

Sign Live! cloud suite validator admin manual

September 2025

intarsys GmbH

Sign Live! cloud suite validator admin manual

Version 8.14

cloud suite validator admin architecture, design & reference

intarsys GmbH
Sign Live! cloud suite validator admin 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	11
1. Overview	12
2. Building blocks	13
2.1 Services	13
2.1.1 GraphQL	13
3. Installation	14
3.1 Overview	14
3.2 Documentation	14
3.3 Installation Process	14
3.3.1 Server	14
3.4 Web Apps	14
3.4.1 Web App "validator-admin"	14
3.5 SDK	14
3.6 Version check web app	15
3.7 Version check server	15
3.7.1 In the WEB-INF directory	15
3.7.2 In the log	15
4. Configuration	16
4.1 Overview	16
4.1.1 Configuration location	16
4.1.2 Built-in configuration	18
4.1.3 Custom property definition	18

Content

4.1.4	Custom bean definition	18
4.1.5	Using profiles	19
4.1.6	String expansion integration	19
4.2	Web application root	20
4.3	Logging	20
4.3.1	Override logging	21
4.3.2	Builtin logging	21
4.3.3	Log correlation	21
4.3.4	Log tweaks	22
4.4	Licenses	22
4.5	Data source	23
5.	String expansion	24
5.1	Basics	24
5.1.1	Why string expansion	24
5.1.2	Terminology	24
5.1.3	Syntax	25
5.1.4	Constant text in expression	25
5.1.5	Namespaces	25
5.1.6	Spring integration	26
5.2	Namespaces	27
5.2.1	Overview	27
5.2.2	app	27
5.2.3	config	27
5.2.4	counters	28
5.2.5	digestSigner	29
5.2.6	entity	30
5.2.7	environment	31
5.2.8	flow	31
5.2.9	identifiers	32
5.2.10	nlsmg	33
5.2.11	properties	33
5.2.12	system	34
5.2.13	time	34
5.3	Formatting	35
5.3.1	Overview	35
5.3.2	String formatting	36
5.3.3	Integer formatting	37
5.3.4	Date formatting	38
5.3.5	Float formatting	39
5.3.6	File path formatting	40
5.3.7	Default value	40
5.3.8	Recursion	41
5.3.9	Conditional evaluation	42
6.	Appendices	44
6.1	Cheat sheet	44

Content

6.1.1	Windows locations	44
6.1.2	Linux locations	44
6.1.3	Property definitions	44
6.1.4	Bean definitions	44
6.1.5	Logging	45
6.1.6	Licenses	45
7.	External References	46

Introduction

Sign Live! cloud suite validator admin provides APIs allowing the web-based management of the gears validator components.

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

1. Overview

Sign Live! cloud suite validator admin provides workflow enabled components, ready to be used easily in web based (and for some part, fat client based) products. The application is vastly configurable and extensible, given that you have access to the system environment hosting the application and that you are fine juggling with Spring configuration files.

However, there are common application management tasks like rule and template management which may be performed by business units external to the administration.

This is exactly where Sign Live! cloud suite validator admin comes into play.

Sign Live! cloud suite validator admin offers web-based AIPs giving integrators control over runtime configuration.

Sign Live! cloud suite validator admin is sharing concepts and patterns with Sign Live! Cloud suite gears and is based on the same architecture.

Accordingly, the terms “validator admin”, “gears validator admin” and “gears” are synonymously used in this document.

2. Building blocks

2.1 Services

2.1.1 GraphQL

The **graphql** service provides read and write access to the system's manageable entities.

You can access the service endpoint at

`http://<host>/< validator admin context>/graphql`

The endpoint offers queries and mutations for exhaustive interaction with the server-side model. Client-side programming tools and APIs make use of the GraphQL schema, which is requestable at this endpoint by standard means, and use the schema information e.g. for the generation of client stubs.

3. Installation

3.1 Overview

The product is shipped in a ZIP or TAR file along with the validator application components. Please see [1] for details on the system requirements.

3.2 Documentation

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

Additionally, the GraphQL API is documented in the form of a GraphQL schema, which allows online discovery of the available operations.

Developer tools like Postman and GraphiQL offer convenient explorative access to model operations, objects and corresponding API documentation.

3.3 Installation Process

3.3.1 Server

Perform the following installation steps:

1. Install Java Runtime
2. Install Tomcat
3. Deploy the "cloudsuite-gears#validator-admin.war" into the Tomcat webapps directory. For further details read chapter 3.4.

3.4 Web Apps

3.4.1 Web App "validator-admin"

This web application provides administrative to validation-related data.

3.5 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.

3.6 Version check web app

3.7 Version check server

3.7.1 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
```

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

4. Configuration

4.1 Overview

Sign Live! cloud suite validator admin 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.

4.1.1 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.

4.1.1.1 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

4.1.1.2 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

```

4.1.2 Built-in configuration

Sign Live! cloud suite validator admin 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.

4.1.3 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

4.1.4 Custom bean definition

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

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

- User level
`${cloudsuite.config.user}/spring-gears-validatoradmin-*`
`${cloudsuite.config.user}/modules/spring-gears-validatoradmin-*`

4.1.5 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

4.1.6 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".

4.2 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
```

4.3 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.

4.3.1 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.

4.3.2 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

4.3.3 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}
```

4.3.4 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
```

4.3.4.1 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.

4.3.4.2 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
```

4.3.4.3 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
```

4.3.4.4 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
```

4.4 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
```

4.5 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>
```

5. String expansion

5.1 Basics

5.1.1 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.

5.1.2 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.

5.1.3 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.
```

5.1.4 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.
```

5.1.5 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.**).

5.1.6 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.

5.2 Namespaces

5.2.1 Overview

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

5.2.2 app

5.2.2.1 Name

app

5.2.2.2 Description

Access to information about the application.

5.2.2.3 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).

5.2.2.4 Availability

This namespace is available after application startup.

5.2.3 config

5.2.3.1 Name

config

5.2.3.2 Description

Access a Spring configuration subset.

5.2.3.3 Variables

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

5.2.3.4 Availability

This namespace is available after application startup.

5.2.3.5 Example

Given an entry in the `gears.properties`

```
config.foo=bar
```

this expression

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

will evaluate to

```
example bar
```

5.2.4 counters

5.2.4.1 Name

counters

5.2.4.2 Description

Zero-based all-purpose counters.

5.2.4.3 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.

5.2.4.4 Availability

This namespace is generally available.

5.2.5 digestSigner

5.2.5.1 Name

digestSigner

5.2.5.2 Description

Access detail information from the current signer process.

5.2.5.3 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

5.2.5.4 Availability

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

5.2.5.5 Example

Signed by \${digestSigner.subject.cn}

will evaluate to something like

→ Signed by Alexander, the great

5.2.6 entity

5.2.6.1 Name

entity

5.2.6.2 Description

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

5.2.6.3 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 <code>®</code> (Unicode U+2122)

5.2.6.4 Availability

This namespace is always available.

5.2.7 environment

5.2.7.1 Name

environment

5.2.7.2 Description

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

5.2.7.3 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.

5.2.7.4 Availability

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

5.2.8 flow

5.2.8.1 Name

flow

5.2.8.2 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)

5.2.8.3 Description

Access flow context information

5.2.8.4 Availability

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

5.2.8.5 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
```

5.2.9 identifiers

5.2.9.1 Name

identifiers

5.2.9.2 Description

Generation of unique identifiers

5.2.9.3 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) .

5.2.9.4 Availability

This namespace is generally available.

5.2.10 nlsmsg

5.2.10.1 Name

nlsmsg

5.2.10.2 Description

Access a NLS message resource.

5.2.10.3 Variables

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

5.2.10.4 Availability

This namespace is available after application startup.

5.2.10.5 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
```

5.2.11 properties

5.2.11.1 Name

properties

5.2.11.2 Description

Access to properties in the application's Spring environment.

5.2.11.3 Variables

Name	Description
*	Any property in the Spring environment.

5.2.11.4 Availability

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

5.2.12 system

5.2.12.1 Name

system

5.2.12.2 Description

Access some system information.

5.2.12.3 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>.

5.2.12.4 Availability

This namespace is only available for trusted use cases.

5.2.12.5 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
```

5.2.13 time

5.2.13.1 Name

time

5.2.13.2 Description

Time-related properties.

5.2.13.3 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.

5.2.13.4 Availability

This namespace is generally available.

5.3 Formatting

5.3.1 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.

5.3.2 String formatting

5.3.2.1 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.

5.3.2.2 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.

5.3.2.3 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
```

5.3.3 Integer formatting

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

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

5.3.3.1 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

5.3.3.2 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
```

5.3.4 Date formatting

Date formatting is initiated by **d**.

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

5.3.4.1 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.

5.3.4.2 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
```

5.3.5 Float formatting

Float formatting is initiated by **f**.

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

5.3.5.1 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.

5.3.5.2 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
```

5.3.6 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 **_**.

5.3.6.1 Instruction

```
expr ":p"
```

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

5.3.6.2 Examples

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

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

evaluates to

```
→ example my_.file
```

5.3.7 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.

5.3.7.1 Instruction

```
expr "!"
```

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

5.3.7.2 Example 1

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

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

evaluates to

```
hello world
```

5.3.7.3 Example 2

You can use literal expressions as default, too.

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

and

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

evaluate to

```
hello world
```

5.3.8 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.

5.3.8.1 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.

5.3.8.2 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!
```

5.3.9 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.

5.3.9.1 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
```

5.3.9.2 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*.

6. Appendices

6.1 Cheat sheet

6.1.1 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

6.1.2 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

6.1.3 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

6.1.4 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`

6.1.5 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
```

6.1.6 Licenses

Licenses are looked up at

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

7. External References

[1] intarsys AG, Sign Live! cloud suite gears validator manual.