.

#### 20.3.12.4 Availability

This namespace is only available for trusted use cases.

#### 20.3.12.5 Example

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

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

```
→ example 32
```

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

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

```
→ example 0
```

### 20.3.13 time

#### 20.3.13.1 Name

time

#### 20.3.13.2 Description

Time-related properties.

#### 20.3.13.3 Variables

Name	Description
<b>millis</b>	The difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.

**uniqueMillis**

The difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. If multiple values are requested within the same millisecond, the result is delayed so that each value is unique.

## 20.3.13.4 Availability

This namespace is generally available.

## 20.4 Formatting

### 20.4.1 Overview

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

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

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

```
#{foo:f}
```

or, in the case of hierarchical names

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

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

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

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

Attention:

Formatting is part of gears' runtime string expansion, therefore you must use `?{}` in Spring configuration files like `.xml` and `.properties`. In service arguments and other places which are available only during runtime you must use `#{}` (see chapter 20.2.6).

### 20.4.2 String formatting

#### 20.4.2.1 Overview

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

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

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

### 20.4.2.2 Instruction

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

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

### 20.4.2.3 Examples

To understand the difference between `?{}` and `${}` see the chapters 20.2.6. and 20.5.

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

```
example ${config.user.greeting:s}
or      ?{config.user.greeting:s}
```

evaluate to

```
→ example hello, world
```

```
example ${config.user.greeting:s(7)}
or      ?{config.user.greeting:s(7)}
```

evaluate to

```
→ example world
```

```
example ${config.user.greeting:s(0,4)}
or      ?{config.user.greeting:s(0,4)}
```

evaluate to

```
→ example hello
```

## 20.4.3 Integer formatting

Integer number formatting is initiated by `i`.

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

### 20.4.3.1 Instruction

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

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

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

### 20.4.3.2 Examples

To understand the difference between `{}` and `${}` see the chapters 20.2.6. and 20.5.  
With *system.counter* containing '17'

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

evaluate to

→ example 17

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

evaluate to

→ example 10001

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

evaluate to

→ example 21

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

evaluate to

→ example 17

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

evaluate to

→ example 11

## 20.4.4 Date formatting

Date formatting is initiated by **d**.

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

### 20.4.4.1 Instruction

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

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

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

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

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

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

The optional second instruction character defines the output format

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

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

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

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

### 20.4.4.2 Examples

To understand the difference between `{}` and `${}` see the chapters 20.2.6. and 20.5.

With `'time.millis'` containing a timestamp for the 12th of October, 2009 at 23 :33:11 and 1 milliseconds.

```
example ${time.millis:d}
or      ?{time.millis:d}
```

evaluate to

→ example 2009\_10\_12-23\_33\_11\_001

```
example ${time.millis:ddm}
or      ?{time.millis:ddm}
```

evaluate to

```
→ example 12.10.2009
```

```
example ${time.millis:dts}
or      ?{time.millis:dts}
```

evaluate to

```
→ example 23:33:11
```

## 20.4.5 Float formatting

Float formatting is initiated by **f**.

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

### 20.4.5.1 Instruction

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

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

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

### 20.4.5.2 Examples

To understand the difference between **?{}** and **\${}** see the chapters 20.2.6. and 20.5. With `config.variables.user.price` containing **1234.567** and an US locale

```
example ${config.user.price.f(0.0)}
or      ?{config.user.price.f(0.0)}
```

evaluate to

```
→ example 1,234.6
```

```
example ${config.user.price.f(000000)}
or      ?{config.user.price.f(000000)}
```

evaluate to



---

→ example 001235

---

## 20.4.6 File path formatting

File path formatting is initiated by **p**.

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

### 20.4.6.1 Instruction

---

```
expr ":p"
```

---

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

### 20.4.6.2 Examples

To understand the difference between **{}** and **\${}** see the chapters 20.2.6. and 20.5.  
With `config.variables.user.foo` containing **my\*?.file**

---

```
example ${config.user.foo:p}
or      ?{config.user.foo:p}
```

---

evaluate to

---

→ example my\_.file

---

## 20.4.7 Default value

If the variable cannot be resolved, normally an exception is raised, either the current processing is terminated or your template will contain some text like "<expression evaluation failed>".

The default value instruction gives you control on what to do when evaluation fails.

### 20.4.7.1 Instruction

---

```
expr ":!"
```

---

Apply the expression after the **!"** if the first one fails or evaluates to nothing.

### 20.4.7.2 Examples

To understand the difference between **{}** and **\${}** see the chapters 20.2.6. and 20.5.  
Assuming that "foo" is undefined and "config.bar" holds "hello"

---

```
${foo:!config.bar} world
or
?{foo:!config.bar} world
```

---

evaluate to

```
hello world
```

You can use literal expressions as default, too.

```

${foo:!'hello'} world
and
${foo:!"hello"} world
or
?{foo:!'hello'} world
and
?{foo:!"hello"} world

```

evaluate to

```
hello world
```

## 20.4.8 Recursion

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

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

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

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

### 20.4.8.1 Instruction

```
expr ":*"
```

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

The nesting depth of recursive evaluations is restricted to 10. Remember that you can chain formatting instructions, for example to apply a string formatting first, followed by a deep evaluation.

### 20.4.8.2 Example

To understand the difference between `?{}` and `${}` see the chapters 20.2.6. and 20.5. So, the complete example for the above should be:

```

example ${config.global.greeting:*}! Your application is fully functional!
or      ?{config.global.greeting:*}! Your application is fully functional!

```

evaluates to

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

## 20.4.9 Conditional evaluation

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

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

### 20.4.9.1 Instruction

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

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

"condition" can be any expression that itself can be evaluated to "true", "t", "1" for →TRUE or "false"; "f", "0" for → FALSE.

To ease handling of conditions, "!" can be used to negate the condition result.

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

### 20.4.9.2 Example

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

The above case for example may be written:

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

Or

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

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

To understand the difference between *?{}* and *\${}* see the chapters 20.2.6. and 20.5.

## 20.5 Tutorial

In this section we practice and demonstrate the learnings from the chapters above in example scenarios.

Prerequisites: gears core and demo apps are installed.

Our goal is to create a visual signature with a label, which contains: date of signing, a globally defined company name, and time of signing. The examples are increasingly complex.

## 20.5.1 Example 1: Hardcoded signature label

In your downloaded product bundle, you can find the folder “..example/tutorial/spring expansion”. Place its content into your `cloudsuite.config.shared` directory (see chapter 21.1). You may want to keep these files open as a reference.

Start the gears core and demo applications.

Now open the demo application in the browser:

`http://<host>:8080/cloudsuite-gears/demo`

Login with a random username (password is not required).

Now head to `http://<host>:8080/cloudsuite-gears/demo/ng/app/settings` by clicking on the gear icon on the demo documents page.



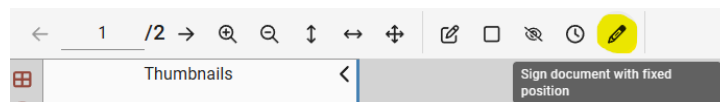
In the input field “Signer configuration” we enter the `FlowSignerConfiguration` id:

```
signatureLabelHardCoded
```

Head back to the documents page

First, let’s sign a document with a signature label:

1. Create an empty document by doing “ctrl + left click” on the upload button; or upload any pdf document.
2. Open the viewer for the uploaded document.
3. Perform “Sign document with fixed position” by clicking on the pencil icon



4. Scroll to the bottom of the document and observe the signature label. You should be able to see a signature with a label:

Hardcoded signature label

To understand how this works, navigate to `cloudsuite.config.shared` and inspect the signer configuration file called “spring-gears-tutorial-string-expansion.xml”. Search for the `FlowSignerConfiguration` bean where `id=signatureLabelHardCoded`.

The signer is configured with signer arguments (`documentSigner.args`). One of those is called `signatureLabel`:

```
<entry key="documentSigner.args.signatureLabel" value="Hardcoded signature label" />
```

Here the value is simply the string you see displayed.

## 20.5.2 Example 2: Signature label via Spring string expansion

Head back to the gears demo settings page and change the signer configuration to:

```
signatureLabelWithVariableSpring
```

Now, create another signature with a label by repeating the steps from chapter 20.5.1. The signature label should now look like this:

```
Signed by Tutorial Company
```

Again, take a look at the FlowSignerConfiguration bean. This time search for the id "signatureLabelWithVariableSpring":

```
entry key="documentSigner.args.signatureLabel" value="Signed by  
${config.company.name}" />
```

Because of `${}`, Spring does string expansion at app startup (see chapter 20.2.6). It finds the variable in our `gears.properties` file and replaces "config.company.name" with "Tutorial Company".

See the `gears.properties` file in the `cloudsuite.config.shared` directory.

```
config.company.name=Tutorial Company
```

### 20.5.3 Example 3: Signature label via gears string expansion

Head back to the gears demo settings page and change the signer configuration to:

```
timestampWithVariableGears
```

Now, create another signature with a label by repeating the steps from chapter 20.5.1. The signature label should now look like this:

```
Signed on 13/02/2025 10:17
```

Again, take a look at the FlowSignerConfiguration bean. This time search for the id "timestampWithVariableGears":

Spring would not be able to evaluate `time.millis`, because "time" is a namespace (see chapter 20.3.13) which can be accessed via gears string expansion, but not by Spring. Therefore we have to use `?{}` to prevent spring from performing string expansion. If we had used `${}` the application would not be able to start.

```
<entry key="documentSigner.args.signatureLabel"  
value="Signed on ?{time.millis:d(dd/MM/yyyy HH:mm)}" />
```

Another reason for using `?{}` in the `gears.properties` file is: We do date formatting (see 20.4.4), which is always done via gears string expansion and not by Spring (see chapter 20.2.6).

### 20.5.4 Example 4: Signature label via global variable

Head back to the gears demo settings page and change the signer configuration to:

```
signatureLabelWithVariableSpringGears
```

Now, create another signature with a label by repeating the steps from chapter 20.5.1. The signature label should now look like this:

```
Signed by username on 13/02/2025 10:40
```

Again, take a look at the FlowSignerConfiguration bean. This time search for the id “signatureLabelWithVariableSpringGears”:

```
<entry key="documentSigner.args.signatureLabel"
      value="${tutorial.signatureLabel}" />
```

Because of `${}`, Spring does string expansion at app startup. It finds the variable in our gears.properties file and replaces “tutorial.signatureLabel” with the value of said variable (see chapter 20.2.6).

See the gears.properties file in the cloudsuite.config.shared directory:

Because of `${}`, Spring does string expansion at app startup (see chapter 20.2.6). It finds the variable in our gears.properties file and replaces “config.timestamp” with the value of said variable.

```
tutorial.signatureLabel=Signed by ?{principal.user.name} on ${config.timestamp}
```

We use `?{}` because the namespace “principal” is only available during runtime. Therefore, Spring would not be able to evaluate the variable during app startup. If we used `${}` the application would not be able to start (see chapter 20.2.6). In this context “.user.name” is the username with which you have logged into the gears demo application.

See the gears.properties file in the cloudsuite.config.shared directory.

The details of the timestamp format were already explained in chapter 20.5.3.

```
config.timestamp=?{time.millis:d(dd/MM/yyyy hh:mm)}
```

## 20.5.5 Example 5: Signature label via signer request

Head back to the gears demo settings page and empty the input field “signer configuration”.

Paste the following into the input field called “Signer args”:

```
documentSigner.args.signatureLabel=Signed by ${principal.user.name} on
${config.timestamp}
```

Now, create another signature with a label by repeating the steps from chapter 20.5.1. The signature label should now look the same as in chapter 20.5.4:

```
Signed by username on 13/02/2025 11:17
```

Explanation:

We didn't set a "signer configuration", therefore the default configuration (demoPlain) is used. The default configuration also holds the argument "signatureLabel", which is overwritten here by setting it in the service call.

Again, we use the namespace "principal" which is available at runtime.

```
documentSigner.args.signatureLabel=Signed by ${principal.user.name} on
${config.timestamp}
```

In this case we use \${}. We are setting a service argument during runtime; therefore, this variable is not evaluated by Spring but via the gears string expansion mechanism (see chapter 20.2.6).

Here we use the "config" namespace (chapter 20.3.3) because it's the only configurable namespace gears string expansion has access to (with default security configuration).

## 20.5.6 Example 6: Signature label via NLS message

Head back to the gears demo settings page and change the signer configuration to:

```
signatureLabelWithNLS
```

Also, enter the following into the input field "Principal claims":

```
role=Operator
```

Empty the "Signer args" input field if you have entered a value from the previous example.

Now, create another signature with a label by repeating the steps from chapter 20.5.1. The signature label should now look like this:

```
Signed by username (Operator) on 13/02/2025 02:27
```

For this example, some knowledge about NLS (message.properties) files is required (see chapter 17). Now, take a look at the FlowSignerConfiguration bean. This time search for the id "signatureLabelWithNLS":

When values from NLS files are accessed via Spring configuration files (beans and gears.properties files), it should usually be done via ?{}. To learn more about this see chapter 17.6.2.

```
<entry key="documentSigner.args.signatureLabel"
value="?{nlsmg.messages#tutorial.signatureLabel}" />
```

To understand this syntax, read chapter 17.4.

You can find the messages.properties files in the directory cloudsuite.config.shared/i18n (see chapter 21.1).

We use `${'}` instead of `?{'` because we are usually accessing NLS variables during runtime and therefore use gears string expansion (see chapter 17.6.2).

messages.properties files have their own logic for handling curly braces, therefore we must escape them by using: `${'}` and `'}` (see chapter 17.6.1).

```
tutorial.signatureLabel=Signed by ${'principal.user.name'}'  
(${'principal.user.claims.role'}) on ${'config.timestamp:*'}
```

Here we access the principal claim “role”, which we have set earlier to: `role=Operator`.

In this context we are only using gears string expansion. The variable `config.timestamp` also holds a variable. Therefore, we must use recursion (see chapter 20.4.8).



# 21. Appendices

## 21.1 Cheat sheet

### 21.1.1 Windows locations

Variable	Description
cloudsuite.config.name	gears
cloudsuite.config.user	%USERPROFILE%/cloudsuite/config
cloudsuite.config.shared	%ProgramData%/cloudsuite/config
cloudsuite.data.user	%USERPROFILE%/cloudsuite/data
cloudsuite.data.shared	%ProgramData%/cloudsuite/data
cloudsuite.temp.dir	%AppData%/local/temp
cloudsuite.log.dir	%ProgramData%/cloudsuite/log

### 21.1.2 Linux locations

Variable	Description
cloudsuite.config.name	gears
cloudsuite.config.user	<user home>/cloudsuite/config
cloudsuite.config.shared	/etc/cloudsuite
cloudsuite.data.user	<user home>/cloudsuite/data
cloudsuite.data.shared	/var/lib/cloudsuite
cloudsuite.temp.dir	/tmp
cloudsuite.log.dir	/var/log/cloudsuite

### 21.1.3 Property definitions

Property files are read from these directories

- \${cloudsuite.config.shared}
- \${cloudsuite.config.user}

Valid property files are

- \${cloudsuite.config.name}.properties
- \${cloudsuite.config.name}-<profilename>.properties

### 21.1.4 Bean definitions

Bean files are read from these directories

- \${cloudsuite.config.shared}
- \${cloudsuite.config.user}

Valid bean files are

- spring-gears-core.xml
- modules/spring-gears-\*.xml

## 21.1.5 Logging

The internal logback definition can be overridden by placing a logback.xml file at

- `${cloudsuite.config.shared}`
- `${cloudsuite.config.user}`

To only change the directory, use this property

### Spring properties

```
cloudsuite.log.directory=/srv/logs
```

To only change the level, use this property.

### Spring properties

```
cloudsuite.log.level=DEBUG
```

## 21.1.6 Licenses

Licenses are looked up at

- `${cloudsuite.config.shared}/licenses`
- `${cloudsuite.config.user}/licenses`

## 21.1.7 Documentation

The online documentation is available at

**`http://<host>/<gears context>/apidoc/index.html`**

## 21.2 Error stage codes

This is the non-exhaustive list of expected error codes in a reply stage with scheme **`urn:intarsys:names:conversation:1.0:schemes:Error`** or an synchronous **`ResponseError`**

Code	Description
AuthenticationCanceled	An authentication activity was canceled by the user. This is user specific, re-issuing the request might produce another result.
AuthenticationFailed	An authentication activity failed. This is user specific, re-issuing the request might produce another result.
AuthenticationTimeout	An authentication activity has timed out. This is user specific, re-issuing the request might produce another result.
ConversationExpired	The conversation id provided is no longer valid. This is an invalid request, re-issuing the request will raise the same exception.
DocumentSigner	The signature failed. See the error message for more detailed explanation.

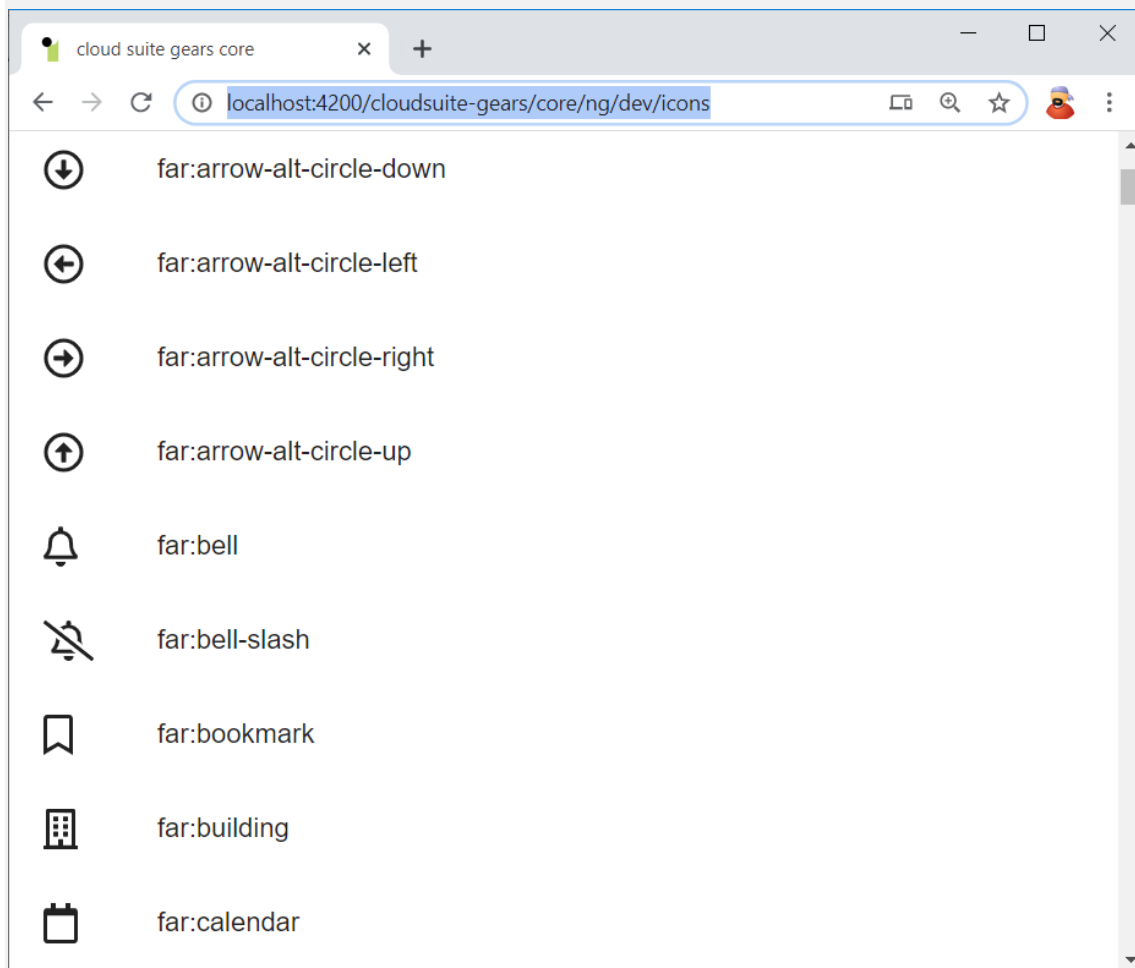
	This may be either an invalid request or a server resource problem, re-issuing the request will most probably raise the same exception.
InternalServerError	An internal server error. This is an internal unexpected error. Re-issuing the request will most probably raise the same exception.
InvalidArgument	Some input value was invalid. This is an invalid request, re-issuing the request will raise the same exception.
InvalidRequest	The request is not valid in the current state. This is an invalid request, re-issuing the request will raise the same exception.
License	The request is not licensed. This is an invalid request, re-issuing the request will raise the same exception. You must install the appropriate licenses first.
ObjectCreation	The request could not create some required dependencies. Most probably this is caused by invalid arguments. This is an invalid request, re-issuing the request will raise the same exception. Future versions may be more concise on this error condition an report a more detailed code.
PDFParse	Parsing a PDF document failed. This is an invalid request, re-issuing the request will raise the same exception.
PDFRender	Rendering a PDF document failed. This is an invalid request, re-issuing the request will raise the same exception.
PoolResourceNotAvailable	Even after waiting for the specified timeout, a security application could not be allocated from the pool. This is a server resource problem. Re-issuing the request to <b>another</b> server might succeed.
PoolStopped	The pools is currently stopped and cannot serve security applications. This is a server resource problem. Re-issuing the request to <b>another</b> server might succeed.
ProcessFailed	An internal process participating in the request processing failed. This is an invalid request, re-issuing the request will raise the same exception.
Retry	The request could not be satisfied for internal reasons (e.g. a defect of a pooled device). You should retry the request. This is a temporary server resource problem, you should re-issue the request. The <b>same server</b> can be used.
SignedDocManipulation	The internal security check after the signature failed. This may indicate some "man in the middle" attempt to manipulate data. This is either a persistent hardware problem or an attack. Re-issuing the request will not succeed.

## 21.3 Available icons

You can get a list of all loaded icons at:

```
<host>/cloudsuite-gears/core/ng/dev/icons
```

This is what you will see:



## 21.4 Proxies

### 21.4.1 Incoming

In many scenarios, "reverse proxies" or other infrastructure devices are installed before the gears server. We refer to all of these devices as "proxy" here for reasons of simplicity.

Here are some important points to remember for these scenarios.

- When the internal connection between the proxy and gears is non-TSL, you should set the "secure" flag in the tomcat connector configuration to indicate the connection is secure anyway. Otherwise Tomcat cannot make informed decisions on certain features (such as Cookie HTTPS only,...). Setting the "scheme" property may be a good idea, too.

- gears recognizes the de-facto standard headers used by proxies to indicate the host name & port. Be sure to set up your proxy correctly to forward these headers.

## 21.4.2 Outgoing

You may have to tweak settings for outgoing connections, too, when dealing with remote signature servers.

- gears does not have any proxy configuration itself, all settings are made via Java VM properties ("http.proxyHost" and "http.proxyPort", respectively, see <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/doc-files/net-properties.html>).
- If you are using an authenticating proxy **and** a TLS channel, you must tweak the Java security settings as this is no longer supported since Java 1.8.111. If you absolutely need this feature, you can start Java with the option

```
-Djdk.http.auth.tunneling.disabledSchemes=""
```

- Some signature servers are bound to a specific client address. If a request from gears is not honored by the remote signature server, be sure to check this restriction.

## 21.5 Spring well known beans

Here's an overview for the "user manageable parts" of the Spring bean definitions. The detailed documentation is in the respective chapters.

conversationRegistry
The de.intarsys.tools.conversation.IConversationRegistry used for conversations
cryptoKeyMaster
An de.intarsys.tools.crypto.api.IByteProvider that provides the master key.
cryptoKdfMaster
An de.intarsys.tools.crypto.api.IKeyDerivationFunction that is used to create the key hierarchy derived from the master key.
dataSource
The javax.sql.DataSource used in gears
flowViewerConfigurationPlain
The FlowViewerConfiguration provided by default in gears.
flowExplorerConfigurationPlain
The FlowExplorerConfiguration provided by default in gears.
flowSignerConfigurationPlain
The FlowSignerConfiguration provided by default in gears.
modelPrincipalDao
The de.intarsys.cloudsuite.gears.model.entity.principal.IPrincipalDao for looking up principals
principalProviderTenant
The de.intarsys.cloudsuite.gears.model.entity.principal.IPrincipalProvider providing the "Tenant"
principalProviderClient
The de.intarsys.cloudsuite.gears.model.entity.principal.IPrincipalProvider providing the "Client"
principalProviderUser
The de.intarsys.cloudsuite.gears.model.entity.principal.IPrincipalProvider providing the "User"
repository
The de.intarsys.cloudsuite.gears.repository.api.IRepository managing the gears document lifecycle
repositoryDao

The de.intarsys.cloudsuite.gears.repository.api.IRepositoryDao implementing physical document storage
securityRealmControlAuthenticationManager
The Spring security authentication manager for the "Control" realm
securityRealmControlAuthenticationFilter
The Spring security filter for the "Control" realm
securityRealmManageAuthenticationManager
The Spring security authentication manager for the "Manage" realm
securityRealmManageAuthenticationFilter
The Spring security filter for the "Manage" realm
securityRealmFlowAuthenticationManager
The Spring security authentication manager for the "Flow" realm
securityRealmFlowAuthenticationFilter
The Spring security filter for the "Flow" realm

## 21.6 Principal roles

urn:intarsys:names:principal:1.0:role:Tenant  
urn:intarsys:names:principal:1.0:role:Client  
urn:intarsys:names:principal:1.0:role:User

## 21.7 Reply stage schemes

urn:intarsys:names:conversation:1.0:schemes:Cancel  
urn:intarsys:names:conversation:1.0:schemes:Error  
urn:intarsys:names:conversation:1.0:schemes:Idle  
urn:intarsys:names:conversation:1.0:schemes:Processing  
urn:intarsys:names:conversation:1.0:schemes:HttpRedirect  
urn:intarsys:names:conversation:1.0:schemes:Result

## 21.8 EncryptionInfo schema

```

EncryptionInfo ::= {
  version: "1.0",
  encryptionAlgorithm: EncryptionAlgorithm,
  recipients: RecipientInfo[]
}

EncryptionAlgorithm ::= SymmetricEncryptionAlgorithm | ...

SymmetricEncryptionAlgorithm ::= {
  type: "urn:intarsys:names:crypto:1.0:algorithm:SymmetricEncryption",
  algorithmName: String,
  iv: bytes
}

RecipientInfo ::= SymmetricRecipientInfo

SymmetricRecipientInfo ::= {
  type: "urn:intarsys:names:crypto:1.0:recipient:SharedKey",
  keyIdentifier: String,
  encryptedKey: byte[],
  encryptionAlgorithm: EncryptionAlgorithm
}

```

### An example meta data structure

```

{
  "encryptionInfo": {
    "version": "1.0",
    "recipients": [{
      "type": "urn:intarsys:names:crypto:1.0:recipient:SharedKey",
      "keyIdentifier": "urn:intarsys:repository:1.0:keyid:Standard",
      "encryptionAlgorithm": {
        "type": "urn:intarsys:names:crypto:1.0:algorithm:SymmetricEncryption",
        "algorithmName": "AES/CTR/NoPadding",
        "iv": "4+RHLtSxxtc1/PtUccX53A=="
      },
      "encryptedKey": "v30eTITcBx287qpRdH0JDQ=="
    }],
    "encryptionAlgorithm": {
      "type": "urn:intarsys:names:crypto:1.0:algorithm:SymmetricEncryption",
      "algorithmName": "AES/CTR/NoPadding",
      "iv": "Z5uL8a3NyCC/zbqUrUigGQ=="
    }
  },
  "type": "d",
  "name": "foo.txt"
}

```

## 22. External References

---

- [1] intarsys GmbH, Sign Live! cloud suite gears cookbook.
- [2] intarsys GmbH, Sign Live! cloud suite gears wp architecture.
- [3] intarsys GmbH, Sign Live! cloud suite gears incubator.
- [4] intarsys GmbH, Sign Live! Security Applications Developers Guide.
- [5] intarsys GmbH, Sign Live! cloud suite gears wp security.
- [6] Swisscom AG, „All-in Signing Service Reference Guide,“ [Online]. Available: [http://documents.swisscom.com/product/1000255-Digital\\_Signing\\_Service/Documents/Reference\\_Guide/Reference\\_Guide-All-in-Signing-Service-de.pdf](http://documents.swisscom.com/product/1000255-Digital_Signing_Service/Documents/Reference_Guide/Reference_Guide-All-in-Signing-Service-de.pdf).
- [7] „CSC API Specifications,“ Cloud Signature Consortium, [Online]. Available: <https://cloudsignatureconsortium.org/resources/download-api-specifications/>.
- [8] ETSI, ETSI EN 319 102-1.
- [9] ETSI, ETSI TS 119 432.
- [10] ETSI, ETSI TR 119 000.
- [11] ETSI, ETSI EN 319 122-1.
- [12] ETSI, ETSI EN 319 132-1.
- [13] ETSI, ETSI EN 319 142-1.
- [14] ISO, ISO 32000-1.