

Sign Live! cloud suite validator manual

September 2025

intarsys GmbH

Sign Live! cloud suite validator manual

Version 8.14

cloud suite validator architecture, design & reference

intarsys GmbH
Sign Live! cloud suite validator manual
Version 8.14

All rights reserved
© 2020 intarsys GmbH
www.intarsys.de

Preface

- Author and company

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

- Trademarks

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

jPod is a trademark of intarsys consulting.

Sun, Java and JavaScript are trademarks of Oracle

Microsoft and Windows are trademarks of Microsoft Corporation.

- Who should read this book

This book 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.

E-Mail support@intarsys.de

Website www.intarsys.de

■ Disclaimer

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

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

Contents

Preface	5
▪ Author and company	5
▪ Trademarks	5
▪ Who should read this book	5
▪ Reviews and comments	5
▪ Disclaimer	6
Contents	7
Introduction	14
1. Overview	15
2. Building blocks	16
2.1 Services	16
2.1.1 Meta	16
2.1.2 Validator	16
2.1.3 ConformityChecker	16
2.1.4 ReportCreator	16
2.1.5 ValidationPipeline	16
2.1.6 Augmentor	16
2.2 Admin application	16
3. Installation	17
3.1 Overview	17
3.2 System requirements	17
3.2.1 Server	17
3.2.2 Client	19
3.2.3 3 rd party	19
3.3 Documentation	19
3.4 Installation Process	20
3.4.1 Server	20
3.5 Web Apps	20
3.5.1 Web App "validator"	20

Content

3.5.2	Web App "validator-admin"	20
3.6	SDK	20
3.7	Miscellaneous	21
3.7.1	CWT Rendering library	21
3.8	Version check web app	21
3.8.1	In the browser log	21
3.9	Version check server	21
3.9.1	In the WEB-INF directory	22
3.9.2	In the log	22
3.9.3	On the client	22
4.	Design details	24
4.1	Flow	24
4.2	Conversation	24
4.2.1	Reply stage	26
4.3	Document	26
4.3.1	Document Properties	26
4.4	Repository	27
4.5	Services	27
5.	Crypto components	29
5.1	Basics	29
5.1.1	IByteProvider & IByteStreamProvider	29
5.1.2	IKeyDerivationFunction	29
5.1.3	ICipherFactory	30
5.1.4	Key tree	31
5.2	Cryptdec	31
5.2.1	Plain	32
5.3	Secret	32
5.3.1	Syntax	32
5.3.2	Configuration	32
5.4	Repository encryption	33
6.	Configuration	35
6.1	Overview	35
6.1.1	Configuration location	35
6.1.2	Built-in configuration	37
6.1.3	Custom property definition	37
6.1.4	Custom bean definition	37
6.1.5	Using profiles	38
6.1.6	String expansion integration	38
6.2	Web application root	39
6.3	Logging	39
6.3.1	Override logging	39
6.3.2	Builtin logging	40
6.3.3	Log correlation	40
6.3.4	Log tweaks	40

Content

6.4	Licenses	41
6.5	Data source	42
6.6	Conversation registry	42
6.7	Basic security	43
6.7.1	Master key	43
6.7.2	Master KDF	44
6.7.3	Application "cryptdec"	44
6.8	Repository	44
6.8.1	Basic settings	44
6.8.2	Encryption	45
6.9	PDF security environment	46
6.10	Principals	47
6.10.1	Overview	47
6.10.2	Principal model	48
6.10.3	Roles	51
6.10.4	Provider	52
6.10.5	Default configuration	53
6.11	Signature environment	54
6.11.1	Timestamp creation	54
6.11.2	Signature validation	55
6.11.3	Trusted list management	55
7.	Authentication	56
7.1	Overview	56
7.2	Opt-out	56
7.3	Concepts	56
7.4	Security realms	58
7.4.1	Overview	58
7.4.2	Common properties	58
7.4.3	Security realm "Flow"	58
7.4.4	Security realm "Control"	58
7.4.5	Security realm "Manage"	59
7.5	Logout	59
7.6	Authentication filter	59
7.6.1	Static authentication	60
7.6.2	Basic authentication	60
7.6.3	OAuth2 authentication	61
7.6.4	X.509 authentication	61
7.7	Authentication manager	61
7.7.1	In memory repository	62
7.7.2	JDBC repository	63
7.8	Principal integration	63
7.8.1	Dao based principal lookup	63
7.8.2	JWT based principal lookup	64
7.9	Well known security beans	64
8.	Authorization	66

Content

8.1	Overview	66
8.2	Concepts	66
8.2.1	Authentication	67
8.2.2	Authority	67
8.2.3	Resource	67
8.2.4	Operation	67
8.2.5	Authorization strategy	67
8.3	Integration (observation)	68
8.4	ACL Authorization	69
8.4.1	Strategy	69
8.4.2	Configuration	70
8.4.3	Resources	72
9.	Services	77
9.1	Overview	77
9.2	API	77
9.3	Protocol	77
9.3.1	Flow creation	77
9.3.2	Conversational response	78
9.3.3	"Final" stages	78
9.3.4	Multipage in-band	80
9.3.5	Redirect URI	82
9.3.6	Multipage out-of-band	82
9.3.7	Summary	83
9.4	Serialization issues	85
9.4.1	Scaling in a Java environment	85
9.4.2	Confidentiality	86
9.4.3	Property order	87
9.5	Error handling	87
9.5.1	Overview	87
9.5.2	ErrorDetail object	87
9.5.3	ResponseError object	87
9.5.4	Synchronous error handling	88
9.5.5	Conversational error handling	88
9.6	Request options	89
9.6.1	Redirect URI	89
9.6.2	Restricted identification	89
9.6.3	Principal	90
9.6.4	Language	90
10.	Integration	92
10.1	Overview	92
10.2	Model	92
10.2.1	Observation	92
10.2.2	Observer	93
10.2.3	Filter	94
10.2.4	View	94

10.3	Implementation	94
10.4	Observer definition	95
10.5	Filter definition	96
10.5.1	Accept	97
10.5.2	Reject	97
10.6	View definition	97
10.6.1	Include	98
10.6.2	Exclude	98
10.6.3	Add property	98
10.6.4	Complete observer example	99
10.7	Appender definition	100
10.7.1	Plain logging	100
10.7.2	Pattern conversion for args	101
10.7.3	Other appenders	102
10.7.4	Logstash support	102
10.7.5	Webhook	103
10.8	Example	104
10.9	Observations	104
10.9.1	Overview	104
10.9.2	application	104
10.9.3	license	104
10.10	Webhook	105
10.10.1	Overview	105
10.10.2	Webhook stub	106
10.10.3	Execution semantics	107
10.10.4	SSL/TLS	108
10.10.5	Authentication	109
11.	Monitoring	111
11.1	Overview	111
11.2	JMX	111
11.2.1	Installation	111
11.2.2	Configuration	111
11.2.3	Client	111
11.2.4	MBean Deployment	113
11.2.5	gears MBeans	113
11.3	Spring actuator	113
11.3.1	Installation	114
11.3.2	Configuration	114
11.3.3	Endpoints	114
12.	Reference	119
12.1	Data models	119
12.1.1	Common models	119
12.2	Principal related data models	124
12.2.1	GenericClaim	124
12.2.2	GenericPrincipal	125

Content

12.2.3	GenericUser	125
12.2.4	JdbcPrincipalDao	126
12.2.5	JdbcUserDao	127
12.2.6	PojoPrincipalDao	127
12.2.7	PojoUserDao	128
12.2.8	ExplicitPrincipalProvider	129
12.2.9	StaticPrincipalProvider	129
12.2.10	SpringSecurityPrincipalProvider	130
12.3	Crypto related data models	130
12.3.1	ISslContextProvider	130
12.3.2	IByteProvider	132
12.3.3	ICipherFactory	133
12.3.4	IKeyDerivationFunction	135
13.	Implementation hints	136
13.1	Chunked transfer	136
14.	String expansion	137
14.1	Basics	137
14.1.1	Why string expansion	137
14.1.2	Terminology	137
14.1.3	Syntax	138
14.1.4	Constant text in expression	138
14.1.5	Namespaces	138
14.1.6	Spring integration	139
14.2	Namespaces	140
14.2.1	Overview	140
14.2.2	app	140
14.2.3	config	140
14.2.4	counters	141
14.2.5	digestSigner	142
14.2.6	entity	143
14.2.7	environment	144
14.2.8	flow	144
14.2.9	identifiers	145
14.2.10	nlsmg	146
14.2.11	properties	146
14.2.12	system	147
14.2.13	time	147
14.3	Formatting	148
14.3.1	Overview	148
14.3.2	String formatting	149
14.3.3	Integer formatting	150
14.3.4	Date formatting	151
14.3.5	Float formatting	152
14.3.6	File path formatting	153
14.3.7	Default value	153

Content

14.3.8	Recursion	154
14.3.9	Conditional evaluation	155
15.	Appendices	157
15.1	Cheat sheet	157
15.1.1	Windows locations	157
15.1.2	Linux locations	157
15.1.3	Property definitions	157
15.1.4	Bean definitions	157
15.1.5	Logging	158
15.1.6	Licenses	158
15.1.7	Documentation	158
15.2	Error stage codes	158
15.3	Proxies	160
15.3.1	Incoming	160
15.3.2	Outgoing	161
15.4	Spring well known beans	161
15.5	Principal roles	162
15.6	Reply stage schemes	162
15.7	EncryptionInfo schema	163
16.	External References	164

Introduction

Sign Live! cloud suite validator 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 validator, along with a detailed reference of how to configure and use the components and services.

1. Overview

Sign Live! cloud suite validator 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.

Sign Live! cloud suite validator is sharing concepts and patterns with Sign Live! Cloud suite gears and is based on the same architecture. Accordingly, the terms “validator”, “gears validator” and “gears” are synonymously used in this document.

2. Building blocks

2.1 Services

2.1.1 Meta

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

2.1.2 Validator

The **validator** service provides basic document validation.

2.1.3 ConformityChecker

The **conformitychecker** allows the application of custom rules on top of the basic validation.

2.1.4 ReportCreator

The **reportcreator** allows rendering machine or human readable reports based on the basic and custom validation output.

2.1.5 ValidationPipeline

The **validationpipeline** combines augmentation, basic validation, conformity checking, and report creation into a single process.

2.1.6 Augmentor

The Augmentor allows the augmentation of signatures.

2.2 Admin application

An administrative application to manage and control information that is used in the validator runtime.

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

You can for example configure trust anchors, report templates and rules.

3. Installation

3.1 Overview

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

- doc
- SDK
- webapps

3.2 System requirements

3.2.1 Server

3.2.1.1 Operating system

The server components are tested and released for

- Ubuntu Linux 22.04 / 24.04
- Red Hat Enterprise Linux 9.1
- Microsoft Windows 2016 / 2019 / 2022 / 2025 server
- Cloud Foundry with Java Buildpack v4.49 or newer

3.2.1.2 Microsoft Windows requirements

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

- Visual Studio 2013 runtime libraries

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

More information on this topic is available in 3.7.1, Rendering library.

3.2.1.3 System software

3.2.1.3.1 **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 17.44+53.

3.2.1.3.2 **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).

3.2.1.3.3 Reverse proxy

In complex installations you should think about installing Sign Live! cloud suite validator 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.

3.2.1.4 Hardware

3.2.1.4.1 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 configuration (e.g. encrypted file repository, using server-side rendering) you may want to increase server power.

3.2.1.4.2 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 configuration (e.g. encrypted file repository, using server-side rendering) you may want to increase available memory.

3.2.1.4.3 Disk storage

Beside the software installation requirements

- ~ 1GB (Tomcat, Java, Sign Live! cloud suite validator 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.

3.2.1.4.4 Peripherals

The system components do not require special peripherals.

3.2.2 Client

3.2.2.1 Service clients

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

3.2.2.2 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 > 137 (Windows 10/11, Android tablet)
- Firefox > 139 (Windows 10/11)
- Edge Chromium > 137 (Windows 10)
- Safari > 16 (MacOS, iPad)

3.2.3 3rd party

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

3.2.3.1 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 2.3
- PostgreSQL 12.15
- MariaDB 10.5

3.2.3.2 Remote service access

The validation processes require access to remote services provided by Trust Service Providers. These services comprise OCSP responders, which typically are accessible via HTTP/HTTPS on ports 80/443 and CRLs, which can be obtained via HTTP/HTTPS on ports 80/443 or LDAP on port 389.

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

3.3 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 validator context>/apidoc/index.html`

3.4 Installation Process

3.4.1 Server

Perform the following installation steps:

1. Install Java Runtime
2. Install Tomcat
3. Deploy the "cloudsuite-gears#validator.war" into the Tomcat webapps directory. For further details read chapter 3.5.
4. Open the following URL in the browser
`http://<host>:8080/cloudsuite-gears/validator`

3.5 Web Apps

3.5.1 Web App "validator"

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

The servlet container used for testing is Tomcat 10.0.x.

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

The validator 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.

3.5.2 Web App "validator-admin"

This web application provides administrative access to validation resources. See [1] for details on this application.

3.6 SDK

The SDK 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.

3.7 Miscellaneous

3.7.1 CWT Rendering library

The CWT rendering backend 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 validator platform code.

This can fail now for a variety of reason, 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 2013 runtimes for dynamic linking are not installed, thus the provided freetype.dll cannot resolve its dependencies
→ Ensure the Visual Studio 2013 runtime library is installed. You can do this simply by installing Sign Live CC!

3.8 Version check web app

3.8.1 In the browser log

All Sign Live! cloud suite validator 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
```

3.9 Version check server

3.9.1 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
```

3.9.2 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

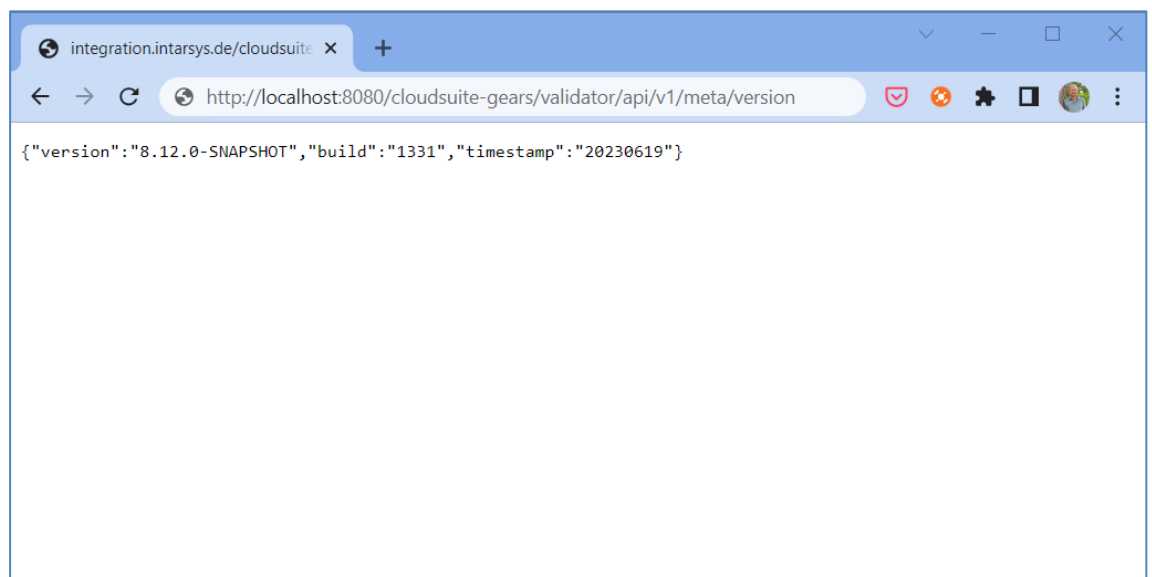
```
[23.04.2018-14:19:02.442][I][d.i.spring.tools      ][localhost-startStop-1][ version info:
[23.04.2018-14:19:02.442][I][d.i.spring.tools      ][localhost-startStop-1][ intarsys-
cloudsuite-gears-validator-backend (intarsys-cloudsuite-gears-validator-backend-
8.0.0.jar), 8.0.0, local, Mon Apr 23 14:15:54 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
```

3.9.3 On the client

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

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

This will return version information on the currently installed services.



4. Design details

4.1 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.

4.2 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 validator 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.

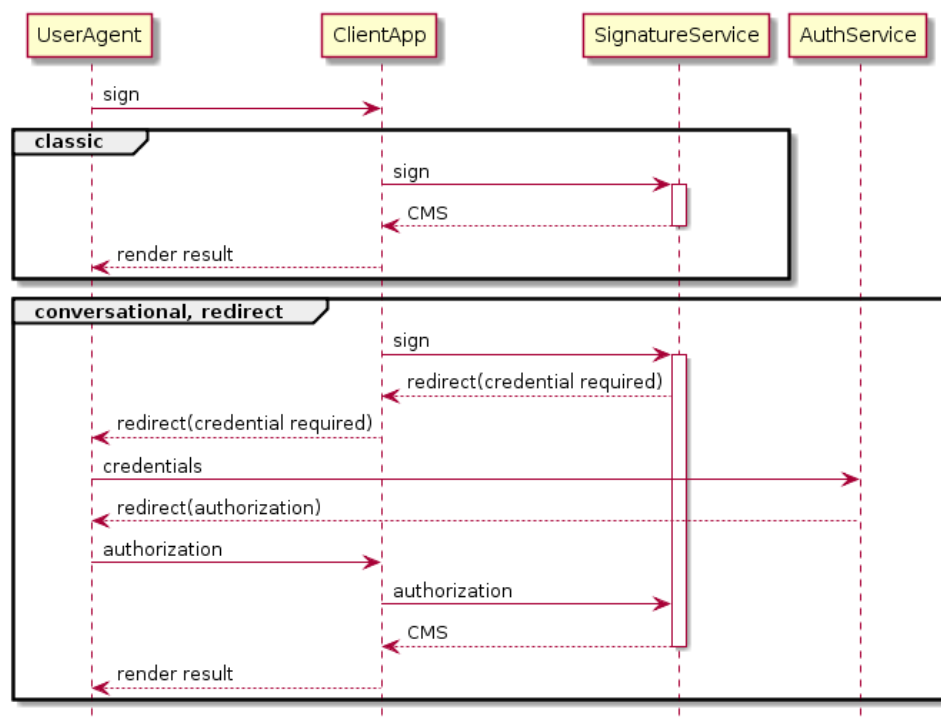
So, the server still receives the request and the hash, but now decides on its own if and how to authenticate and authorize the user. Most important, he may send a request to enter a PIN, OTP or any other kind of credential. This is a "sub interaction" to the client initiated one. The client has to take 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 validator 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.

4.2.1 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.

4.3 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.

4.3.1 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.

4.4 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 "**Fehler! Verweisquelle konnte nicht gefunden werden.**" 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.

4.5 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/validator/api/v1/flow/validator/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. 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.validator.service.validator.api.ResultValidator",
        "value": {
          "documentNames": [ "foo.txt" ],
          "report": {
            {
              "type": "d",
              "name": "mydoc.report.xml",
              "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 **ResultValidator** holding the validation report for the requested document(s).

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

5. Crypto components

5.1 Basics

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

5.1.1 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.

5.1.2 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.

5.1.3 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

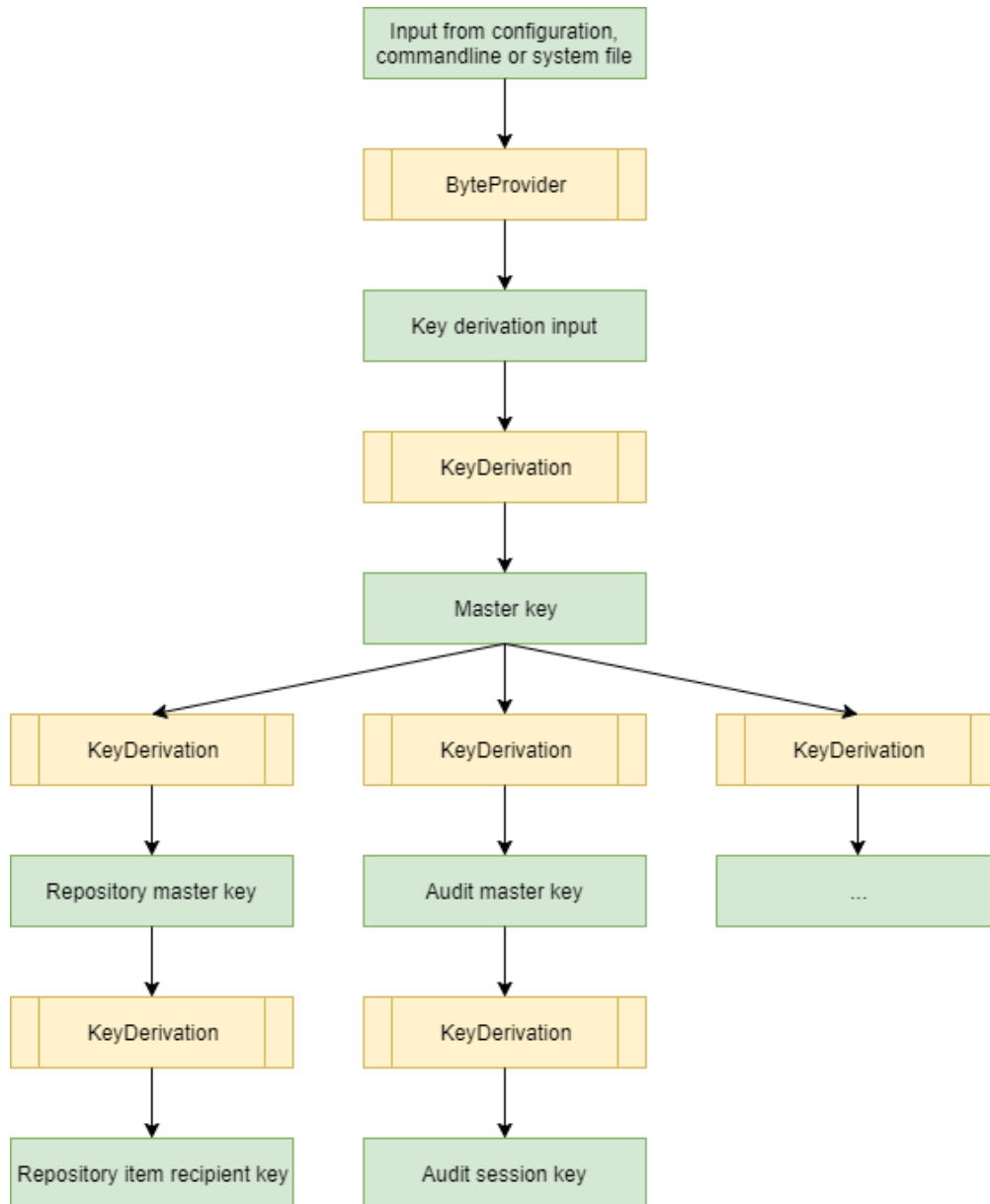
Takes the secret and feeds it into a KDF before forwarding it to another ICipherFactory

de.intarsys.tools.crypto.standard.StaticKeyCipherFactory

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

5.1.4 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.

5.2 Cryptdec

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

5.2.1 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
```

5.3 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.

5.3.1 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.

5.3.2 Configuration

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

5.3.2.1 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.

5.3.2.2 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.

5.3.2.3 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.

5.4 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.

6. Configuration

6.1 Overview

Sign Live! cloud suite validator 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.

6.1.1 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.

6.1.1.1 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

6.1.1.2 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.

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
```

6.1.2 Built-in configuration

Sign Live! cloud suite validator 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.

6.1.3 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

6.1.4 Custom bean definition

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

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

6.1.5 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

6.1.6 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".

6.2 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 its 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
```

6.3 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.

6.3.1 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.

6.3.2 Builtin 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

6.3.3 Log correlation

gears tries its best to allow the correlation of log messages. For this purpose a special log context property "corr" is provided whenever applicable. You can add this information to your log pattern using

```
%X{corr}
```

6.3.4 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
```

6.3.4.1 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.

6.3.4.2 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
```

6.3.4.3 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
```

6.3.4.4 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
```

6.4 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
```

6.5 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>
```

6.6 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="10000"/>
</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.

6.7 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.

6.7.1 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.

6.7.2 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.

6.7.3 Application "cryptdec"

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

6.8 Repository

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

6.8.1 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.

6.8.2 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>
...
```

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>
    <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>
</bean>
```

- 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.

6.9 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>
```

6.10 Principals

6.10.1 Overview

Sign Live! cloud suite validator 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

6.10.2 Principal model

6.10.2.1 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.

6.10.2.2 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
```

6.10.2.3 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>

```

6.10.2.4 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.

6.10.2.5 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.

6.10.2.6 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

6.10.3 Roles

6.10.3.1 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
```

6.10.3.2 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
```

6.10.3.3 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
```

6.10.3.4 Other roles

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

6.10.4 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.

6.10.4.1 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>
```

6.10.4.2 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>
```

6.10.4.3 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>
```

6.10.5 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.

6.11 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.

6.11.1 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

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

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

7. Authentication

7.1 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!

7.2 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 basic”.

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.

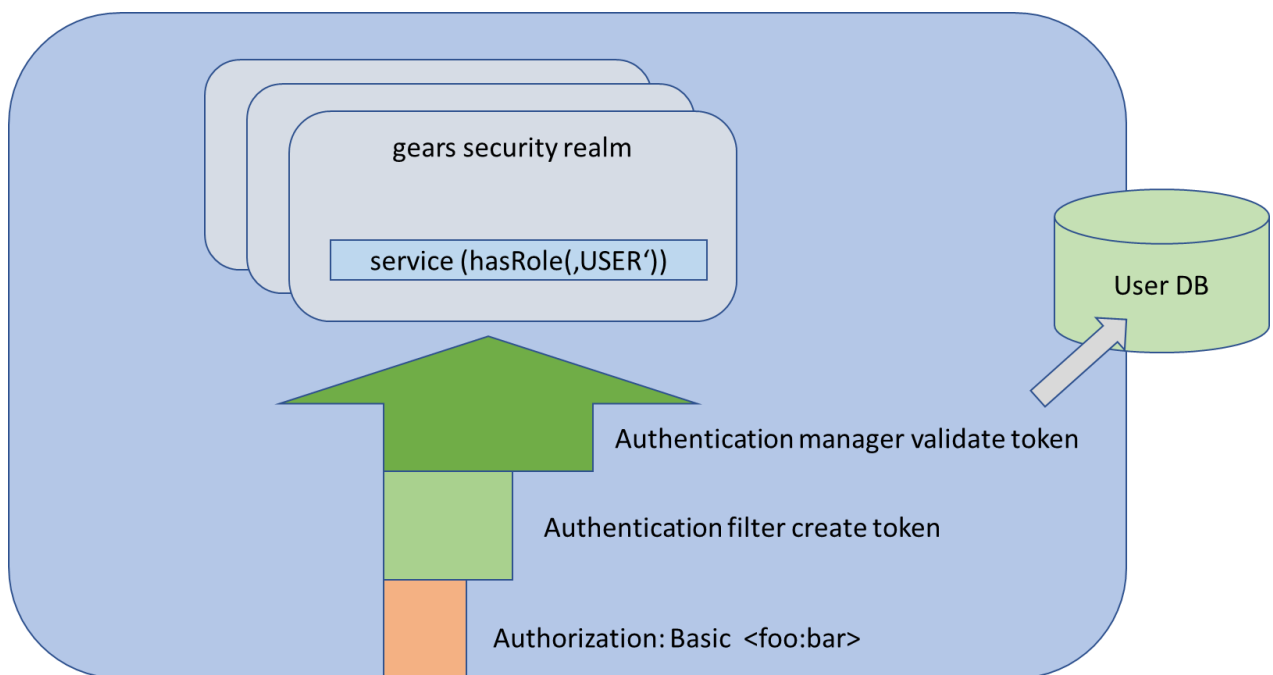
Now you can (and should) construct a spring security configuration of your own.

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”.

7.3 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.

7.4 Security realms

7.4.1 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

7.4.2 Common properties

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

7.4.3 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/validator/**
- /api/v1/flow/conformitychecker/**
- /api/v1/flow/reportcreator/**
- /api/v1/flow/validationpipeline/**

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

The realm bean is "securityRealmFlow".

7.4.4 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".

7.4.5 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".

7.5 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 thrown away.

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.

7.6 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

7.6.1 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, ...).

7.6.2 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 [2].

7.6.3 OAuth2 authentication

Spring has broad support for OAuth2 token support.

Detailed scenarios using OAuth2 are described in [2].

7.6.4 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 [2]. 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 [2].

7.7 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 get up and running fast.

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 build 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).

7.7.1 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 "demo/config" folder of the gears product bundle.

7.7.2 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 "demo/config" folder of the gears product bundle.

7.8 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.

7.8.1 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>
```

7.8.2 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.

7.9 Well known security beans

Just in case you feel tempted to tweak the builtin 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 security:authentication-manager definition for the “ control ” realm
securityRealmManageAuthenticationManager	A spring security:authentication-manager definition for the “ manage ” realm
securityRealmFlowAuthenticationManager	A spring security:authentication-manager definition for the “ flow ” realm
securityRealmControlAuthenticationFilter	A servlet filter that provides the request authentication information for “ control ” realm
securityRealmManageAuthenticationFilter	A servlet filter that provides the request authentication information for “ manage ” realm
securityRealmFlowAuthenticationFilter	A servlet filter that provides the request authentication information for “ flow ” realm

8. Authorization

8.1 Overview

gears validator 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

8.2 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.

8.2.1 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.

8.2.2 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.

8.2.3 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.

8.2.4 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.

8.2.5 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

8.3 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 und 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>

```

8.4 ACL Authorization

8.4.1 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 gears 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.

8.4.2 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 if 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>
```

8.4.3 Resources

8.4.3.1 Device

8.4.3.1.1 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:*
```

8.4.3.1.2 Operations

For the device you can control the “sign” operation, Access is checked immediately before the signing takes place.

```
sign
```

8.4.3.1.3 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>
```

8.4.3.2 Service Arguments

8.4.3.2.1 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/validator/create

There are four services available under access control

- /v1/flow/validator/create
- /v1/flow/conformitychecker/create
- /v1/flow/reportcreator/create
- /v1/flow/validationpipeline/create

So, a correctly qualified resource reads like

Service:/v1/flow/validator/create

Remember that you can use a wildcard instead of the id to address all resources.

Service:*

8.4.3.2.2 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

8.4.3.2.3 Examples

These are some examples for service ACL configuration in spring

Example: Allow access to use args for signer creation for the authority ROLE_VALIDATE.

```
<bean class="de.intarsys.aaa.authorization.acl.Grant">
  <property name="authority" value="ROLE_VALIDATE"/>
  <property name="resource" value="Service:/v1/flow/validator/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>
```

8.4.3.3 Configuration

8.4.3.3.1 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!

8.4.3.3.2 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
```

8.4.3.3.3 Examples

These are some examples for configuration ACL configuration 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>
```


9. Services

9.1 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".

Correspondingly, the swagger UI is available at "swagger.html".

The corresponding JSON and YAML descriptions are available at

`http://<host>/<gears context>/api/swagger.[json|yaml]`

9.2 API

The services are documented in OpenAPI and can be inspected using the swagger UI.

This is the authoritative description and guide.

9.3 Protocol

This chapter gives detailed advice how to handle the conversational protocol in your client.

9.3.1 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.

9.3.2 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

9.3.3 "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
    }
  }
}
```

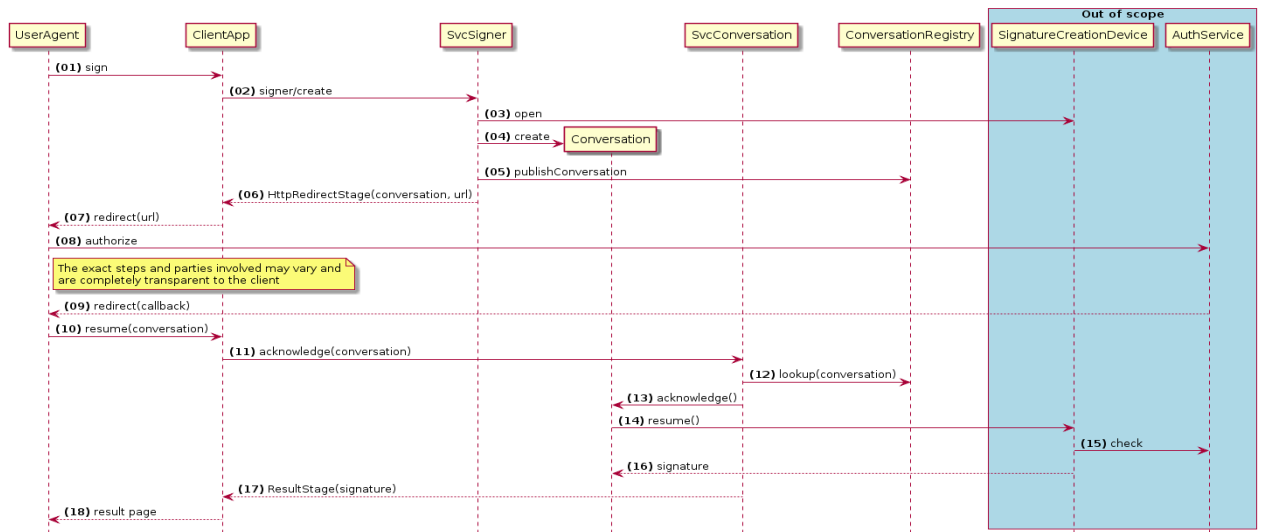
9.3.4 Multipage in-band

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 3rd 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)

9.3.5 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)

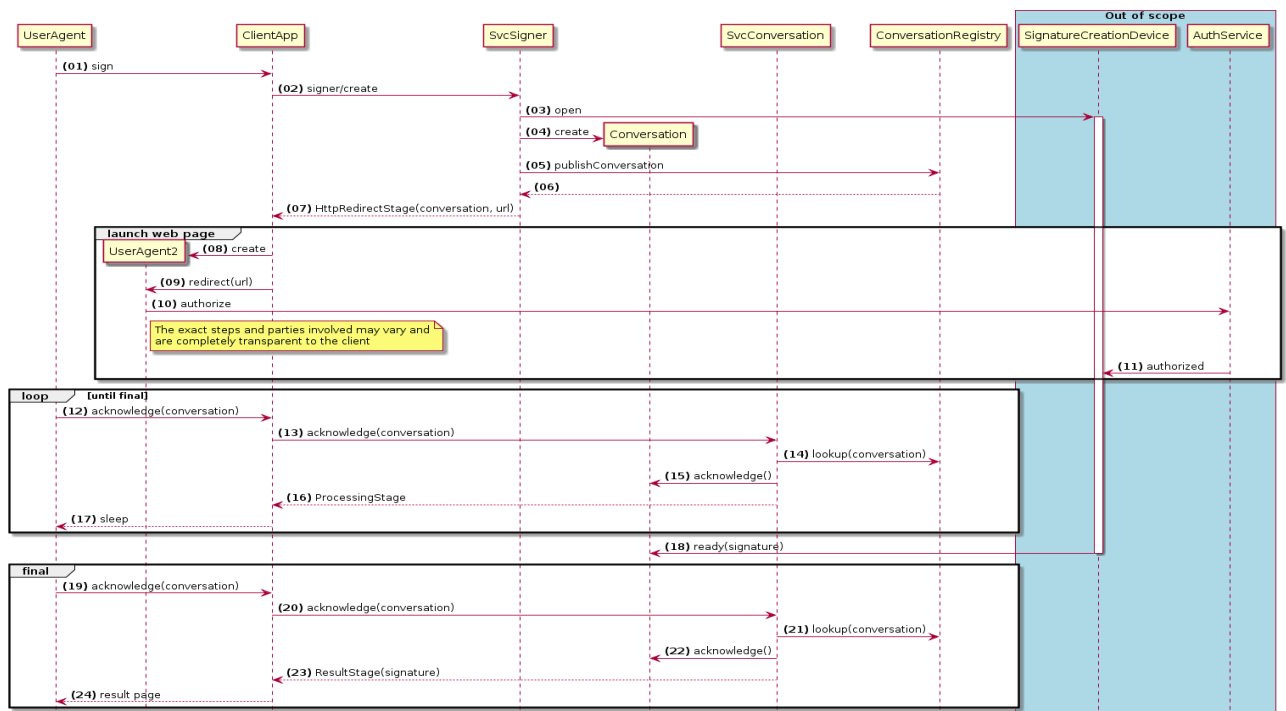
service call

```
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>"
  },
  ...
]
```

9.3.6 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 a 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).

9.3.7 Summary

So, lets 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...
 - HttpRedirect 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)
 - HttpRedirect 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.

9.4 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.

9.4.1 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.

9.4.2 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.

9.4.3 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.

9.5 Error handling

9.5.1 Overview

A list of expected error codes can be found in the appendices.

9.5.2 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",
}
```

9.5.3 ResponseError object

ResponseError

The response object in case of synchronous error conditions.

Properties

<code>_error</code>	
ErrorDetail	An ErrorDetail object describing the error for this request

9.5.4 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",
  }
}
```

9.5.5 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"
  }
}
```

9.6 Request options

Request options are generic contextual information to the request, that are not directly related to the business function that is executed.

9.6.1 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
```

9.6.2 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"
  },
  ...
}
```

9.6.3 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"
      }
    }
  },
  ...
}
```

9.6.4 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"
  },
  ...
}
```

10. Integration

10.1 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.

10.2 Model

10.2.1 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 3rd 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

10.2.2 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

10.2.3 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.

10.2.4 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.

10.3 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.

10.4 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 or 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"> • "FILE" Use the appender "FILE" attached to the root logger • "foobar/AUDIT" Use the appender "AUDIT" attached to the logger "foobar"

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>
</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>
```

10.5 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.

10.5.1 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.

10.5.2 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.

10.6 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.

10.6.1 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").

10.6.2 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.

10.6.3 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"

10.6.4 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>

```

10.7 Appender definition

10.7.1 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}
```

10.7.2 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.

10.7.2.1 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]
```

10.7.2.2 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
```

10.7.3 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.

10.7.4 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":
"K31yVTU2dnByRk1KSE41T0VHL2taTk0vbXhhRUM3Uk1GaWhFNGx5QU83dz0="
      }
    }
  ]
}
```

10.7.5 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.

10.8 Example

There is an example configuration in the gears "example" folder (gears observations) that defines the above-mentioned observation primitives.

10.9 Observations

10.9.1 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.

10.9.2 application

source	application
	Observations for the application process
code	started
	This is issued when the application is up and running
-	
	no other properties provided

10.9.3 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.

10.10 Webhook

10.10.1 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 3rd 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.

10.10.2 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.

10.10.2.1 Type

The bean type is

```
de.intarsys.tools.webhook.impl.StandardWebhookStub
```

10.10.2.2 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
readTimeout	

integer	<p>The maximum time in milliseconds to wait for data from the webhook server to arrive.</p> <p>-1 means no change to the system default 0 means infinity</p> <p>Default: -1</p>
followRedirects	
boolean	<p>Flag if the caller follows redirects returned by the webhook server.</p> <p>Default: false</p>
method	
string	<p>The HTTP method to be used for contacting the webhook server.</p> <p>Must be one of GET POST</p> <p>Default: GET</p>
swallowExceptions	
Boolean	<p>A flag if exceptions are swallowed or propagated to the calling code.</p> <p>Default: true</p>
sslContextProvider	
ISslContextProvider	<p>A factory object for the SSL context to be used for webhook connections.</p> <p>Default: The global gears SSL context provider.</p>

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>
```

10.10.3 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

10.10.3.1 Webhook arguments

The arguments to a webhook execution are collected in an event object. The properties of this object depend on the execution context.

10.10.3.2 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
```

10.10.3.3 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" }
```

10.10.4 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>
```

10.10.5 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).

10.10.5.1 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 Z251OmdyYXQ=
```

Example POST

```
POST http://localhost:56161/test/webhook/post
Authorization: Basic Z251OmdyYXQ=
Content-Type: application/json

{"foo":"bar"}
```

10.10.5.2 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"}
```

10.10.5.3 TLS client authentication

You can use the SSL/TLS configuration described in the chapter before to set up an authenticating client.

11. Monitoring

11.1 Overview

"Monitoring" is the process of publishing internal state to external processes.

There are numerous tools and standards to support this process. Two of the most prevalent standards, "Java Management Extensions" (JMX) and "Spring Actuator" (V 2.1.x) are built-in in gears.

11.2 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 3rd party documentation like

<https://www.oracle.com/technetwork/articles/javase/jmx-138825.html>

for more details.

11.2.1 Installation

There is no installation step required for JMX support.

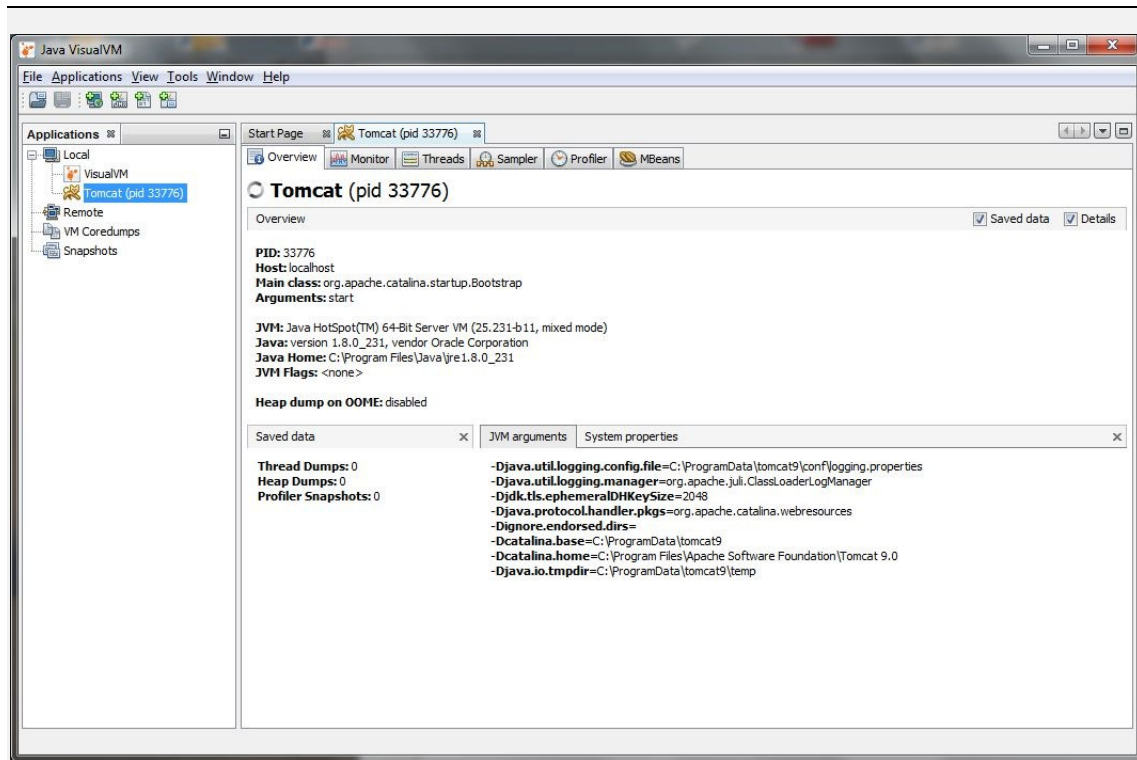
11.2.2 Configuration

There is no configuration step required for JMX support

11.2.3 Client

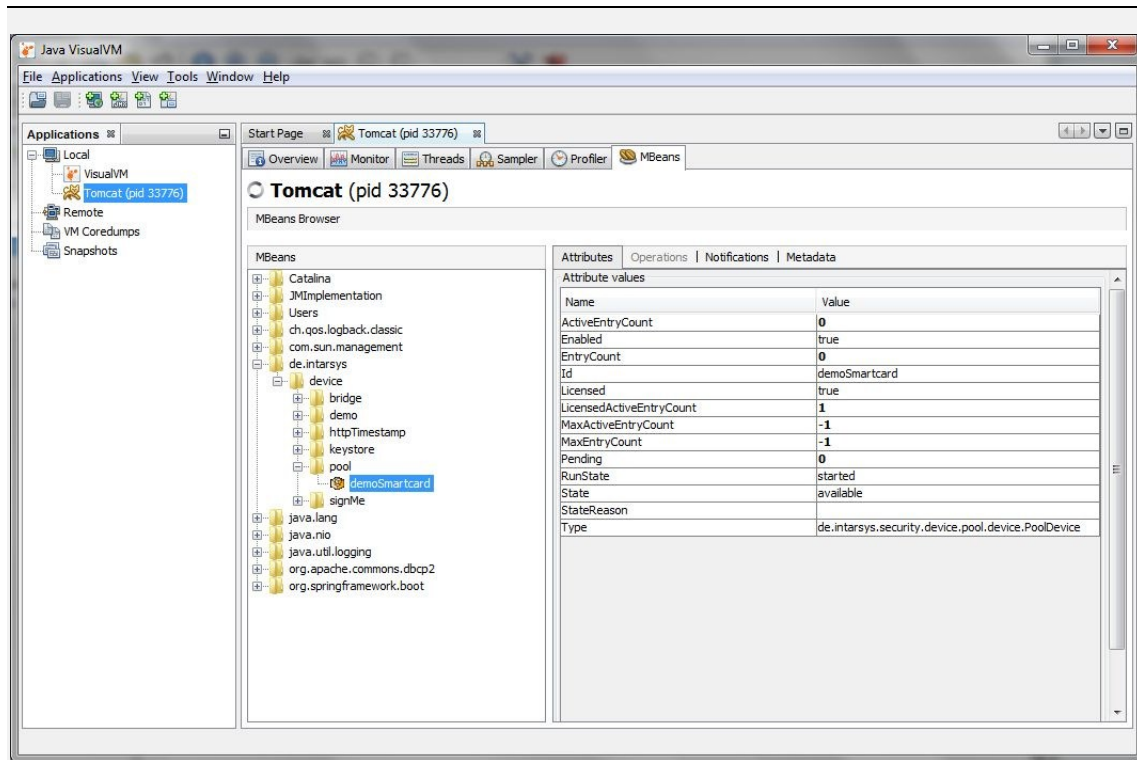
There are numerous client products for dealing with JMX, both open source and commercial.

For a quick look at the JMX server state you can leverage the standard JDK tool "jvisualvm". 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.



11.2.4 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.

11.2.5 gears MBeans

11.2.5.1 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.

11.3 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/>.

11.3.1 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/*`

11.3.2 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.

11.3.3 Endpoints

11.3.3.1 info

Request

`http://<host>/<gears context>/manage/info`

Response (with the **demo** profile activated)

```
{  
  "gears": {  
    "message": "Sign Live! cloudsuite gears"  
  }  
}
```

11.3.3.2 health

The health endpoint provides application status information. Besides the standard Spring boot health indicators, validator 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": {
    "db": {
      "status": "UP",
      "details": {
        "database": "H2",
        "validationQuery": "isValid()"
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 1021821579264,
        "free": 753991966720,
        "threshold": 10485760,
        "path": "C:\\repositories\\is-app-gears-validator\\intarsys-cloudsuite-gears-validator-test\\.",
        "exists": true
      }
    },
    "licensing": {
      "status": "UP",
      "details": {
        "licenses": [
          {
            "status": "UP",
            "details": {
              "properties": [
                {
                  "name": "id",
                  "value": "de.intarsys.cloudsuite.product.gears.validator"
                },
                {
                  "name": "bundle",
                  "value": "professional"
                },
                {
                  "name": "validate",
                  "value": "true"
                },
                {
                  "name": "account_validate",
                  "value": "100000:year"
                },
                {
                  "name": "embed",
                  "value": "true"
                },
                {
                  "name": "account_embed",
                  "value": "100000:year"
                }
              ],
              "owner": "intarsys.de",
              "valid": true,
              "productId": "de.intarsys.cloudsuite.product.gears.validator",
              "validTo": "10/14/2025",
              "validFrom": "07/16/2025",
              "productVersion": "8"
            }
          }
        ],
        "products": [
          {
            "status": "UP",
            "details": {
              "state": "valid",
              "id": "de.intarsys.cloudsuite.product.gears.validator",
              "version": "8.15.0-SNAPSHOT",
              "label": "Sign Live! cloud suite gears validator",
              "accounts": [
                ],
              "main": true,
              "licenses": [
                {

```



```

        "properties": [
          {
            "name": "id",
            "value": "de.intarsys.cloudsuite.product.gears.validator"
          },
          {
            "name": "bundle",
            "value": "professional"
          },
          {
            "name": "validate",
            "value": "true"
          },
          {
            "name": "account validate",
            "value": "100000:year"
          },
          {
            "name": "embed",
            "value": "true"
          },
          {
            "name": "account_embed",
            "value": "100000:year"
          }
        ],
        "owner": "intarsys.de",
        "valid": true,
        "productId": "de.intarsys.cloudsuite.product.gears.validator",
        "validTo": "10/14/2025",
        "validFrom": "07/16/2025",
        "productVersion": "8"
      }
    ]
  },
  {
    "status": "DOWN",
    "details": {
      "state": "na",
      "id": "de.intarsys.license.global",
      "version": null,
      "label": "All products",
      "accounts": [
        {
          "property": "account_validate",
          "limit": 100000,
          "label": "account_validate",
          "unit": "year",
          "balance": 99991,
          "nextReset": "2026-07-01T00:00:00+02:00[Europe/Berlin]",
          "product": "de.intarsys.license.global"
        },
        {
          "property": "account_embed",
          "limit": 100000,
          "label": "account_embed",
          "unit": "year",
          "balance": 100000,
          "nextReset": null,
          "product": "de.intarsys.license.global"
        }
      ],
      "main": false,
      "licenses": [
        {
          "status": "DOWN",
          "details": {
            "state": "na",
            "id": "de.intarsys.license.unavailable",
            "version": null,
            "label": "Unregistered product",
            "accounts": [

```

```
        ],
        "main": false,
        "licenses": [
            ]
        }
    ]
}
},
"ping": {
    "status": "UP"
},
"ssl": {
    "status": "UP",
    "details": {
        "validChains": [
            ],
        "invalidChains": [
            ]
        }
    }
}
}
```

12. Reference

12.1 Data models

12.1.1 Common models

12.1.1.1 Configuration

The configuration allows the definition of a flow execution context as a template.

A configuration can be provided

- in a spring XML configuration file
- in a flow create request
- depending on the system environment, a configuration can be resolved from an arbitrary provider (e.g., a database).

The semantics of a configuration may depend on the concrete flow where it is used.

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 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	

12.1.1.1.1 spring XML examples

spring configuration example, variables

spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.validator.service.validator.impl.FlowValidatorConfig
uration">
  <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

spring XML fragment

```
<bean
class="de.intarsys.cloudsuite.gears.validator.service.validator.impl.FlowValidatorConfig
uration">
  <property name="id" value="myConfig"/>
  <property name="arguments">
    <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">
        <entry key="id" value="action1"/>
        <entry key="message" value="Hello 1"/>
      </w:action>
      <w:action factory="Alert">
        <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>

```

12.1.1.1.2 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

JSON fragment

```
{
  "id": "myConfig",
  "label": "Meine Konfiguration",
  "arguments": [{
    "foo": "bar"
  }, {
    "a.wombat": "x"
  }, {
    "a.gnat": "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.SignFormEditor",
    "args": {
      "expert": true
    }
  }]
}
```

12.2 Principal related data models

12.2.1 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>
```

12.2.2 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>
```

12.2.3 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>
```

12.2.4 JdbcPrincipalDao

de.intarsys.cloudsuite.gears.model.entity.user.JdbcUserDao

A JDBC based implementation for user entity lookup.

Properties

dataSource	
DataSource 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"> • name • key • 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="lookupSql" value="select Principal.name, Claim.key, Claim.value from
Principal inner join Claim on Principal.name = Claim.name where Principal.name=?" />
</bean>
```

12.2.5 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"> • name • salt (optional) • password (optional) • passwordHash (optional)

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>
```

12.2.6 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	The list of model objects for the in-memory table.

required	
----------	--

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>
```

12.2.7 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>

```

12.2.8 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>

```

12.2.9 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>
```

12.2.10 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>
```

12.3 Crypto related data models

12.3.1 ISslContextProvider

de.intarsys.tools.ssl.ConfigurableSslContextProvider

Create an SSL context based on user defined properties.

Properties

protocol	
string	<p>The protocol for the SSL implementation.</p> <p>This should be one of</p> <p>TLS TLSv1 TLSv1.1 TLSv1.2</p> <p>Default TLS</p>
provider	
string	The name of a well known provider
keyManagerProvider	
IKeyManagerProvider	<p>An optional reference to an IKeyManagerProvider.</p> <p>This may no be used in conjunction with the direct assignment of keystore properties</p>
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	<p>The timeout for cached SSL/TLS connections in seconds.</p> <p>Default: 86400 (=24 hours)</p>
trustManagerProvider	
ITrustManagerProvider	<p>An optional reference to an ITrustManagerProvider.</p> <p>This may not be used in conjunction with the direct assignment of truststore properties</p>
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.

12.3.2 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.

12.3.3 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.

12.3.4 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.

13. Implementation hints

13.1 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 validator 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;
}
```

14. String expansion

14.1 Basics

14.1.1 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.

14.1.2 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.

14.1.3 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.
```

14.1.4 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.
```

14.1.5 Namespaces

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

- **properties** for selecting among VM properties
- **environment** for selecting execution context information like working directories

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 the chapters below.

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 **Fehler! Verweisquelle konnte nicht gefunden werden.**).

14.1.6 Spring integration

The concept of string expansion is used throughout the SignLive! 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. 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 post process 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.

14.2 Namespaces

14.2.1 Overview

Here we will learn about the most important namespaces and their respective variables available in Sign Live! cloud suite validator. 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.

14.2.2 app

14.2.2.1 Name

app

14.2.2.2 Description

Access to information about the application.

14.2.2.3 Variables

Name	Description
name	The application's name.
version	The full version of the application.
major	The major version of the application (may be empty).
minor	The minor version of the application (may be empty).
micro	The micro version of the application (may be empty).

14.2.2.4 Availability

This namespace is available after application startup.

14.2.3 config

14.2.3.1 Name

config

14.2.3.2 Description

Access a Spring configuration subset.

14.2.3.3 Variables

Name	Description
<code>*</code>	Any Spring property starting with "config."

14.2.3.4 Availability

This namespace is available after application startup.

14.2.3.5 Example

Given an entry in the `gears.properties`

```
config.foo=bar
```

this expression

```
example ${config.foo}
```

will evaluate to

```
example bar
```

14.2.4 counters

14.2.4.1 Name

counters

14.2.4.2 Description

Zero-based all-purpose counters.

14.2.4.3 Variables

Name	Description
<code><name></code>	An arbitrarily named counter, for example, for creating unique ids. On the first request, the counter is initialized and returns 0. Each consecutive request increments the counter and returns the new value.

14.2.4.4 Availability

This namespace is generally available.

14.2.5 digestSigner

14.2.5.1 Name

digestSigner

14.2.5.2 Description

Access detail information from the current signer process.

14.2.5.3 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
c	The country
o	The organization
ou	The organizational unit

SignatureEvidence properties

Name	Description
items	A list of SignatureItemEvidence

SignatureItemEvidence properties

Name	Description
label	The name of the signed document
digest	The digest of the signed document

14.2.5.4 Availability

This namespace is available in the context of a signature process.

14.2.5.5 Example

Signed by \${digestSigner.subject.cn}

will evaluate to something like

→ Signed by Alexander, the great

14.2.6 entity

14.2.6.1 Name

entity

14.2.6.2 Description

Characters that are difficult to type or to use in some contexts.

14.2.6.3 Variables

Name	Description
amp	ampersand &
backslash	backslash \
copy	copyright © (Unicode U+00A9)
cr	carriage return (Unicode U+000D, \r in Java strings)
gt	greater than >
lf	line feed (Unicode U+000A, \n in Java strings)
lt	less than <

nl	line separator of the VM (usually <code>\n</code> on Unix and <code>\r\n</code> on Windows)
quot	double quote <code>"</code>
squot	single quote <code>'</code>
slash	slash <code>/</code>
trade	trademark [®] (Unicode U+2122)

14.2.6.4 Availability

This namespace is always available.

14.2.7 environment

14.2.7.1 Name

environment

14.2.7.2 Description

Access to the application's file environment. All paths are absolute.

14.2.7.3 Variables

Name	Description
basedir	The application's base directory. Most operations will be performed relative to this directory.
profiledir	The directory for user-specific data.
datadir	The directory for application-private data.
workingdir	The application's working directory.
tempdir	The application's directory for temporary files.

14.2.7.4 Availability

This namespace is available after application startup for trusted use cases.

14.2.8 flow

14.2.8.1 Name

flow

14.2.8.2 Variables

Name	Description
id	The id of the flow (it is equivalent to the conversation id)
variables.<name>	Any variable that was assigned to the flow when created (via variables argument or configuration)

14.2.8.3 Description

Access flow context information

14.2.8.4 Availability

This namespace is available in the context of a flow service execution.

14.2.8.5 Example

```
example ${flow.id}
```

will evaluate to the conversation id for the flow

```
example faadc46e-c367-4963-b497-eb607b8d3f6a
```

```
example ${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
```

14.2.9 identifiers

14.2.9.1 Name

identifiers

14.2.9.2 Description

Generation of unique identifiers

14.2.9.3 Variables

Name	Description
uuid	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) .

14.2.9.4 Availability

This namespace is generally available.

14.2.10 nlsmsg

14.2.10.1 Name

nlsmsg

14.2.10.2 Description

Access a NLS message resource.

14.2.10.3 Variables

Name	Description
*	A message resource path according to the definition in chapter "NLS"

14.2.10.4 Availability

This namespace is available after application startup.

14.2.10.5 Example

With a message resource like the one added in chapter NLS

```
Hi ${nlsmsg.de.intarsys.gears.core.demo.ui.messages#MyButton.label}
```

will evaluate to

```
Hi Tolle Beschriftung
```

14.2.11 properties

14.2.11.1 Name

properties

14.2.11.2 Description

Access to properties in the application's Spring environment.

14.2.11.3 Variables

Name	Description
*	Any property in the Spring environment.

14.2.11.4 Availability

This namespace is available after application startup for trusted use cases.

14.2.12 system

14.2.12.1 Name

system

14.2.12.2 Description

Access some system information.

14.2.12.3 Variables

Name	Description
architecture	Returns the architecture of the current platform, which is either “64-bit” or “32-bit”.
getenv.<name>	The value of the environment variable <name>.
properties.<name>	The value of the VM’s system property <name>.

14.2.12.4 Availability

This namespace is only available for trusted use cases.

14.2.12.5 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
```

14.2.13 time

14.2.13.1 Name

time

14.2.13.2 Description

Time-related properties.

14.2.13.3 Variables

Name	Description
millis	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.

14.2.13.4 Availability

This namespace is generally available.

14.3 Formatting

14.3.1 Overview

The content provided by the variables may sometimes be not well suited for use in a template. For example, take the `${system.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 post process 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.

14.3.2 String formatting

14.3.2.1 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.

14.3.2.2 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.

14.3.2.3 Examples

With `variables.user.greeting` containing **hello, world**

```
example ${variables.user.greeting:s}
```

evaluates to

```
→ example hello, world
```

```
example ${variables.user.greeting:s(7)}
```

evaluates to

```
→ example world
```

```
example ${variables.user.greeting:s(0,4)}
```

evaluates to

```
→ example hello
```

14.3.3 Integer formatting

Integer number formatting is initiated by **i**.

This instruction allows you to convert number values to integer string representations.

14.3.3.1 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

14.3.3.2 Examples

With *system.counter* containing '17'

```
example ${system.counter:i}
```

evaluates to

```
→ example 17
```

```
example ${system.counter:ib}
```

evaluates to

```
→ example 10001
```

```
example ${system.counter:io}
```

evaluates to

```
→ example 21
```

```
example ${system.counter:id}
```

evaluates to

```
→ example 17
```

```
example ${system.counter:ix}
```

evaluates to

```
→ example 11
```

14.3.4 Date formatting

Date formatting is initiated by **d**.

This instruction allows you to convert date values to “human readable” strings.

14.3.4.1 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.

14.3.4.2 Examples

With 'system.millis' containing a timestamp for the 12th of October, 2009 at 23 :33:11 and 1 milliseconds.

```
example ${system.millis:d}
```

evaluates to

```
→ example 2009_10_12-23_33_11_001
```

```
example ${system.millis:ddm}
```

evaluates to

```
→ example 12.10.2009
```

```
example ${system.millis:dts}
```

evaluates to

```
→ example 23:33:11
```

14.3.5 Float formatting

Float formatting is initiated by **f**.

This instruction allows you to convert numeric values to strings using a predefined pattern.

14.3.5.1 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.

14.3.5.2 Examples

With *variables.user.price* containing **1234.567** and an US locale

```
example ${variables.user.price.f(0.0)}
```

evaluates to

```
→ example 1,234.6
```

```
example ${variables.user.price.f(000000)}
```

evaluates to

```
→ example 001235
```

14.3.6 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 **_**.

14.3.6.1 Instruction

```
expr ":p"
```

The evaluation result is converted to a string, then every suspect character is replaced by an underscore.

14.3.6.2 Examples

With *variables.user.foo* containing **my*.file**

```
example ${variables.user.foo:p}
```

evaluates to

```
→ example my_.file
```

14.3.7 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.

14.3.7.1 Instruction

```
expr "!"
```

Apply the expression after the "!" if the first one fails or evaluates to nothing.

14.3.7.2 Example 1

Assuming that "foo" is undefined and "bar" holds "hello"

```
${foo:!bar} world
```

evaluates to

```
hello world
```

14.3.7.3 Example 2

You can use literal expressions as default, too.

```
${foo:!'hello'} world
```

and

```
${foo:!"hello"} world
```

evaluate to

```
hello world
```

14.3.8 Recursion

The result of evaluating an expression may contain other variables - it needs to be reevaluated. For example, in this environment

- **variables.user.name** equals **Jim**
- **variables.global.greeting** equals **Hello, \${variables.user.name}**
- **variables.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.

14.3.8.1 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.

14.3.8.2 Example

So, the complete example for the above should be:

```
${variables.global.greeting:*}! Your application is fully functional!
```

evaluates to

```
Hello, Jim! Your application is fully functional!
```

14.3.9 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.

14.3.9.1 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
```

14.3.9.2 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}
```

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*.

15. Appendices

15.1 Cheat sheet

15.1.1 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

15.1.2 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

15.1.3 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

15.1.4 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`

15.1.5 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
```

15.1.6 Licenses

Licenses are looked up at

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

15.1.7 Documentation

The online documentation is available at

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

15.2 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 and 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 another 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 another 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 same server 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.

15.3 Proxies

15.3.1 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.

15.3.2 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.

15.4 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
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

15.5 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

15.6 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

15.7 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"
}

```

16. External References

[1] i. GmbH, Sign Live! cloud suite validator admin manual.

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