uxodocs

FlowerDocs / uxodocs

Author: Your name

Date: 2026-03-13

Contents

1. Answers with reason 0
2. Appropriation 0
3. Assignment 0
4. Access control list (ACL) 0
5. Order of access control inputs (ACE) 0
6. ACL Proxy 0
7. Principle 0
8. Subscription 0
9. Operations Manager 0
10. Document management 0
11. Treatment 0
12. Processing stage tags 0
13. Attachment 0
14. Replies 0
15. Consultation 0
16. Validation 0
17. FlowerDocs GUI 0
18. FlowerDocs Core 0
19. ARender Rendition Server 0
20. OpenSearch 0
21. Content storage 0
22. Redis 0
23. Operating system 0
24. Linux 0
25. Application components 0
26. OpenSearch and Redis 0
27. Sizing 0
28. Encoding 0
29. Linux 0
30. Service init.d 0

31. Service systemd 0
32. JVM configuration 0
33. Commands 0
34. Prerequisites 0
35. ARender Rendition installation 0
36. FlowerDocs installation 0
37. Creation 0
38. Access 0
39. Scope 0
40. ACLs 0
41. Component classes 0
42. Task classes 0
43. Folder classes 0
44. Virtual folder classes 0
45. Tag classes 0
46. Components 0
47. Documents 0
48. Folders 0
49. Tasks 0
50. Tag categories 0
51. Process 0
52. Facts 0
53. Installation procedure 0
54. Configuration 0
55. Security 0
56. Start 0
57. Validation 0
58. File system 0
59. Amazon S3 0
60. Changing index parameters during restoration 0
61. Restoration to a different cluster 0
62. Snapshot status 0

63. Progression 0
64. Stop 0
65. Source: Elasticsearch 0
66. Target: OpenSearch 0
67. Identifier 0
68. Title, description and category 0
69. Columns to be displayed 0
70. Filter 0
71. Sorting 0
72. In practice 0
73. Appropriation 0
74. Assignment 0
75. Creation of a document containing the logo 0
76. Add customized logo from FlowerDocs administration 0
77. The job offer 0
78. The application 0
79. Creating the stacked icon 0
80. Adding a button to the header containing our icon 0
81. Adapting the size of icons 0
82. Relative positioning of icons 0
83. Bonus: a touch of color 0
84. Project creation 0
85. Configuration 0
86. Maven project 0
87. Auto-configuration 0
88. Compilation 0
89. Linux service 0
90. Executable JAR 0
91. Open 0
92. Navigation 0
93. URL-based tabs 0
94. Folder-based tabs 0

95. Search-based tabs 0
96. Task creation 0
97. Confirmations 0
98. Save 0
99. Cancel 0
00. Document 0
01. Virtual folder 0
02. Customize the virtual folder search 0
03. Task 0
04. Field definition 0
05. Configuration 0
06. ComponentSearchPresenter 0
07. KeywordCriteriaPresenter 0
08. AdvancedCriteriaPresenter 0
09. Free criteria 0
10. Fixed criteria 0
11. Criterion types and operators 0
12. Examples 0
13. Assigning criteria to the advanced search 0
14. Tab 0
15. Indexation 0
16. Tabular and thumbnails (switcher) 0
17. Tabular only 0
18. Thumbnail only 0
19. Component creation 0
20. Screen change 0
21. Global 0
22. By scope 0
23. Security 0
24. From the graphical user interface 0
25. REST API 0
26. Access concerned 0

27. Content generation with obfuscation 0
28. Annotation styles 0
29. Toaster management 0
30. Scope-specific plugin 0
31. Global plugin 0
32. Simple 0
33. Sentinel 0
34. Configuration 0
35. A version 0
36. Version tracking 0
37. Creating a version 0
38. Restoring a version 0
39. Comparing two versions 0
40. Tracking modes 0
41. Version label 0
42. Storage 0
43. Directory type 0
44. Directory definition by scope 0
45. Default directory definition 0
46. Configuration examples 0
47. Access account 0
48. Link to FlowerDocs 0
49. Endpoints 0
50. Examples 0
51. Document-specific permissions: 0
52. Task-specific permissions: 0
53. Virtual folder-specific permissions: 0
54. Default setting 0
55. Concept 0
56. Key management 0
57. Derived key 0
58. Keystore 0

59. Algorithm 0
60. File system encryption 0
61. Directory analysis 0
62. Directory encryption 0
63. Derived key 0
64. Keystore 0
65. Texts 0
66. Conditions 0
67. Defining a new property 0
68. Using a property 0
69. Other actions 0
70. Filter evaluation 0
71. Storage 0
72. Special features 0
73. Assignment 0
74. Task response 0
75. Folder contents 0
76. Search 0
77. Context-related 0
78. Utilities 0
79. Pre-imported classes 0
80. Creating a folder when creating a document 0
81. Creating an audit fact 0
82. Workflow state machine (task class transitions) 0
83. Search and conditionally create a virtual folder 0
84. Modifying tags and ACLs 0
85. Throwing exceptions 0
86. Instantiate an object 0
87. Restrictions 0
88. Application metrics 0
89. Components 0
90. OperationHandler 0

91. Other 0
92. Technical metrics 0
93. REST API 0
94. JMX 0
95. Examples 0
96. Adding or modifying a tag 0
97. Tag recovery 0
98. File access 0
99. Files modification 0
00. Assigning tasks 0
01. Responding to an assignment 0
02. Form validity change subscription 0
03. Actions in a form 0
04. Adding cards 0
05. Form status 0
06. Form 0
07. Criteria 0
08. Subscription 0
09. Bucket modification 0
10. Hiding a bucket 0
11. In a form 0
12. Go to an action 0
13. Available functions 0
14. Interactions with component actions 0
15. Button action 0
16. Icon action 0
17. Menu action 0
18. Responsive action 0
19. DOM action 0
20. Card 0
21. Container cards 0
22. Search result 0

23. Attachment 0
24. Loading 0
25. Changes 0
26. Actions 0
27. Dynamic call 0
28. Link with a tag 0
29. Object identification 0
30. Context 0
31. Additional fields 0
32. Select clause 0
33. Clauses Filter 0
34. Clauses Order 0
35. Document recovery 0
36. Version recovery 0
37. Retrieving associated files 0
38. Document modification with content replacement 0
39. Data modification 0
40. Add file 0
41. Rename file 0
42. Document deletion 0
43. File deletion 0
44. Content recovery 0
45. Index document content 0
46. Remove content indexing from a document 0
47. All annotations recovery 0
48. From a json 0
49. From xml 0
50. Part of annotations deletion 0
51. All annotations deletion 0
52. Deleting a version 0
53. All versions deletion 0
54. Simple answer 0

55. Answer with reason 0
56. Retrieve all task classes 0
57. Retrieve a defined list of task classes 0
58. Model 0
59. Reply 0
60. Example 0
61. Model 0
62. Reply 0
63. Examples - Download 0
64. Examples - Temporary file 0
65. Model 0
66. Example 0
67. Model 0
68. Example 0
69. Model 0
70. Example 0
71. Model 0
72. Example 0
73. Model 0
74. Example 0
75. Generate a token with a specific lifetime 0
76. Generate a new token 0
77. Generate a token to access documents 0
78. Maven 0
79. Spring Boot application 0
80. Service account 0
81. Token generation 0
82. Access to services 0
83. Log management 0
84. Access to FlowerDocs IHM 0
85. Access account & Scope 0
86. Browser dimensions 0

87. FlowerDocs 0
88. SMTP 0
89. Email body 0
90. Signature 0
91. Sessions 0
92. Caches 0
93. Temporary files 0

Concepts / Getting Started

INFO

Get to grips with the concepts behind the FlowerDocs data model, designed to facilitate:

- understanding
- reusability of defined objects
- solution administration

This section briefly describes the concepts involved in setting up a FlowerDocs-based solution.

INFO

Are your documents still in *paper format*?

Contact us to find out more about our expertise in dematerialisation and set up a dematerialisation chain to feed your FlowerDocs solution.

Concepts / Scope

> **INFO**
>
> Scopes make it possible to isolate data between different customers/businesses while using the same platform.

A scope defines an application silo by isolating its data and configuration. As such, it defines :

- user teams
- users who can access it
- languages used

To connect to a scope, you need to enter its identifier as a URL parameter (for example: `http://flowerdocs.com/gui?scope=GEC`.
If FlowerDocs is behind a proxy, using the HTTP header `scope` redirects the user to **FlowerDocs GUI** with the URL parameter `scope` and the value provided.

If the parameter is not set, the default scope defined with the `scope.default` property in the `gui.properties` configuration file is used.

It is possible to display the scope selection on the login page by configuring the `scope.edit property` in the `gui.properties` configuration file.

INFO

Access authorization to a scope is determined by the read permission on the access control list defined at scope level. More details on this mechanism can be found here.

Concepts / Component classes / Overview

A component class defines the common characteristics of a logical set of components. These sets are characterised by tags (or metadata), security, business or technical rules that are specific to them.

This section defines the notion of component class, used to characterise the components (documents, folders, tasks, etc.) handled within the application. In this way, every component refers to a class of components via its identifier.

When defining identifiers, we recommend using only: alphanumeric characters, "-" and "_" (spaces are not recommended).
Regex: `^[a-zA-Z0-9-_]$`
This recommendation is mandatory for all FlowerDocs versions above 2.6.

A component class defines a type of component:

- tags that can be linked to a component
- the default security to be applied by defining an ACL identifier to be applied
- tag categories to visually group tags into functional blocks
- international labels to provide a multilingual application
- technical character

Depending on the category of the component class, special features can be added.
A tag referenced to a component class can be characterized by the following parameters:

- Mandatory: indicates whether a value is mandatory for validation
- Technical: indicates whether the user has access to this tag
- Read-only: indicates whether or not the user can modify the tag value
- Multivalued: indicates whether or not the tag can have several values
- Default value: the default value when the tag is not filled in
- A validation mask (regular expression): if defined, overrides the one defined in the tag class
- Display order
- A description to display a tooltip

> **INFO**
>
> The default value is set by **FlowerDocs Core** only at the time of creation: if the tag value is emptied, the default value will not be set.
> On creation, the default value is set if the tag is non-mandatory and not present. If it is non-mandatory with an empty value, then the empty value is set.
> On the GUI side, the default value is set when the component creation form is opened.

Concepts / Component classes / Default value

Default values are only valid at creation: if the tag value is empty, the default value will not be set.

Variables can be used as default values:

- ``$`dayDate```: the current date
- `${user.id}`: the identifier of the connected user

> **INFO**
>
> Unlike virtual folders, which are made up of searches that dynamically find components, folders have a physical model through a parent-child relationship with the components they contain.
>
> A folder class defines the folder model to be created. Its specificity is the notion of the child.

Concepts / Component classes / Children

The folder class defines which child classes it is authorised to contain. As many as possible can be defined. If you need to contain all the classes of a component type, for example, all the documents, you can add a child with the identifier `*` as follows:

```
<ns2:children category="DOCUMENT">
                <id>*</id>
</ns2:children>
```

INFO

A task class is used to define the task model to be created. Its special features are :

- an icon (example: `fa fa-envelope`)
- a workflow identifier
- possible answers
- expected attachments

Concepts / Component classes / Attachments

When processing a task, it may be necessary for the user to add one or more components.

These components are called task attachments and are globally defined by task class.

Each expected attachment has the following attributes:

- a identifier
- an internationalized name
- definition of a component class (identifier and category)
- a tag list: precisely defines attachment metadata

These attachments can also be characterized by the following parameters:

- required: defines whether the attachment must be added to validate changes made to the task, required later or optional
- technical: defines whether or not the attachment should be displayed to users
- multivalued: defines whether the attachment can be composed of several components
- read-only: defines whether users can modify the attachment (the read-only aspect of an attachment also depends on the permissions defined at task level)

To describe an attachment, you can define :

- a description that will be displayed when the mouse hovers over the attachment

- a summary displayed under the name of the component added as an attachment

Summary

The summary can be configured, by scope language, to display the tags of the attached component (example: `${Priority} Created by ${owner}`)

By default, the summary displays the component creation date.

To go even further, you can use the JS API to define your own summary. A dedicated section here.

Answers

Responses are used to steer tasks in a direction suggested by a workflow. Each answer has the following attributes:

- an identifier
- an internationalized name
- an internationalized confirmation message

Answers with reason

You can define a patterned response as follows:

```
<ns4:answers xsi:type="ns4:ReasonedAnswerDefinition"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <id>Initiate</id>
        <ns4:displayNames language="EN">
                <ns3:value>Send</ns3:value>
        </ns4:displayNames>
        <ns4:displayNames language="FR">
                <ns3:value>Send</ns3:value>
        </ns4:displayNames>
        <ns4:reasons tagName="Matricule" order="0"
mandatory="true">
                <ns4:descriptions language="EN">
                        <ns3:value>MXXXX - Enter the id of new
adherent</ns3:value>
                </ns4:descriptions>
                <ns4:descriptions language="FR">
                        <ns3:value>MXXXX - New adherent
identifier</ns3:value>
                </ns4:descriptions>
                <ns4:pattern>^M[0-9]*$</ns4:pattern>
        </ns4:reasons>
        <ns4:reasons tagName="Comments" order="0" mandatory="true"
/>
</ns4:answers>
```

To add a motif with a pattern, the following properties must be defined:

- `descriptions`, the internationalized description of the tag when the user has not yet entered a value in the field
- `pattern`, the rule to apply

For example, the rule `^M[0-9]*$` accepts only values beginning with `M` and ending with digits.

> **INFO**
>
> Unlike folders, which have a parent-child relationship with the components they contain, virtual folders are made up of searches that allow components to be found dynamically.
>
> A virtual folder class defines the virtual folder model to be created. What sets it apart is the notion of research.

Virtual folders of the same class share the same search list, so they all have the same component presentation.

The special feature of file searches is the ability to aggregate results according to a specific criterion, and the variability of searches.

Concepts / Component classes / Aggregation

If the search defined in the folder contains an aggregation, the components brought up by the search will be grouped together in the form of `Bucket`. Aggregations can also be nested to create `buckets` based on multiple criteria.

It is recommended not to aggregate more than 3 levels.

When the tree structure of a virtual folder is in `View` mode, we recommend not exceeding 100 documents displayed in ARender.

Research variabilization

In searches for a class of virtual folders, the defined criteria can be either fixed values or variables. It works in the same way as conditional tags for variable resolution.

Thanks to this mechanism, it is possible to define a single virtual folder of the *Personal tray* using a criterion `assignee = ${user.id}`. Variables of `tags` type are based on the tags carried by the virtual folder.

We recommend using criteria with fixed values (such as choice lists) and not exceeding 100 possible values.

Aggregations on date, integer, string and text criteria are not supported by FlowerDocs.

Concepts / Tags / Overview

`tags` are metadata defined on a component to characterize it and make it easier to find.

This metadata can be of different types to ensure consistency of the data stored in FlowerDocs.

A tag class defines the typology and characteristics of a tag within a scope. This tag class applies to all tags associated with components, and is made up of the following elements:

- `id`: technical tag identifier
- `displayNames`: tag display name (ex : Agency name, building no., etc.) internationalized
- `searchable`: Boolean to make the tag searchable or not
- `Type`: tag id

Type	Description
`STRING`	String (e.g. reference, customer name ...)
`TEXT`	Text field (ex: comment ...)
`CHOICELIST`	Choice list (e.g. type of document ...)
`ICON`	Icon list (e.g. input channel ...)

Type	Description
`FREELIST`	Choice list (e.g.: reference linked to an external IS ...)
`CONDITIONAL`	Conditional value list (e.g.: list of departments according to direction ...)
`FLOAT`	Decimal number (e.g.: rate, proportion ...)
`INT`	Integer (e.g.: number of supporting documents ...)
`CURRENCY`	Decimal number limited to 2 decimal places (e.g.: amount ...)
`DATE`	Date (e.g. due date ...)
`BOOLEAN`	Check box (e.g.: medical confidentiality ...)
`USER`	Linked to a directory user (e.g. manager)

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:TagClass xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/tagclass"
        xmlns:ns3="http://flower.com/docs/domain/i18n">
        <id>Identifier</id>
        <ns2:displayNames language="EN">
                <ns3:value>Display name</ns3:value>
        </ns2:displayNames>
        <ns2:displayNames language="FR">
                <ns3:value>Display name</ns3:value>
        </ns2:displayNames>
        <ns2:data>
        <owner>owner</owner>
        <creationDate>2017-10-03 23:26:32.169
+0200</creationDate>
        <lastUpdateDate>2017-04-02 11:44:40.909
+0200</lastUpdateDate>
    </ns2:data>
        <ns2:type>TYPE</ns2:type>
    <ns2:searchable>true</ns2:searchable>
</ns2:TagClass>
```

INFO

Add tags to your components by referencing a tag class at component class level.

The FlowerDocs data model allows you to define textual tags:

- character strings `STRING`
- text fields `TEXT`

Concepts / Tags / Character strings

This type allows free input of a character string. With this type of tag, you can add a regular expression to the `pattern` attribute to validate the user's input.

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8"?>
<ns2:TagClass xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/tagclass"
        xmlns:ns3="http://flower.com/docs/domain/i18n">
        <id>CustomerName</id>
    <ns2:type>STRING</ns2:type>
    <ns2:pattern>[A-Z]*</ns2:pattern>
</ns2:TagClass>
```

Text

This type allows free input in a text zone.

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8"?>
<ns2:TagClass xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/tagclass"
        xmlns:ns3="http://flower.com/docs/domain/i18n">
        <id>Comment</id>
    <ns2:type>TEXT</ns2:type>
</ns2:TagClass>
```

This section describes the different types of list-based tags:

- Choice list `CHOICELIST`
- Icon list `ICON`
- Free list `FREELIST`

For performance reasons, we do not recommend the use of choice lists or icon lists with more than 1000 values.

For integration with larger volumes, please contact the FlowerDocs team.

Also, FlowerDocs only displays a maximum of 100 items in the list, even if it contains more. Simply refine the selection to display items not displayed by default.

Concepts / Tags / List of choices

A choice list tag allows you to restrict the possible values of a tag to defined choices.

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:TagClass xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/tagclass"
        xmlns:ns3="http://flower.com/docs/domain/i18n">
        <id>TypeCourrier</id>
        <ns2:type>CHOICELIST</ns2:type>
        <ns2:allowedValues symbolicName="FactureClient">
                <ns2:displayNames language="FR">
                        <ns3:value>Customer invoice</ns3:value>
                </ns2:displayNames>
        </ns2:allowedValues>
        <ns2:allowedValues symbolicName="BonDeCommande">
                <ns2:displayNames language="FR">
                        <ns3:value>Order form</ns3:value>
                </ns2:displayNames>
        </ns2:allowedValues>
</ns2:TagClass>
```

Icon list

An icon list tag is an extension of a list of choices for which an icon can be defined.

This icon must be defined with the appropriate CSS class using the library available in FlowerDocs Font Awesome 6.5.2

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:TagClass xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/tagclass"
        xmlns:ns3="http://flower.com/docs/domain/i18n">
        <id>Priority</id>
        <ns2:type>ICON</ns2:type>
        <ns2:allowedValues symbolicName="1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="AllowedValueWithIcon">
                <ns2:icon>fa fa-flag black</ns2:icon>
                <ns2:displayNames language="FR">
                        <ns3:value>Critique</ns3:value>
                </ns2:displayNames>
        </ns2:allowedValues>
</ns2:TagClass>
```

If two icons are to be placed side by side, an icon of the following type can be used: `stacked(fas fa-exclamation red,fas fa-flag red)`.

Free list

A free list tag is an extension of a choice list without validation. This type of list can be used to value a tag with values that are not known to FlowerDocs.

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:TagClass xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/tagclass"
        xmlns:ns3="http://flower.com/docs/domain/i18n">
        <id>ReferenceClient</id>
        <ns2:type>FREELIST</ns2:type>
</ns2:TagClass>
```

INFO

This type of tag is particularly well suited to integration with third-party repositories (for example, using a lookup.

The `CONDITIONAL` type is used to define conditions on the various choices (or sets of choices) offered to the user in indexing or search forms.

Objects `com.flower.docs.domain.tagclass.ConditionalAllowedValue` supporting multiple conditions can be defined for a given choice. In this case, all that's needed is for one condition to be satisfied for the user to be offered a choice.

Concepts / Tags / Tag conditions

Conditions can relate to the tags of a component or a search form. *They consist of an identifier, an operator and a value.*

To identify the tag to which the condition applies, you need to use a character string of the type `${tags.<tag_id>}` where `<tag_id>` is the tag identifier.

For tags, the operators `==` or `!=` can be used to indicate the presence or non-presence of a value, respectively.

Typically, the different types of conditions supported for tags are as follows:

- `${tags.<tag_id>}==X`: Existing tag containing (at least) the value *X*
- `${tags.<tag_id>}!=X`: Tag does not exist or does not contain the value indicated by *X*

> **INFO**
>
> These conditions apply only to tags displayed in the indexing or search form.

`CONDITIONAL` tags are not used in multivalued mode, otherwise the conditions for each value cancel each other out.

Please note that a conditional tag can only be defined on two levels. Only the first level includes `ConditionalAllowedValue` tags, the lower level only includes `allowedValues` tags. In addition, the symbolicName of each `allowedValues` tag must be unique.

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<TagClass xmlns="http://flower.com/docs/domain/tagclass"
xmlns:ns4="http://flower.com/docs/domain/i18n">

        <id>B_ServiceDestinataire</id>
        <ns2:type>CONDITIONAL</ns2:type>
    <ns2:allowedValues xsi:type="ns2:ConditionalAllowedValue"
symbolicName="SERVICE" xmlns:xsi="http://www.w3.org/2001/XMLSchem
instance">
            <ns2:allowedValues symbolicName="NA">
                <ns2:displayNames language="EN">
                    <ns3:value>--Undefined department--</ns3:valu
                </ns2:displayNames>
                <ns2:displayNames language="FR">
                    <ns3:value>--Service indéfini--</ns3:value>
                </ns2:displayNames>
            </ns2:allowedValues>
            <ns2:allowedValues symbolicName="ERROR">
                <ns2:displayNames language="EN">
                    <ns3:value>Error</ns3:value>
                </ns2:displayNames>
                <ns2:displayNames language="FR">
                    <ns3:value>Error</ns3:value>
                </ns2:displayNames>
            </ns2:allowedValues>

<ns2:conditions>${tags.B_DirectionDestinataire}==00NA</ns2:condit
    </ns2:allowedValues>

</TagClass>
```

The FlowerDocs data model allows you to define three types of digital tags:

- integers `INT`

- decimal numbers `FLOAT`
- amounts `CURRENCY`

Concepts / Tags / Integer

This type is used to enter an integer number.

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8"?>
<ns2:TagClass xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/tagclass"
        xmlns:ns3="http://flower.com/docs/domain/i18n">
        <id>NumberDocument</id>
    <ns2:type>INT</ns2:type>
</ns2:TagClass>
```

Decimal

This type is used to enter a decimal number.

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8"?>
<ns2:TagClass xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/tagclass"
        xmlns:ns3="http://flower.com/docs/domain/i18n">
        <id>RateResa</id>
    <ns2:type>FLOAT</ns2:type>
</ns2:TagClass>
```

Amount

This type allows you to enter a decimal number with only 2 decimals stored and displayed.

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8"?>
<ns2:TagClass xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/tagclass"
        xmlns:ns3="http://flower.com/docs/domain/i18n">
        <id>AmountTTC</id>
    <ns2:type>CURRENCY</ns2:type>
</ns2:TagClass>
```

The FlowerDocs data model allows you to define tags of type `DATE`.

Concepts / Tags / Stored value

The stored value of a `DATE` tag is the timestamp associated with the date. This timestamp corresponds to the number of milliseconds elapsed since 1st January 1970 UTC. In the case of a date prior to this reference date, the number of milliseconds is negative.

Display format

Although the value of a `DATE` tag is stored as a timestamp, it can be displayed in different ways.

Configuring the format in which a date is displayed allows you to define how it will be presented to users.

This configuration can be carried out using the `pattern` field at the level of a tag class or tag reference, and accepts the following formats:

Name	Description	Example
DATE_FULL	Full date	Friday 23rd October 2020
DATE_LONG	Long date	23rd October 2020
DATE_MEDIUM	Medium date	23rd Oct. 2020
DATE_SHORT	Short date	23/10/2020
DATE_TIME_FULL	Full date + time	Friday 23rd October 2020 00:00:00 UTC+2
DATE_TIME_LONG	Long date + time	23rd October 2020 00:00:00 UTC+2
DATE_TIME_MEDIUM	Medium date + time	23 Oct. 2020 00:00:00

Name	Description	Example
DATE_TIME_SHORT	Short date + time	23/10/2020 00:00
ISO_8601	ISO 8601	2020-10-23T00:00:00.000+02:00
RFC_2822	RFC 2822	Fri, 23 Oct 2020 00:00:00 +0200

Concepts / Tags / Categories

Tag categories enable logically related tags to be grouped together for display.

Each tag category defines:

- `id`: category identifier
- `displayNames`: internationalized labels
- `tags`: tags contained in the category
- `icon` : icon based on FontAwesome fond (example : `fas fa-star`)
- `description`: description to be displayed to the user
- `reduced`: indicates whether the category is folded by default

To be displayed, a category must be linked to a component class.

> **INFO**
>
> Tags defined in a category and not referenced at component class level are not displayed on the indexing form.

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns3:TagCategory icon="flower-button header-icon fa fa-envelope
flat-purple"
        description="Fill in the fold identification
information" visible="true" inline="false" reduced="false"
        xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/i18n"

xmlns:ns3="http://flower.com/docs/domain/componentclass">
        <id>MailInfo</id>
        <ns3:displayNames language="FR">
                <ns2:value>Mail information</ns2:value>
        </ns3:displayNames>
        <ns3:displayNames language="EN">
                <ns2:value></ns2:value>
        </ns3:displayNames>
        <ns3:tags>MailDate</ns3:tags>
        <ns3:tags>SubjectMail</ns3:tags>
</ns3:TagCategory>
```

Concepts / Components / Overview

The objects handled through the interface and APIs are components. Each component is classified according to business or technical criteria using a component class.

> **INFO**
>
> Find out more about the different component categories:
>
> - Documents
> - Folders
> - Virtual folders
> - Tasks

Concepts / Components / Documents

A document is a component in its own right, with the distinctive feature of having one or more contents. These are also known as files.

In order to restrict access to, or modifications of, a document's content, a number of permissions are available to control possible operations:

- Viewing content: `READ_CONTENT`
- Adding or deleting content from a document: `UPDATE_CONTENT`
- Downloading document content: `DOWNLOAD_CONTENT`
- Enable printing in the viewer: `PRINT`
- Viewing document content annotations: `READ_ANNOTATION`
- Annotating document content: `CREATE_ANNOTATION`
- Document composition: `BUILD_NEW_DOCUMENT`

Some actions are only available if the user has write access to the document. To do this, he/she must have the `UPDATE` permission and have reserved the document (see Reservation.

- Attaching the document to a folder: `ATTACH`
- To delete the document/folder link, the `DETACH` permission is evaluated on the folder, not on the document

INFO

To go further, consult the Javadoc:

- Document
- Document class

INFO

Organize your documents by folder for easy retrieval.

Concepts / Components / Content

A folder is a component with children. These children can be documents or other folders.

Constraints can be defined at folder class level to restrict the children that can be added to a folder. To do this, it is possible to reference the authorised component classes.

Adding components

Via the interface, when the folder defines children, an `Add` button is used to select the component category to be created.

When indexing, it is possible to select the component class. All tags in common are listed on the write indexing form.

Note: If there is only one component class, this is automatically selected.

Permissions

In order to restrict the access or modifications that can be made to a folder, several permissions are available to control the operations that can be carried out:

- Add or remove a child from a folder : `UPDATE_CONTENT`
- Detaching a document from a folder: `DETACH` (action possible from document indexing)
- Downloading the contents of a folder as an archive: `DOWNLOAD_CONTENT` Downloaded content contains only the documentary content of the folder and does not includes sub-folders.

Some actions are only available if the user has write access to the folder. To do this, it must have the `UPDATE` permission and have reserved the folder (see Reservation.

- Attaching the folder to another folder: `ATTACH`

> **INFO**
>
> To go further, consult the Javadoc:
>
> - Folder
> - Folder classes

Virtual folders are a key concept in FlowerDocs. This type of component defines a folder whose contents are dynamically resolved. Its content is therefore said to be "virtual".

Concepts / Components / The content

The contents of a virtual folder are determined by component searches. The contents of a folder are therefore dynamically resolved according to the user's context. These searches can also be used to organize content into sub-level : the `aggregations`. These are used to aggregate (or group) components found according to their tags.

These searches and aggregations can be modified at any time, so that the organization of your components can evolve according to your needs.

We recommend not aggregating more than 3 levels, using tags with defined values (such as choice list tags) and not exceeding 100 possible values.

When the tree structure of a virtual folder is in `View` mode, we recommend not exceeding 100 documents displayed in ARender.

Adding components

Via the interface, depending on the search parameters set (category searched and `classid` criteria), a `Create a {component category}` is used to create the desired component.

When indexing, it is possible to select the component class. All tags in common are listed on the write indexing form.

Note: If there is only one component class, this is automatically selected.

Permissions

To restrict access or modifications to a virtual folder, several permissions are available to control the operations that can be carried out:

- Download the contents of a virtual folder as a zip archive: `DOWNLOAD_CONTENT`
- Attach the virtual folder to a folder : `ATTACH`

Some actions are only available if the user has write access to the virtual folder. To do this, the permission has to be set on `UPDATE` for this user and this user should have reserved the virtual folder (see Reservation).

> **INFO**
>
> To go further, consult the Javadoc:
>
> - Virtual folder
> - Virtual folder class

INFO

FlowerDocs can be integrated with any workflow engine on the market.

In FlowerDocs terms, a workflow is a logical sequence of several classes of tasks where each step corresponds to a type of task (a class of tasks).

Example: `1. Invoice import` --> `2. Accounting validation -->` 3. Overhead validation --> `4. Payment`

These three stages correspond to three classes of tasks, each of which has two types of response:

- Confirm
- Refuse

Concepts / Components / Attachments

When processing a task, it may be necessary for the user to add one or more components. These documents are called task attachments and are globally defined by task class.

In order to perform actions on attachments, the following permissions may be required:

- Add attachment: `CREATE` for the component class to be created
- Update the content of an attachment: `UPDATE_CONTENT` for the component class to be created
- Delete an attachment: `DELETE_CONTENT`

_Note:__ These actions are not possible if the attachment definition is read-only.

Answers

When a user has finished filling in the necessary information for a task, they can select an answer from among those proposed. These answers enable a workflow to move in one of the proposed directions.

To apply a response to a task, the `APPLY_ANSWER` permission is required.

When applying a response, it is possible via API or WS to add parameters that will not be persisted on the task. This information can then be used within the operation subscriptions activated on the response.

Assignment

To edit a task, the user must:

- have the right to update the task: `UPDATE`
- be assigned to the processing of the task

In order to assign a user to process a task, several means are available. From a technical point of view, all you need to do is fill in the task's `assignee` field with the identifier of the user, group or team to which the task should be assigned.

Appropriation

From a user point of view, appropriating a task means assigning it to oneself. A task can be appropriated if it is not assigned to any user or to a team to which the logged-in user does not belong. This type of assignment requires the `APPROPRIATE` permission.

It is also possible to assign a task to yourself that has already been assigned to another user if you have the `APPROPRIATE_ALREADY_ASSIGNED` permission.

By task class, it is possible to**activate the self-assignment option**. So when a user opens an unassigned task and has the `APPROPRIATE` or `APPROPRIATE_ALREADY_ASSIGNED` right, the task will automatically be assigned to them.

Auto-assignment on task classes is disabled by default.

Assignment

Assigning a task to a user consists in assigning a task from user A to user B. This type of assignment can be made even if the task is already assigned to another user.

This type of assignment requires the `ASSIGN` permission.

> **INFO**
>
> To find out more, take a look at some references:
>
> - Task
> - Task class

FlowerDocs' search capabilities make the most of the document base, while maintaining the ergonomics required by end-users.

The classic search functions of a DMS are described, including:

- selection of fields to be displayed
- search filters
- sorting on metadata
- pagination of results
- etc.

more details can be found in the technical documentation

> **INFO**
>
> We recommend using a maximum of 20 criteria in a search.

Searches can be saved and shared between FlowerDocs users.

Searches return a maximum of 10,000 results; an error occurs when paging after this limit. This limit is the same using user interface and web services.

Concepts / Search / Full-text search

With FlowerDocs, it is possible to configure a full-text search. This allows a user to search on the content of a document rather than on its metadata.

In a search form, you can add the “Content” criterion to perform a full-text search.

The full-text search function is based on the OpenSearch search function, with the following specific features for writing down search items :

- The order of words in the search doesn't matter
  - Search for the following value “{word A} {word B}” retrieves documents containing words A and B, regardless of where words A and B appear in the content.
- There are AND, OR and NOT operators, which allow you to specify what you are looking for :
  - Searching for “{word A} {word B}” retrieves documents containing words A and B, but not documents containing only word A, word B or containing neither A nor B.
  - Searching for “{word A} OR {word B}” retrieves documents containing word A, word B or both words, but does not retrieve documents containing neither word.
  - Searching for “{word A} AND NOT {word B}” retrieves documents containing word A, but not word B.

The following characters must not be used in full-text searches: `+`, `-`, `=`, `||`, `>`, `<`, `!`, `(`, `)`, `{`, `}`, `[`, `]`, `^`, `"`, `~`, `*`, `?`, `:`, `\`, `/`

To find a document containing the name "John-Peter", use the search "John Peter".

INFO

Full-text search works only on documents whose content has been indexed beforehand. The content is indexed using a operation subscription.

Concepts / Search / Stored searches

> **INFO**
>
> It is possible to save a configured search from the user interface.

When a user consults a search, he or she can save it for easier access later. This can be useful in the case of a recurring search, where the user has modified the criteria to better match their regular needs. Once a search has been saved, the user can retrieve it in the tab `Stored searches`.

FlowerDocs lets you save a search with empty criteria, for example, to keep criteria in memory.

Shared searches

> **INFO**
>
> A user can share a saved search with other collaborators.

When a user consults one of his saved searches, he can share it with other collaborators. It is possible to share a search with a user, a group or a team. Collaborators to whom the search has been shared can find it again in their `Shared searches` tab.

Concepts / Security / Principle

Within the FlowerDocs platform, an identity is a user, a group of users or a team. This concept identifies the users who use the platform so that:

- data security is guaranteed according to the authenticated user
- users can collaborate
- **FlowerDocs GUI** 's configuration is adapted to users' needs

These identities are stored in a corporate directory configured by scope or in the [internal users]’ repository (/documentation/config/core/securite/realm.md).

Users

A represents a physical person or a third-party application. Every interaction with the platform must be linked to an authenticated user. A user can belong to a group or a team.

Groups

A represents a set of users or other groups. This notion is generally used to apply specific permissions according to the groups to which a user belongs.

Teams

The notion of is similar to that of a group, except that it is managed and stored by the FlowerDocs platform. This allows you to have user groupings distinct from those defined in the corporate directory. Teams are generally used in organisations where the hierarchy defined in the corporate directory differs from the hierarchy defined in the FlowerDocs platform.

Teams have a list of properties that can be used to configure **FlowerDocs GUI**.

Roles

The FlowerDocs platform offers several native roles with specific permissions. A role can be assigned to a user by defining a team whose identifier is the role name.

Name	Description
`FUNCTIONAL_ADMIN`	Functional administrator *(data model)*
`SECURITY_ADMIN`	Security Administrator *(ACL, identities...)*
`ADMIN`	Scope administrator
`SYSTEM_ADMIN`	Platform administrator

INFO

Several mechanisms can be used to authenticate users (or clients).

Concepts / Security / Company directory

FlowerDocs natively supports the LDAPv2 protocol for user authentication:

- recovery of groups
- special attributes (mail, ...)
- change password

Qualified directories are:

- Microsoft Active Directory
- OpenLDAP
- Apache Directory Server

OpenID Connect

To integrate FlowerDocs with an existing authentication system shared across the information system, FlowerDocs supports the OpenID Connect protocol. This protocol is used to authenticate a user to **FlowerDocs GUI**.

Token

The **FlowerDocs Core** APIs enable user tokens to be generated. The token generated can then be re-used to authenticate requests made to **FlowerDocs Core** or to open **FlowerDocs GUI**.

More information can be found here.

> **INFO**
>
> The role of a security object is to manage the various permissions on objects in FlowerDocs (components, saved searches, scopes, etc.). These security objects are referenced on FlowerDocs objects.

Within the application, we define two types of security objects:

- Access control list (ACL)
- Proxy

These security objects are used to authorise (or prohibit) an identity to perform an action.

Concepts / Security / Security objects

Access control list (ACL)

An access control list contains one or more access control entries (`ACE`).

In each entry, different permissions are defined for a set of identities (user, group, team).

For example, when a user wants read access to a component, the application will evaluate the read permission according to the ACL carried by the component.

> **INFO**
>
> If the user does not belong to any identity defined in the ACL referenced by a component, it is considered that he/she has no access rights to this component.
>
> Component classes carry an ACL which is applied by default when a component of this class is created. This ACL can be modified later via integration.

Order of access control inputs (ACE)

The order in which `ACE` are defined is important, as it corresponds to the order in which they are evaluated. The first entry corresponds to the user, one of his

groups, one of his teams or *, will be the one used to evaluate the various user permissions for a component.

For example, if the first entry has the identity * to view the component, and the second has the identity of the user's identifier X which does not allow the component to be viewed, the user will be able to view this component.

By inverting the two `ACL` entries, all users except user X will be able to see the component.

- When setting up `ACL`, it is advisable not to define entries with the identity *.

Indeed, this would mean that a user who is not part of a group could still have the permissions of the `ACL`. It is preferable to add the name of all groups/profiles as an entity.

- It is recommended not to exceed 1000 ACL on a scope.

ACL Proxy

This feature is in beta. For any integration requirements using ACl's proxies, please contact the FlowerDocs team to help you find the best solution for your needs.

A proxy can be used to define dynamic security based on one or more rules.

Each rule corresponds to a list of conditions and an ACL identifier. If all the conditions of a rule are met, the user's permissions are determined according to the referenced ACL.

Three types of conditions are supported by the application:

- Condition on tags
- Condition on class
- Condition on user

EXAMPLE

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ACLProxy name="Acl proxy for document"
xmlns="http://flower.com/docs/domain/acl"
xmlns:common="http://flower.com/docs/domain/common">
        <common:id>acl-proxy-document</common:id>
        <rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:type="ACLConditionalRule">
                <conditions>${user.authorities}.contains("DSI")
</conditions>

<conditions>${tags.MailType}==Cancellation</conditions>
                <aclId>acl-courrier-ingoing</aclId>
        </rules>
        <rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:type="ACLConditionalRule">

<conditions>${user.authorities}.contains("ACCOUNTING")
</conditions>

<conditions>${data.classid}==IngoingMail</conditions>
                <aclId>acl-courrier-ingoing</aclId>
        </rules>
        <rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:type="ACLConditionalRule">

<conditions>!${user.authorities}.contains("LEGAL")</conditions>

<conditions>${tags.MailType}!=Contract</conditions>

<conditions>${tags.MailType}!=Cancellation</conditions>
                <aclId>acl-courrier-entrant</aclId>
        </rules>
</ACLProxy>
```

> **INFO**
>
> Like ACLs, if none of the rules is satisfied, the user is considered to have no access rights to the component.

Security object Association

A security object is associated with an object through its `ACL` field. This field contains only the ACL identifier. In this way, an ACL can be defined on several components, providing the possibility of managing security policies common to sets of components.

The ACL referenced, for example on a component, will be used to determine whether or not a user is authorised to perform an action on it.

Creating a component is a special case. As this does not yet exist, it is the ACL defined at component class level that is evaluated. To authorise a user to create a component, he or she must have the `CREATE` permission on the ACL referenced at class level.

Concepts / Operation

Principle

The Operation API reacts to the execution of operations within **FlowerDocs Core**. A **operation** is an action performed by a user on a component.

The **operations managers** (or `com.flower.docs.operation.api.OperationHandler`) are called when an operation is executed to react to it and apply specific processing. They can be called before (*pre-treatment*) or after (*post-processing*) the execution of the operation.

The execution of an *operation* can be divided into three phases:

1 . The **OperationHandler** recorded in the pre-processing phase are called

2. The**Operation** is executed

3. The **OperationHandler** recorded in the pre-processing phase are called

> **INFO**
>
> The following steps may be interrupted by an exception raised by an operation handler, if this behavior is enabled.

Subscription

In order for an operation manager to react to the execution of an operation, it must be subscribed to it. Subscribing to the execution of an operation involves creating a configuration document of class `OperationHandlerRegistration`.

The subscription defines the type of operations to which the manager must react by configuring a (creation, update...) and a . Its (*before* or *after*) indicates whether the manager should react before or after execution of the operation.

In addition, the subscription determines whether the reaction to the execution of an operation is synchronous or asynchronous (executed in another thread so as not to block the operation performed by the user).

To limit the number of calls, it is possible to define an execution filter that FlowerDocs will resolve to trigger or not the call to the operation manager depending on the context.

The fields on which filters can be applied depend on the chosen. Here is an exhaustive list :

- **Document, folder and virtual folder** : tag list, component class
- **Tasks** : tag list, component class, assigned user, process identifier
- **Task class** : identifier, process identifier
- **Others** : identifier

Via the administration interface, it is possible to select other fields, notably on administration components: they will not be taken into account.

Operations Manager

An **operations Manager** (or `com.flower.docs.operation.api.OperationHandler`) is a code fragment called when an operation is executed. They can be divided into three categories:

- native : provided natively by **FlowerDocs Core** and executed within its JVM
- specific: developed specifically and added as libraries to **FlowerDocs Core** (*on-premise only*)
- hooks: exposed as REST web services

To contextualize their execution, an object `com.flower.docs.operation.api.OperationContext` is provided as input. The context can be used to retrieve information concerning the execution of the operation, such as the component concerned or the modifications made.

Concepts / Principle

A component's history is made up of a set of facts. Each fact logs an action performed by a user on a component and stores the following information:

Name	Automatic	Description
`id`	yes	Unique identifier
`creationDate`	yes	Completion date
`user`	yes	Identifier of the user who performed the operation
`requestId`	yes	Identifier of the request at the origin of the action
`technical`	yes	Determines whether the fact is technical or business
`action`	no	Action performed
`objectId`	no	Object identifier in question
`objectType`	no	Object type in question

Technical facts

Technical facts are generated automatically by **FlowerDocs Core** when a historical operation is executed. For each component category, fact logging for a given action can be enabled or disabled.

Documents | Tasks | Folders | Virtual folders

Action	Default	Description
`create`	yes	Creation
`read`	no	Access
`get_content`	no	Content access
`update`	yes	Update
`add_content`	no	Adding content
`delete_content`	no	Deleting content
`revert`	yes	Restoring a version
`delete`	yes	Physical removal

To modify historical actions, the `core.properties` file must be modified using the default configuration:

```
fact.registrations.document=create,update,delete,version,revert
fact.registrations.folder=create,update,add_content,delete_content,de
fact.registrations.virtual.folder=create,update,delete
fact.registrations.task=create,update,delete,answer,assign,add_conten
```

Business facts

A business fact is generated programmatically to record a particular state or action for a component. This generation must be configured or developed specifically for the situations concerned thanks to:

- APIs exposed for each component category
- the [ContextUtil] object(/documentation/config/core/appendices/context-util.md)

The user responsible for generating a business fact must have the `ADMIN` role.

In the graphical user interface, a technical fact is linked to a business fact if they have the same request identifier (`requestId`). Technical facts generated before and linked to a business fact are displayed in detail.

HTTP **Script**

```
POST {core}/rest/documents/{id}/facts HTTP/1.1
token: {token}
Content-Type: application/json

    "action": "custom",
    "description": "Generated by REST API.”,
    "updatedFields": [

            "name": "tag",
            "value": "text

    ]
```

Custom Facts

A custom fact is created outside the product's native historical logs. It can be used by integrators to trace specific actions based on their needs. It can be:

- A business fact : Traces a business action (e.g., creating a letter).
- A technical fact : Traces a technical operation (e.g., integration with a CRM).

The distinction is made via a boolean, but the underlying Java object is the same. When creating custom facts via an `OperationHandler`, the user logged in the fact will, by default, be the administrator executing the OH. This is due to administrative privileges that ensure security and prevent unauthorized manipulation.

If you want to log the actual user who initiated the action, you can customize the description.

Example using FactBuilder:

```
var builder =
FactBuilder.objectId(component.getId()).type('DOCUMENT');
var userDisplayName =
util.getUserService().get(...).getDisplayName();
builder.action('CREATE').description(userDisplayName + ' created
the document.');
util.createFact(builder.build());
```

History configuration

FlowerDocs provides a document class `FactFieldsConfiguration` that lets you simply define the tags to be historised on generated facts.

This document allows you to define:

- the object type: `DOCUMENT`, `TASK`, `VIRTUAL_FOLDER`, `FOLDER`
- component class identifiers
- tag identifiers

This configuration document is accessible from the FlowerDocs administration interface: Configuration > Historical facts.

Concepts / Reservations

> **INFO**
>
> A reservation allows you to temporarily give write access to a component in the graphical interface in order to avoid concurrent modifications.

A component is reserved when it is opened in read/write mode by a GUI user. If a reserved component is opened by another user, the form is displayed as read-only.

Reservations are automatically deleted when:

- the user exits the screen from which the component was reserved (by taking an action or closing the browser)
- its session expires
- the user logs out

From the graphical user interface, the current user's reservations can be consulted using:

GET /rest/session/reservations

> **INFO**
>
> Discover these concepts in a simple case study: the **Supplier invoice management**

Concepts / Scenario

A supplier invoice management solution digitally manages the processing of invoices issued by suppliers.

To complicate our case study, let's add a little process! Invoices over €2,500 must be submitted to a third party for validation.

Invoices can be consulted by two departments within the company:

- accounting department: in order to process invoices, a list of invoices to be processed will be presented to users
- the purchasing department: in order to be able to negotiate future orders, users will be able to consult all invoices issued by a given supplier

Solution

Document management

These invoices have a number of features in common:

- A sender
- Issue date
- A part number
- A label
- Amount before tax
- Etc.

In this example, we opt for components of category *Document* since they will have a content (image, PDF...). We choose a single document class *SupplierInvoice* referencing tag classes:

Tag classes	Type	Mandatory
Sender	Character string	Yes
DateEmission	Date	Yes
NumeroPiece	Character string	Yes
AmountHT	Decimal	Yes

The invoice label will be used to fill in the document name

Treatment

In order to notify the accounting department of the arrival of an invoice to be processed, we'll trigger the processing of an invoice processing task when a document of class *SupplierInvoice*. We therefore define the task class *ProcessingInvoice* with the following features:

Processing stage tags

Attachment

So that the accounting department can process the invoice based on the information (and file) in the received invoice, we add an attachment to the task class :

- category: *Document*
- class : *SupplierInvoice*
- mandatory : *Yes*

Replies

The invoices we receive are always perfect, so we add just one answer *Confirm* giving the accounting department only the option of validating the invoice received.

Consultation

Validation

> **INFO**
>
> This type of design can be applied to *(all)* your use cases for electronic document management.

Installation / Presentation layer

FlowerDocs GUI

FlowerDocs GUI is a WEB application providing the presentation layer integrating the ARender viewer interface. It is based on Spring Boot and WEB technologies (HTML, JavaScript...).

Default port : 8080. It can be overridden using the `server.port` property in the `gui.properties` file.

ARender HMI, developed by Uxopian, is a WEB application integrated into **FlowerDocs GUI** as an `iframe`. It visualizes documents generated via ARender Rendition Server.

Default port : 8080. It can be overridden using the `server.port` property in the `application.properties` file.

Service layer

FlowerDocs Core

FlowerDocs Core is the **ECM engine**. It is a Java Web application based on Spring Boot. It can be used to manage very large quantities of documents (add, delete, update documents, search, dynamic folders, etc.). It exposes **SOAP** and **REST** Web Services that are consumed by the various clients:

- the native Web Interface **FlowerDocs GUI**
- third-party applications
- third-party applications, etc.

For data persistence, OpenSearch is used for the metadata part and a storage space for the document content part. Communication with OpenSearch is via its REST API. Document content is stored according to need, and the exchange protocol depends on the technology being studied.

Integration with a corporate directory (LDAP(s)-type) is required to access information linked to the application's users.

Default port : 8081. It can be overridden using the `server.port` property in the `core.properties` file.

ARender Rendition Server

ARender Rendition, a product developed by Uxopian, is the rendition engine used to generate images corresponding to the various pages of the documents to be visualized. This engine exposes a REST API enabling the various elements required for visualization to be retrieved from the client workstation.

Default port : 8761.

Data

OpenSearch

OpenSearch engine, a standalone Java application, is used to index and search data. It provides a distributed, multi-entity search engine, as well as NoSQL document storage. FlowerDocs uses these features to store document metadata in the NoSQL part and advanced search engine capabilities for queries on documents, tasks and folders.

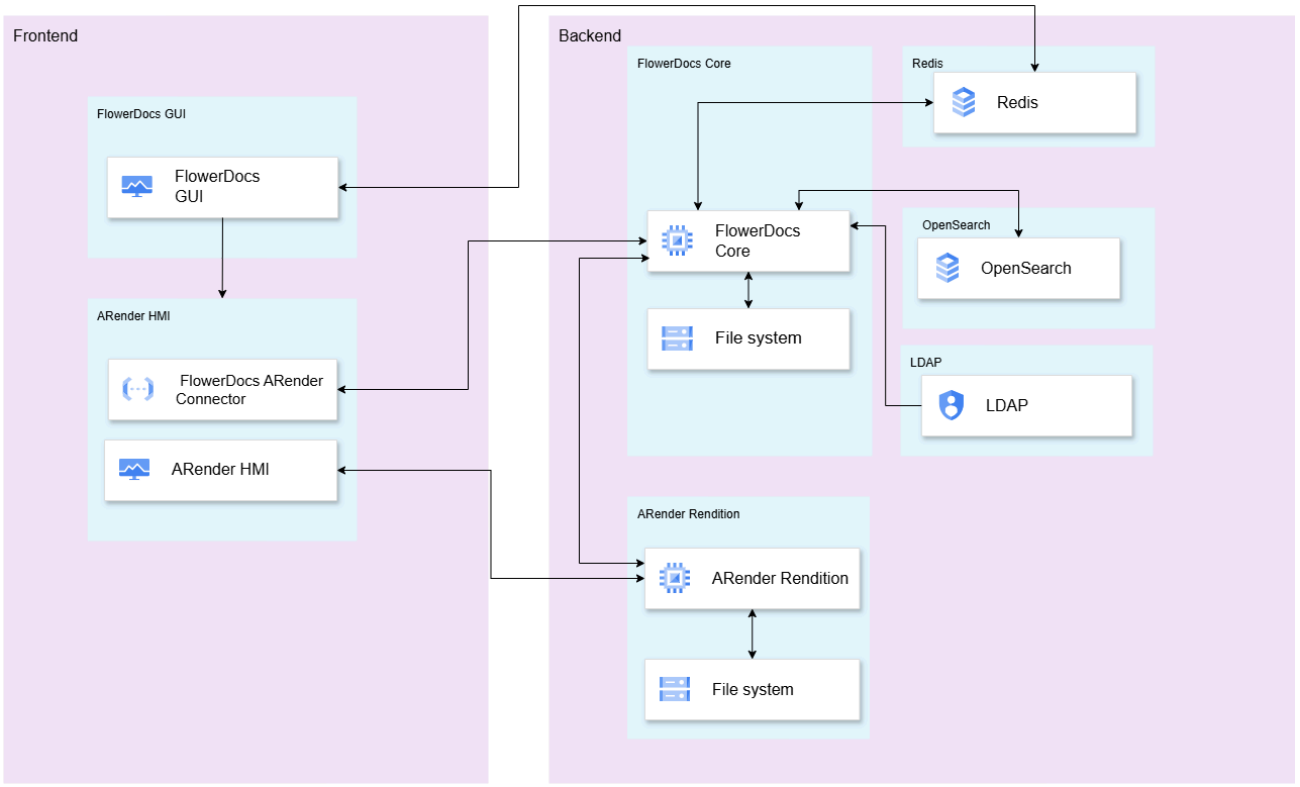
Communication is via the OpenSearch Transport Protocol, based on a specific TCP-based module.

Default port : 9200.

Content storage

The **contents** of the documents (PDF, Microsoft Office Word, etc.) managed by FlowerDocs are not stored in OpenSearch but on a dedicated storage space which can be :

- a NAS-type file system via **NFS**.
- an object storage service such as **AWS S3** via the REST APIs made available.



Redis

Redis is an in-memory database used mainly as a message broker. With FlowerDocs, it is used to manage certain caches as well as for its OperationHandler asynchronous processing queue mechanism.

Default port : 6379.

> **INFO**
>
> This section describes the prerequisites for the **FlowerDocs GUI**, **FlowerDocs Core** and applications.

Installation / Prerequisites

Be careful to harmonise the date/time configuration on the various servers making up the target architecture by using the same timezone (FlowerDocs servers but also all third-party components). Date gap can cause problems when FlowerDocs checks the validity of a token (SSO, Web Services, etc.).

FlowerDocs must only be used in HTTPS.

Operating system

Linux

Installing a FlowerDocs platform requires Linux servers. FlowerDocs is supported and qualified on the Amazon Linux 2023 system.

*Any other Linux system is deemed to be functional if it can install the required version of Java.

Application components

Java Runtime 11 must be installed to run FlowerDocs.

This documentation does not cover the installation of the ARender rendition server (version 2023.15.0). It must therefore be installed beforehand and be accessible via HTTP or HTTPS protocol from **FlowerDocs GUI**, **FlowerDocs Core** and (version 2023.15.0).

OpenSearch and Redis

You need to install OpenSearch and Redis, which are prerequisites for FlowerDocs to work properly. OpenSearch and Redis versions corresponding to FlowerDocs versions are indicated at the beginning of the release notes.

Sizing

The architecture depends on the estimated load, but it is recommended to have at least :

Component	vCPU	RAM	Note
FlowerDocs	2	4 Go	The **FlowerDocs GUI**, **FlowerDocs Core** and applications must be installed on separate servers.
ARender Rendition Server	4	8 Go	Rendition server sizing is strongly linked to the typology and number of documents viewed.

The sizing of a FlowerDocs platform needs to be carefully considered in order to achieve the best possible performance. The most important factors are the number of concurrent users and the number of documents viewed.

We recommend isolating each component on separate machines:

- **FlowerDocs Core**
- **FlowerDocs GUI**
-
- ARender Rendition
- Redis

- OpenSearch

Encoding

It is necessary to use UTF-8 encoding on component installation servers, to ensure proper handling of accented letters and other characters in the application.

Applications

The FlowerDocs platform requires the installation of the following executable JARs:

- `flower-docs-gui-webapp-2025.3.0.jar`
- `flower-docs-core-webapp-2025.3.0.jar`
- `arondor-arender-hmi-spring-boot-2023.15.0.jar`

These applications can be configured by property files located in the same directory as the application.

Applications can be configured via the following files:

Application	Configuration file name
FlowerDocs Core	`core.properties`
FlowerDocs GUI	`gui.properties`
	`arender-custom-server.properties`

The FlowerDocs ARender connector `flower-docs-arender-hmi-2025.3.0.jar` is also required for the application to be able to retrieve documents stored in FlowerDocs.

> **INFO**
>
> The necessary configuration is described on the dedicated page here.

Verification of downloaded application integrity

After downloading the application, check the integrity of the executable before installing it. It is important to check that it has not been altered (corrupted or fraudulently modified `.jar` file).

To do this, you need to calculate the file's hash using the `SHA-256` hash function, so that you can verify it.

- Use the command: `sha256sum {fileName}`.
- Then retrieve the result of the command, which corresponds to the file's fingerprint.
- Check that the fingerprint of each downloaded file is equal to the content of the corresponding `.sha256` file.

Download

> **INFO**
>
> Would you like to download another version? Go to old releases.

> **INFO**
>
> This section describes how to install the **FlowerDocs GUI**, **FlowerDocs Core** and applications.

In the rest of this page, `${APP_HOME}` corresponds to the folder in which each application will be deployed.

Installation / FlowerDocs GUI

- After completing the downloads mentioned in the prerequisites, place the `flower-docs-gui-webapp-2025.3.0.jar` application in the `${APP_HOME}` folder.
- Add `application.properties` file with the following properties:

```
spring.task.execution.pool.core-size=16
spring.task.execution.pool.max-size=16
spring.task.execution.thread-name-prefix=flowerdocs-async-
```

- Add the `gui.properties` configuration file to `${APP_HOME}`.

- After completing the downloads mentioned in the prerequisites, place the `arondor-arender-hmi-spring-boot-2023.15.0.jar` application in the `${APP_HOME}` folder.
- Create the folders `${APP_HOME}/configurations` and `${APP_HOME}/lib`.
- Add the `arender-custom-server.properties` configuration file to `${APP_HOME}/configurations`.
- Add ARender FlowerDocs connector `flower-docs-arender-hmi-2025.3.0.jar` to `${APP_HOME}/lib`.

For high availability, it is necessary to activate session affinity on each HMI.

Please note that Hazelcast in ARenderHMI with FlowerDocs is not functional and should not be activated.

FlowerDocs Core

- After completing the downloads mentioned in the prerequisites, place the `flower-docs-core-webapp-2025.3.0.jar` application in the `${APP_HOME}` folder.
- Add the `core.properties` configuration file to `${APP_HOME}`.

Installation / Manual launch

To manually launch the FlowerDocs GUI, FlowerDocs Core and ARender HMI applications, simply issue the following commands:

GUI Core ARender HMI

```
./flower-docs-gui-webapp-2025.3.0.jar
```

Installation in service

Linux

FlowerDocs applications can be installed simply as a Linux service `init.d` or `systemd`.

In this section, it is assumed that the JARs `flower-docs-gui-webapp-2025.3.0.jar`, `flower-docs-core-webapp-2025.3.0.jar` and `arondor-arender-hmi-spring-boot-2023.14.0.jar` are placed in the `/opt/flowerdocs` folder. This folder is also used as a configuration directory.

Service init.d

To install FlowerDocs applications as an `init.d` service, simply create a symbolic link in the `/etc/init.d` directory:

GUI Core ARender HMI

```
chmod +x /opt/flowerdocs/flower-docs-gui-webapp-2025.3.0.jar
ln -s /opt/flowerdocs/flower-docs-gui-webapp-2025.3.0.jar
/etc/init.d/gui
```

With this type of service, the user to whom the JAR belongs is used to run the application. One log file per application is stored in the `/var/log` directory.

So that the service starts automatically when the system is booted:

GUI Core ARender HMI

```
update-rc.d gui defaults
```

If the service is not found, it may be necessary to run the following command:

```
systemctl daemon-reload
```

Service systemd

To install FlowerDocs applications as a `systemd` service, the `gui.service`, `core.service` and `arender-hmi.service` scripts must be created in the `/etc/systemd/system` directory such as :

GUI | Core | ARender HMI

```
[Unit]
Description=FlowerDocs GUI
After=syslog.target

[Service]
User=flowerdocs
ExecStart=/opt/flowerdocs/flower-docs-gui-webapp-2025.3.0.jar
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```

To have the service started automatically by `systemd`, run the following commands:

GUI Core ARender HMI

```
systemctl enable gui.service
```

JVM configuration

To configure the JVM of the application launched by the Linux service, you need to add a configuration file in the same directory as the JAR. This configuration file must have the same name as the JAR, with the extension `conf`.

Commands

To start the services, simply issue the commands:

GUI Core ARender HMI

```
service gui start
```

Other standard commands are also supported: `status`, `stop` or `restart`.
This section describes the various FlowerDocs GUI configurations to be defined in the application's `gui.properties` file.

Installation / Configuration / General

Property	Description
system.admin.username	System admin username
system.admin.password	System account password, can be encrypted with a secret
token.key	Token shared between **FlowerDocs Core**, **FlowerDocs GUI** and
secret	Secret used to encode password *(optional)*
gui.context	Application context
gui.password.change.enabled	Enables password modification on login page
gui.client.arender.url	ARender HMI URL
scope.edit	Allow to select the target scope on the login page
gui.session.timeout	User session validity time in seconds
ws.url	**FlowerDocs Core** WebServices URL

Logging

Property	Description
logging.file.name	Log file path and name
logging.level.root	Log level: `WARN`, `ERROR`, `INFO`, `DEBUG`

Redis

Property	Description
redis.nodes	Addresses of the various Redis separated by a `,`

It is not recommended to modify Arender properties by setting parameters in the `gui.properties` file. Properties that are not defined in the documentation are not qualified by FlowerDocs: the correct operation of the application is therefore not guaranteed with these modifications.
This section describes the various configurations for the application, needed in the application's `arender-custom-server.properties` file.

Installation / Configuration / General

Propriété	Description
arender.server.rendition.hosts	Addresses of the various ARender renditions separated by a `,`.
token.key	Token shared between **FlowerDocs Core**, **FlowerDocs GUI** and
ws.url	**FlowerDocs Core** WebServices access URL

Logging

In order to configure logging for and the FlowerDocs ARender connector, it is necessary to create a `configurations` folder next to the application containing the following `logback.xml` file:

LOGBACK.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <!-- File Appender for Web-UI server log -->
    <appender name="SERVER"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>arender-hmi-server.log</file>
        <encoder>
            <pattern>%d{ISO8601} %p [%t] %c:%L - %m%n</pattern>
        </encoder>
        <rollingPolicy
class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
            <fileNamePattern>arender-
server.%i.log.zip</fileNamePattern>
            <minIndex>1</minIndex>
            <maxIndex>20</maxIndex>
        </rollingPolicy>
        <triggeringPolicy
class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
            <maxFileSize>50MB</maxFileSize>
        </triggeringPolicy>
    </appender>

    <!-- File Appender for Web-UI perf log -->
    <appender name="PERF"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>arender-hmi-perf.log</file>
        <encoder>
            <pattern>%msg%n</pattern>
        </encoder>
        <rollingPolicy
class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
            <fileNamePattern>arender-
perf.%i.log.zip</fileNamePattern>
            <minIndex>1</minIndex>
            <maxIndex>6</maxIndex>
        </rollingPolicy>
```

```
        <triggeringPolicy
class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
            <maxFileSize>50MB</maxFileSize>
        </triggeringPolicy>
    </appender>

    <!-- Console Appender -->
    <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{ISO8601} %p [%t] %c:%L - %m%n</pattern>
        </encoder>
    </appender>

    <!-- Logger -->
    <logger name="arender-startup" level="INFO" />
    <logger name="com.arondor.common.reflection.parser.spring"
level="OFF"/>
    <logger name="com.arondor.viewer.common.logger"
level="INFO" additivity="false">
        <appender-ref ref="PERF" />
    </logger>
    <logger name="com.arondor" level="INFO">
        <appender-ref ref="SERVER" />
    </logger>
    <logger name="com.flower" level="INFO">
        <appender-ref ref="SERVER" />
    </logger>
    <logger name="com.arondor.common.management" level="FATAL"
/>

    <root level="WARN">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

It is not recommended to modify ARender properties by setting parameters in the `arender-custom-server.properties` file. Properties that are not defined in the documentation are not qualified by FlowerDocs: the correct operation of the application is therefore not guaranteed with these modifications.
This section describes the various FlowerDocs Core configurations to be defined in the application's `core.properties` file.

Installation / Configuration / General

Property	Description
system.admin.username	System account identifier
system.admin.password	System account password, can be encrypted with a secret
token.key	Token shared between **FlowerDocs Core**, **FlowerDocs GUI** and
secret	Secret used to encode password *(optional)*
core.context	Application context

Logging

Property	Description
logging.file.name	Log file path and name
logging.level.root	Log level: `WARN`, `ERROR`, `INFO`, `DEBUG`

OpenSearch

Property	Description
es.nodes	Addresses of the various OpenSearch nodes separated by a `,`
es.cluster	OpenSearch cluster name
es.username	User name *(optional)*
es.password	User password *(optional)*

Redis

Property	Description
redis.enabled	Enables Redis, a prerequisite for high availability
spring.redis.host	Host name Redis
spring.redis.port	Redis listening port

ARender

Property	Description
arender.rendition.nodes	Address of the ARender rendition or a Load Balancer

It is not recommended to modify ARender properties by setting parameters in the `core.properties` file. Properties that are not defined in the documentation are not qualified by FlowerDocs: the correct operation of the application is therefore not guaranteed with these modifications.

Installation / Docker installation

This page describes how to launch a Docker environment, including the prerequisites.

For configuration of OpenSearch, Redis and OpenLDAP applications, please refer to the product documentation for production use.

Prerequisites

- If you are external to Arondor/Uxopian, please contact FlowerDocs support to obtain the Docker images.
- Get the `flower-templates-2025.3.0-package.zip` package.
- Unzip the package

In the `docker` folder, a `.env` file is present. It may be hidden, depending on your system. This file defines the variables used in Docker Compose files.

ARender Rendition installation

- Open the `docker` folder
- Copy the `.env`, `arender-stack.yml` files to the folder of your choice
- Run the following command in this folder: `docker-compose -f arender-stack.yml up`

FlowerDocs installation

- Open the `docker` folder
- Copy the `.env`, `flowerdocs-stack.yml`, `custom-opensearch.yml`, `core.properties`, `gui.properties` files into the folder of your choice.
- Within the `flowerdocs-stack.yml` file, change the LDAP administrator password `LDAP_ADMIN_PASSWORD`.
- Run the following command in this folder: `docker-compose -f flowerdocs-stack.yml --profile="*" up`

For use in a non-development environment, `token.key` and `system.admin.password` must be changed. More details are available here.

Déploiement de scope

The rest of this documentation allows you to start up a FlowerDocs application with a scope that includes the minimal technical model.

Creation

FlowerDocs Core must be started.

- Open the ``flower-templates-2025.3.0-package` folder.
- Run the following command, adapting the value of the `ldapPassword` variable:

```
docker run --volume=${PWD}\default-scope:/clm/default-scope --network=flowerdocs-net artifactory.arondor.cloud:5001/flower-docs-clm:2025.3.0 delete create --template=default-scope --scope=DEFAULT --password=yourPassword --ws.url=http://flower-docs-core:8081/core/services --data.dir=/clm/ --ldap.type=OPENLDAP --ldap.baseDN="dc=flowerdocs,dc=com" --ldap.user="cn=admin,dc=flowerdocs,dc=com" --ldap.url=ldap://openldap-flowerdocs:389 --ldap.password=changeme --ldap.attributes.id=CN --ldap.attributes.search=displayName --ldap.attributes.displayName=displayName --ldap.attributes.members=uniqueMember
```

Access

- Access the **FlowerDocs GUI** url: `http://localhost:8080/gui/?scope=DEFAULT`
- Login as system administrator

    - Username= `system`
    - Password = `yourPassword` or the one you entered in the configuration files.
- To add new users, go to : `http://localhost:8080/gui/?admin=true#/admin:/iam/users`

This section describes the actions to be taken after an application upgrade.

Installation / Label cache

The labels used by the application are stored in a cache in Redis, which is not purged using the cache purge available to administrators within the application.

After an application version upgrade, it is necessary to purge the label cache to take account of modifications directly in Redis, by performing the following actions:

- Connect to the Redis installation server
- Run the Redis client: `redis-cli`
- Run the command if redis is password-protected: AUTH [username] password
- Run the following command: `DEL "labels:*"`

This tool lets you interact with FlowerDocs from the command line. It offers a range of features:

- import
- export
- merge several modules
- administration

Download:

Note: In this section, the extraction file is referred to as `${CLM_HOME}`

Installation / Execution

The CLM is an executable JAR and can be run with a command like:

```
java -jar flower-docs-clm-2025.3.0-bundle.jar
```

To interact with FlowerDocs, you need to define the URL for accessing web services, as well as login details:

```
java -jar flower-docs-clm-2025.3.0-bundle.jar --
ws.url=http://<server>:<port>/<context>/services        --password=
<password>
```

The default user name is `system`, which can be changed by adding the `--USER=<user>` parameter.

The `clm.batch.size=1` parameter sets the batch size for processing jobs in CLM. It can prove useful for resolving conflict issues during updates, as it ensures that each operation is processed individually, thus preventing concurrency errors.

Usage example :

```
java -jar flower-docs-clm-2025.3.0-bundle.jar update --
ws.url=http://<server>:<port>/<context>/services --clm.batch.size=1
[…]
```

> **INFO**
>
> **Note:** In the following sections `<clm>` is the command to run CLM from the command line.

Jobs

A list of jobs is supplied to the CLM to indicate the instructions to be executed. These jobs are supplied after the CLM execution command:

```
<clm> job1 job2
```

The various possible jobs are listed in the following sections.

Installation / Scope initialization / Scope import

The import of a scope is based on a template that serves as a model for the creation of a new scope. This notion of template allows you to initialize different scopes based on the same template.

The location of templates can be defined by supplying the `--data.dir=<path to templates>` parameter.

A template is a set of XML files in a directory whose name is the template identifier. Its structure is based on the following files:

- acls
- classes
  - tag
  - document
  - folder
  - task
  - virtualFolder
- documents
- conf
- folders
- tasks
- tagCategories

- virtualFolders
- workflows
- facts
- scope.xml

Note: *The* `conf` *folder is used to isolate technical documents (such as LDAP configuration, CSS, etc.)*

In configuration XML files for `OperationHandlerRegistration`, `Script`, and `Template`, it is recommended to use the following structure to reference files:

```
<ns3:files size="0">
      <id>Unassign_history.js</id>
      <ns4:Name>Unassign_history.js</ns4:Name>
      <ns4:FormatCode>application/javascript</ns4:FormatCode>
  </ns3:files>
```

This way, referenced files will be correctly interpreted and displayed on the UI.

Create scope with default template

The default FlowerDocs template can be downloaded from the downloads page. It can then be used as a basis for creating a customized scope.

To create a scope from one or more templates, use the CLM, covered in the following pages.

The default template contains a configuration file for the company form : LDAP.xml

This contains configuration variables that can be set directly at scope creation via the CLM. To do this, simply add the following parameters to the command line sent to the CLM :

- --ldap.type=OPENLDAP
- --ldap.baseDN="dc=flowerdocs,dc=com"
- --ldap.user="cn=admin,dc=flowerdocs,dc=com"
- --ldap.url=ldap://openldap-flowerdocs:389
- --ldap.password=yourLDAPPassword
- --ldap.attributes.id=CN
- --ldap.attributes.search=displayName
- --ldap.attributes.displayName=displayName
- --ldap.attributes.members=uniqueMember

Installation / Scope initialization / Create

The CLM can be used to create a scope from a template using the `create` job.

```
<clm> create --template=<template> --scope=<scope> --admin=<admin>
```

Parameters:

Parameter	Description
template	Template identifier (name of the folder in which it is located)
scope	Identifier of the scope to be created
admin	Name of the user who owns the new scope
ws.url	The FlowerDocs CORE url. Example : localhost:8081/core/services
user	The admin account ID
password	The password for the admin account

Delete

To delete a scope, you can use the `delete` job, such as:

```
<clm> delete --scope=<scope>
```

Reset

Jobs can be merged; resetting a scope can be done using the `delete` and `create` jobs:

```
<clm> delete create --template=<template> --scope=<scope>
```

Update

The `update` job is used to update an existing scope.

- Components present in the template but absent from the initial scope will be added.
- Components that differ from the initial scope will be updated.
- Documents present in the initial scope but absent from the template will not be deleted.

This job can be used, for example, to update a scope's configuration without deleting existing documents.

The update can proceed even if the template doesn't include a scope.xml file. In this case, objects that may be configured in this file, as teams, and which have been created after the scope initialisation will not be removed.

Example: Update `HR` scope

```
<clm> update --template=simple --scope=HR
```

There are also more specific update jobs. These allow you to update only a certain type of configuration.

Job	Description
update-config	Updating configuration files

Job	Description
update-model	Update component classes, tags, categories and workflow
update-content	Updating content (documents, folders, virtual folders and tasks)
update-scope	Update scope.xml file (which manages ACLs, display names, scope teams, etc.)

Export

A scope can be exported using the `export` job. This job creates a template from an existing scope.

```
<clm> export --scope=<scope> --template=<template>
```

Parameters:

Parameter	Description
scope	Identifier of the scope to be exported
template	Identifier of the template to be created (name of the folder in which it is located)
ws.url	The FlowerDocs CORE url. Example : localhost:8081/core/services
user	The admin account ID
password	The password for the admin account

By default, the scope will be exported to a `data` folder in the runtime directory.

To change the directory where the export will be stored, add the `--data.dir=<chemin>` parameter to the command `<clm>`

There are also more specific export jobs.

Job	Description
export-config	Export configuration files
export-model	Export component classes, tags, categories and workflow
export-content	Export content (documents, folders, virtual folders and tasks)
export-scope	Export scope.xml file (which manages ACLs, display names, scope teams, etc.)
export-annotations	Export document's annotations

The `export` or `export-config` jobs do not get annotations. Only the `export-annotations` job export it. Before version 2025.2.0, annotations were exported by `export-config` job.

Job list

Only some of the operations can be carried out. Below is a complete list of possible operations:

Import	Export	Merge
scope-import	scope-export	scope-merge
tag-category-import	tag-category-export	tag-category-merge
tag-class-import	tag-class-export	tag-class-merge
document-class-import	document-class-export	document-class-merge
folder-class-import	folder-class-export	folder-class-merge
task-class-import	task-class-export	task-class-merge
workflow-import	workflow-export	workflow-merge
virtual-folder-class-import	virtual-folder-class-export	virtual-folder-class-merge
acl-import	acl-export	acl-merge
technical-document-import	technical-document-export	technical-document-merge
document-import	document-export	document-merge

Import	Export	Merge
folder-import	folder-export	folder-merge
virtual-folder-import	virtual-folder-export	virtual-folder-merge
task-import	task-export	task-merge
facts-import		facts-merge

In addition to these operations, there is a job to purge the caches of the **FlowerDocs Core** part named `purge-cache`.

Example: Import tag classes only

```
<clm> tag-class-import --scope=<scope>
```

INFO

The `force` parameter is used to determine the behaviour to adopt in the event of conflicts between elements in several modules:

- true: elements present in a module overwrite elements in the initial scope
- false (default): elements present in a module are ignored if they are present in the initial scope.

The order followed is the order passed as input to the CLM.

Installation / Scope initialization / Execution

The CLM offers the option of merging several scopes.

```
<clm> merge --template=<template> --modules=<module1>,<module2>
```

Parameters:

Parameter	Mandatory	Description
template	Yes	Identifier of template to be built (name of output folder)
modules	Yes	Names of templates to be merged, separated by commas
force	No	true or false *(default: false)*

Merge rules

Generally speaking, merge follows the following rules:

- If an element is present in one scope and absent in another, it will be present in the merged scope
- If an element is present in several scopes:
  - Some sub-elements are merged (see detailed rules below)
  - Other properties are either ignored or overwritten if defined, depending on the value of the `force` parameter.

Scope

Merge of data in the `scope.xml` file.

The merge is performed as follows:

- If a profile is present in several modules, the list of identities and properties is merged
- 2 properties with the same name but different values are seen as different objects

ACLs

- The list of entries is merged
- In the case of proxy ACLs: the list of proxy ACLs is merged. If a proxy ACL is present in several modules, it is ignored or overwritten depending on the value of the `force` parameter

Component classes

- The list of tag references and tag categories are merged
- If a tag reference is present for the same class in several modules, it is ignored or overwritten depending on the value of the `force` parameter

Task classes

- The list of attachments and replies are merged
- If an attachment is present for a class in several modules, it is ignored or overwritten depending on the value of the `force` parameter
- If a response is present for a class in several modules, it is ignored or overwritten depending on the value of the `force` parameter

Folder classes

- The list of attachments is merged
- If an attachment is present for a class in several modules, it is ignored or overwritten depending on the value of the `force` parameter

Virtual folder classes

- The search list is merged
- If a search is present for a class in several modules, it is ignored or overwritten depending on the value of the `force` parameter

Tag classes

- If a "Choice list" tag class is present in several modules, the lists of values are merged

- Conditional value lists are not merged.

Components

- The tag list is merged
- If a tag is present for the same component in several modules, its value is ignored or overwritten depending on the value of the `force` parameter

Documents

- If a document is present in several modules, its content is ignored or overwritten depending on the value of the `force` parameter

Folders

- If a folder is present in several modules, the list of attachments is merged

Tasks

- If a task is present in several modules, the lists of attachments and participants are merged

Tag categories

- If a profile is present in several modules, the list of identities and properties is merged

Process

- If a process is present in several modules, the list of classes is merged
- Other process properties, such as the first step, are ignored or overwritten depending on the value of the `force` parameter

Facts

- The list of facts is merged
- If a fact is present in several modules, it is ignored or overwritten depending on the value of the `force` parameter

Example

The GEC scope contains a `MailToBeProcessed` task class with an `Incoming mail` attachment. The icon property is set to `icon_A`.

The OutgoingMail scope contains a `MailToBeProcessed` task class with an `outgoing mail` attachment. The icon property is set to `icon_B`.

The merge is performed in GEC-OutgoingMail order.

In all cases, the `MailToBeProcessed` class will have `Incoming mail` and `Outgoing mail` as attachments, because the list of attachments is merged.

If the `force` parameter is `false`, the icon will be set to `icon_A`

If the `force` parameter is `true`, the icon will be set to `icon_B`

Summary

Item	Overloaded / ignored properties	Merged sub-elements
Scope	Data Description Labels Languages Organizational unit Rules file	Profiles (see "Profiles" line)
Profiles	Description Name	Identities properties
ACLs	Description Name	Inputs
All component classes	Data Assets Technical Descriptions Labels Retention period	Tag reference Tag categories
Task classes	Icon process	Replies Attachments
Folder classes		Attachments

Item	Overloaded / ignored properties	Merged sub-elements
Virtual folder classes		Searches
Tag classes	Data Labels Pattern Searchable Type	Value lists
All components	Data Name	Tags
Folders		Attachments
Tasks	Assigned to Reply Process	Attachments Participants
Documents	Content Version Version label Version ID Mime type Parent ID	
Tag categories	Labels Description Icon Folded	Tags

Item	Overloaded / ignored properties	Merged sub-elements
	Aligned Visible	
Process	Labels Description First stage Style	Classes
Facts	All	

Installation / OpenSearch / Getting Started

The OpenSearch connector is based on the version 1.3.19 of OpenSearch.

Direct access to the OpenSearch repository used by FlowerDocs is not recommended. FlowerDocs does not guarantee that the OpenSearch data model will remain unchanged over time, as it may be modified during a release upgrade.

For all data stored in OpenSearch, we recommend that our clients/integrators use FlowerDocs web services, which are maintained over time.

> **INFO**
>
> This documentation is based on the `${FD_HOME}` folder, which contains the application configuration files.

Installation / OpenSearch / FlowerDocs Core

All properties used to configure **FlowerDocs Core** must be added to a `core.properties` file located in the `${FD_HOME}` directory.
This file is loaded when the JVM is started up, so modifications are only taken into account after the application server has been restarted.

```
file.dir=C:\\FlowerDocs\\Files\\
```

FlowerDocs GUI

All properties used to configure **FlowerDocs GUI** must be added to a `core.properties` file located in the `${FD_HOME}` directory.
This file is loaded when the JVM is started up, so modifications are only taken into account after the application has been restarted.

```
ws.url=http://<flower host>:<flower port>/<flower context
path>/services
arender.rendition.nodes=http(s)://rendition-host:8761
```

Installation / OpenSearch / OpenSearch installation

Installation procedure

There are several ways to install OpenSearch, explained step by step in the official documentation.

Configuration

To configure your OpenSearch instance, edit the

`OPENSEARCH_HOME`/config/opensearch.yml` file as follows:

- To define the name of the OpenSearch cluster, uncomment the `cluster.name` property and change its value (eg: `flower-es-dev`)
- In the case of an OpenSearch cluster, uncomment the `node.name` property and change its value to that of your choice (e.g. `node-1`)
- Add the `action.auto_create_index: false` property
- If several OpenSearch nodes are deployed on the same computer, you need to change the value of the `http.port` property
- To access OpenSearch from a remote server, you need to uncomment the `network.host` property and set one of the following values:

- `0.0.0.0`
- DNS name
- IP address

Security

OpenSearch internal users are defined in the `OPENSEARCH_HOME`/plugins/opensearch-security/securityconfig/internal_users.yml` file.

Only keep the users you need, and change their passwords to avoid using default values for security reasons.

To change a user's password:

- run command: `OPENSEARCH_HOME`/plugins/opensearch-security/tools/hash.sh -p <new_password>`
- Replace the password hash in the `OPENSEARCH_HOME`/plugins/opensearch-security/securityconfig/internal_users.yml` file for the desired user.
- in order for the changes to take effect, run the `OPENSEARCH_HOME`/plugins/opensearch-security/tools/securityadmin.sh` script

For more information on securing an OpenSearch instance, official documentation is available here.

Start

Manuel

Finally, start OpenSearch by going to the

`OPENSEARCH_HOME`/bin` folder and running the `opensearch` script.

Service

To install OpenSearch as a `systemd` service, the following file must be created in the /etc/systemd/system directory as follows:

OPENSEARCH.SERVICE

```
[Unit]

```properties
Description=opensearch
Wants=network-online.target
After=network-online.target
```

[Service]

```
RuntimeDirectory=opensearch
PrivateTmp=true
```

WorkingDirectory=$`OPENSEARCH_HOME`

```
User=${OPENSEARCH_USER}
Group=${OPENSEARCH_USER}
```

ExecStart=$`OPENSEARCH_HOME`/bin/opensearch -p $`OPENSEARCH_HOME`/opensearch.pid -q

```
StandardOutput=journal
StandardError=inherit
```
```

Specifies the maximum file descriptor number that can be opened by this process

LimitNOFILE=65536

Specifies the memory lock settings

LimitMEMLOCK=infinity

Specifies the maximum number of processes

LimitNPROC=4096

Specifies the maximum size of virtual memory

LimitAS=infinity

Specifies the maximum file size

LimitFSIZE=infinity

Disable timeout logic and wait until process is stopped

TimeoutStopSec=0

SIGTERM signal is used to stop the Java process

KillSignal=SIGTERM

Send the signal only to the JVM rather than its control group

KillMode=process

Java process is never killed

SendSIGKILL=no

> # When a JVM receives a SIGTERM signal it exits with code 143
>
> SuccessExitStatus=143
>
> [Install] WantedBy=multi-user.target
>
> ```
> 
> ```

To have the service started automatically by systemd, run the following commands:

Enable OpenSearch service

```
systemctl enable opensearch.service
```

Validation

To check that OpenSearch is working properly, go to http://localhost:9200/.

Installation / OpenSearch / Content storage

The OpenSearch connector provides two types of storage for document content (or files).

File system

This connector enables files to be stored on a file system considered local by the JVM (local, NFS, etc.). The directory used can be configured:

Linux

```
file.dir=/opt/FlowerDocs/Files/
```

Amazon S3

The Amazon S3 connector stores document content in an S3 bucket. To use this connector, the following configuration is required:

```
core.services.file.dao=s3
s3.region=<region AWS>
```

Amazon S3 client configuration

Property	Default value	Description
s3.max.connections	100	Maximum number of open HTTP connections
s3.max.error.retry	2	Maximum number of retries for replayable requests (error 5xx)
s3.socket.timeout	100000	Waiting time (in ms) for data to be transferred

By default, the file containing the access and secret key pair must be located in the `${USER_HOME}/.aws/credentials` folder, and the `default` profile used.

Another key file or profile can be used using the following parameters:

```
s3.profile=<profile name>
s3.configFilePath=<path to directory>/<file with key>
```

AWS instance roles can be used by adding the following property:

```
s3.instanceProfile=true
```

A single bucket can be used for all FlowerDocs scopes with the following configuration:

```
s3.bucket.scoped=false
s3.bucketName=<bucket name>
```

The backup & restoration procedure described below is based on the Opensearch snapshot mechanism.

Installation / OpenSearch / Output directory

Set the directory where snapshots will be saved, add to the

`The backup & restoration procedure described below is based on the Opensearch snapshot mechanism.`

`OPENSEARCH_HOME`/config/opensearch.yml` file in the "Paths" section:
`path.repo: ["/mnt/snapshots"]`

Then restart all OpenSearch nodes.

Then run the command:

HTTP

```
PUT /_snapshot/my-snapshots-repository

        "type": "fs",
        "settings": {
                "location": "/mnt/snapshots",
```

To view the information in the created directory, run the command:

HTTP

```
GET /_snapshot
```

Finally, to check that the created directory is functional, run the command:

HTTP

```
POST /_snapshot/my-snapshots-repository/_verify
```

This command lists all nodes where verification has been successful.

Snapshot generation

A directory can contain several snapshots for a single cluster. A snapshot is identified by a unique name within a cluster. The following command creates the `my-snapshot` snapshot in the directory created previously:

HTTP

```
PUT /_snapshot/my-snapshots-repository/my-snapshot?
wait_for_completion=true
```

The `wait_for_completion` parameter specifies whether or not the request should return a response immediately after snapshot initialization (default) or wait for snapshot completion. During snapshot initialization, information on all previous snapshots is loaded into memory, which means that for large directories this can take several seconds (or even minutes).

By default, a snapshot of all open and started indexes is created. However, it is possible to specify only the indexes to be included in the snapshot with the following command:

HTTP

```
PUT /_snapshot/my-snapshots-repository/my-snapshot

        "indices": "index_1,index_2",
        "ignore_unavailable": "true",
        "include_global_state": false
```

To view snapshot information, enter the following command:

HTTP

```
GET /_snapshot/my-snapshots-repository/my-snapshot
```

Finally, you can delete a snapshot with the command:

HTTP

```
DELETE /_snapshot/my-snapshots-repository/my-snapshot
```

Restoration

A snapshot can be restored with the following command:

HTTP

```
POST /_snapshot/my-snapshots-repository/my-snapshot/_restore
```

This command restores all snapshot indexes. However, it is possible to restore only certain indexes with the following command:

An index can only be restored if it is closed. If the index does not exist in the cluster, it is created when the snapshot is restored.

Changing index parameters during restoration

During restoration, it is possible to modify certain index parameters. In the example below, the `index_1` index is restored with 3 replicas and a default refresh interval of 1s:

Restoration to a different cluster

A snapshot is not cluster-specific. A snapshot from cluster A can be restored to another cluster B.

Save the directory containing the snapshot in cluster B and start the restore process.

Pay attention to cluster capacity. The number of indexes available on the cluster must be equal to or greater than the number of indexes in the snapshot. If the cluster is smaller, it is possible to change the index parameters during restoration, for example by reducing the number of replicas.

Snapshot management

Snapshot status

- The list of running snapshots can be viewed with the command:

HTTP

```
GET /_snapshot/_status
```

- The search can be narrowed down to a directory:

HTTP

```
GET /_snapshot/my-snapshots-repository/_status
```

- The status of a particular snapshot can also be viewed:

HTTP

```
GET /_snapshot/my-snapshots-repository/my-snapshot/_status
```

Progression

The `GET /_snapshot/my-snapshots-repository/my-snapshot/_status` command is used to view snapshot information before the snapshot is stopped. This makes it possible to view snapshot information while the snapshot is running, unlike the `GET /_snapshot/my-snapshots-repository/my-snapshot` command, which waits until the snapshot has finished before displaying the information.

Stop

If a snapshot has been executed by mistake, or if the execution is abnormally long, it can be stopped using a delete snapshot operation. The operation will then stop the snapshot before deleting it.

To cancel a restore, the indices being restored must be deleted. All data from deleted indices will also be deleted from the cluster.
To upgrade OpenSearch, you need to install a new OpenSearch cluster in the target version. Next, the following procedure describes how to push data from one OpenSearch cluster to another after creating your scope from the CLM.

Installation / OpenSearch / Launch of re-indexing

In this section, we will ask OpenSearch to populate an index from another remote index.

OpenSearch will first check the request and then return the identifier of an asynchronous task. To do this, run the following request on the target cluster, adapting it to the environments:

```
POST /_reindex?wait_for_completion=false

  "source": {
    "remote": {
      "host": "http://localhost:9200",
      "username": "user",
      "password": "password"
    },
    "index": "<scope>-flower-docs",
    "type": "document",
    "size": 100
  },
  "dest": {
    "index": "<scope>-flower-docs"
```

Note: In this request, `<scope>` is the identifier of the FlowerDocs scope in lower case.

The response returned by OpenSearch is of the type:

```
"task": "<nodeId>:<taskId>"
```

Re-indexing status

As the re-indexing task is asynchronous, it is possible to know its status:

```
GET /_tasks/<nodeId>:<taskId>
```

Depending on the response returned, it is therefore possible to determine whether the re-indexing has been carried out successfully or whether it is still running.

- A `completed` flag indicates whether or not re-indexing has been completed
- The recovered task mentions a `response` or `error` object indicating whether the task was successfully completed

All re-indexing tasks can also be viewed with:

```
GET /_tasks?detailed=true&actions=*reindex
```

When upgrading FlowerDocs from version 2.5 and lower to version 2.6 and its minor versions, it is necessary to migrate the data. The indexing engine used in FlowerDocs 2.6.0 is OpenSearch, a fork of Elasticsearch version 7.10.2. Elasticsearch's internal data model has been overhauled following version 6.0 by removing support for types in indexes. A migration with rewriting is therefore necessary to carry out this version upgrade.

For this purpose, FlowerDocs CLM provides a `job`, responsible for re-indexing scope data from an Elasticsearch 5.2.1 instance into an OpenSearch 1.3.19 instance.

Installation / OpenSearch / Data reindexing

To reindex Elasticsearch data in OpenSearch, the following CLM command must be executed:

```
java -Des.nodes=<URL Cible OpenSearch> -jar <clm> es reindex --source=<URL source Elasticsearch> --scope=<scope cible> --reindex-source=true
```

At a minimum, the following information is required:

- `--scope`: Scope to be re-indexed
- `--source`: Source Elasticsearch
- `-Des.nodes` : OpenSearch target (JVM parameter, not command parameter)

It is necessary to define which urls are allowed for reindexing in the OpenSearch configuration `opensearch.yml` with the property `reindex.remote.whitelist: "host1:port, host2:port"`

Progress monitoring

The reindexing process may take longer or shorter depending on the volume of data to be reindexed. In order to monitor the progress of the process, this utility provides information at regular intervals on the percentage of completion for:

- each Elasticsearch index
- the complete reindexing process

The time interval between each progress point occurs every 20 seconds, and can be configured by adding the `reindexation.sleep` parameter with a value in milliseconds as follows:

```
<clm> es reindex --reindexation.sleep=<intervalle en ms>
```

Authentication management

Source: Elasticsearch

In order to provide the credentials to connect to a secure source Elasticsearch instance, the following parameters must be provided to the command:

- `--user`: Elasticsearch user ID
- `--password`: Elasticsearch user password

```
java -jar <clm> es reindex --user=<Elasticsearch user> --password=
<Elasticsearch password>  <others parameters>
```

Target: OpenSearch

In order to provide the credentials to connect to a secure OpenSearch target instance, the following parameters must be provided to the JVM:

- `es.user`: OpenSearch user ID
- `es.password`: OpenSearch user password

```
java -Des.nodes=<target URL OpenSearch> -Des.user=<OpenSearch user>
-Des.password=<OpenSearch password> -jar <clm> es reindex
<parameters>
```

Eliminating reindexing tasks

By default, all tasks are deleted at the end of command execution. Tasks can be stored for analysis via OpenSearch APIs. To do this, the following parameters must be supplied to the command:

- `delete-tasks=false`, no task is deleted after reindexing.
- `delete-failed-tasks=false`, only successful tasks are deleted, error tasks are retained.

```
<clm> es reindex --delete-tasks=false
<clm> es reindex --delete-failed-tasks=false
```

Index creation in OpenSearch

If indexes already exist in OpenSearch, you can disable their creation by adding the `create-indexes` parameter:

```
<clm> es reindex --create-indexes=false
```

Mapping creation in OpenSearch

If the mappings for each index already exist in OpenSearch, it is possible to disable their update by adding the `update-mappings` parameter:

```
<clm> es reindex --update-mappings=false
```

Configuring customized reindexing scripts

It is possible to override FlowerDocs' internal reindexing scripts by supplying your own scripts. To take them into account, you need to enter the path of a folder containing the scripts, by adding the `reindexation.scripts.path` parameter as follows:

```
<clm> es reindex --reindexation.scripts.path=<path to directory>
```

The reindex file must be named `Documents-reindex.json`.

FlowerDocs Academy / Create your first search form / Goals

The aim of searches is to provide business users with tools tailored to their needs. All the essential information will be at your fingertips.

In this tutorial, we will create an example of a **Client folder search**. This will simply display client folders with their reference, surname and first name.

Definitions

Search forms are pre-configured search templates, which are then made available to end users.

A search has two main parts: firstly, the search form, which allows you to create a query containing your search criteria, and secondly, the search results, which are displayed afterwards.

INFO

When you have completed this training module, you will be able to:

- Configure a search form,
- Perform hidden requests,
- Link a search form to a team.

FlowerDocs Academy / Create your first search form / Create / Advanced search

XML files are used to configure search forms. Use the internal editor to edit the file.

The editor can be accessed via the **Administration > Display > XML menu** :

- Click on the **+** to create a new XML file,
- In the first field, enter **Client folder search** to name the XML file,
- Enter the XML code described in the rest of this tutorial,
- Click on the **Save**button.

Identifier

The first thing to do is to give the search a `dossierClientSearch` unique identifier. This identifier will make it easy to retrieve this search.

This identifier will also be used to associate the search form with the team that will have access to it.

```
        <bean id="dossierClientSearch"
class="com.flower.docs.gui.client.search.ComponentSearchPresenter"
scope="prototype">
        </bean>
```

Note: These two tags correspond to the start and end tags of a search definition. As a result, everything that will be described in the rest of this example must be inserted between these two tags.

Title, description and category

From a technical point of view, the identifier enables the search to be retrieved, but is not visible to users.

In this case, the name, description and category will make it easier to use the advanced search:

- Set the `title` property to its value **Client folders**,
- Describe the property `description` with its value **Search for a client folder by reference, surname or first name.**

```
<property name="title" value="Dossiers Clients" />
<property name="description" value="Rechercher un dossier client
par sa référence, son nom ou son prénom."/>
```

Categorize the search using the `categorieSelectorPresenter` property:

- Enable a single category with the `FakeCategorySelectorPresenter` class,
- Use `VIRTUAL_FOLDER` for client folders.

```
        <property name="categorySelectorPresenter">
                <bean
class="com.flower.docs.gui.client.search.criteria.item.FakeCategorySe
                                <property name="value">
                                <value
type="com.flower.docs.domain.component.Category">VIRTUAL_FOLDER</valu
                                </property>
                </bean>
        </property>
```

The `keywordCriteriaPresenter` keyword search will not be used. Disable this search using the `enabled` property and its value `false`:

```
<property name="keywordCriteriaPresenter">
        <bean
class="com.flower.docs.gui.client.search.criteria.KeywordCriteriaPres
                <property name="enabled" value="false" />
        </bean>
</property>
```

All the properties described in the rest of this page must be placed in the `AdvancedCriteriaPresenter` bean. Define advanced search with the `AdvancedCriteriaPresenter` property:

```
<property name="advancedCriteriaPresenter">
    <bean
class="com.flower.docs.gui.client.search.criteria.advanced.AdvancedCr
/>
</property>
```

Disable the class selector `displayClassSelector` using its `false` value which allows only client folders to be searched as follows:

```
<property name="displayClassSelector" value="false" />
```

To force the opening of the advanced search, add the `forceOpen` property with its `true` value:

```
<property name="forceOpen" value="true" />
```

Set the `addEmptyCriterion` property to `true` so that the user can add criteria:

```
<property name="addEmptyCriterion" value="true" />
```

Define the `showSearchButton` to launch the search:

```
<property name="showSearchButton" value="true" />
```

FlowerDocs Academy / Create your first search form / Create / Criteria

> **INFO**
>
> In order to use a tag as a search criterion, it must be defined as **searchable** when it was created!

Search criteria represent the content of the search form.

The advanced search can be configured to display, by default, criteria that the user only needs to fill in before running the search. This property is placed in the `AdvancedCriteriaPresenter` bean. Criteria are added via the `fixedCriterionPresenters` property:

```
<property name="fixedCriterionPresenters">
        <list>
        </list>
</property>
```

Between the `tags<list>` and `</list>` add your search criteria.

Place the criterion definition at the beginning of the XML body. The `NomAdherentCriterion` search criterion is defined as follows:

- the `NomClient` tag which is a search criterion
- The criterion is a `STRING` character string
- `EQUALS_TO` is the operator that will help the search to find the client

XML CODE: DEFINITION OF CRITERIA

```
<bean id="NomAdherentCriterion"
class="com.flower.docs.gui.client.search.criterion.SimpleCriterio
    <property name="model">
        <bean class="com.flower.docs.domain.search.Criterion">
            <property name="name" value="NomClient" />
            <property name="type">
                <value
type="com.flower.docs.domain.search.Types">STRING</value>
            </property>
            <property name="operator">
                <value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>
            </property>
        </bean>
    </property>
</bean>
```

Note: The definition remains the same for the `RefClientCriterion` criteria and `PrenomClientCriterion`, we just need to change the values of the properties `description`, `nom` and the `id` identifier of the bean.

The different properties of a criterion :

- `name`: (here equal to `NomClient`) the symbolic name of the tag to be used as a criterion.
- `type`: the type of value to be entered in the field (Field type).
- `operator`: the default operator displayed in the search (Operator).

Reference the criteria in the list using the bean identifier:

```
<property name="fixedCriterionPresenters">
        <list>
                <ref bean="RefClientCriterion" />
                <ref bean="NomAdherentCriterion" />
        <ref bean="PrenomClientCriterion" />
        </list>
</property>
```

FlowerDocs Academy / Create your first search form / Create / Hidden request

In the previous section, you saw how to define the form and search criteria. You will now see how to configure the results display.

To do this, you are going to use a hidden request because the user cannot interact with it.

This XML block should be placed in the `ComponentSearchPresenter` bean, at the same level as the `advancedCriteriaPresenter` property.

This hidden search is defined using the `hiddenRequest` property:

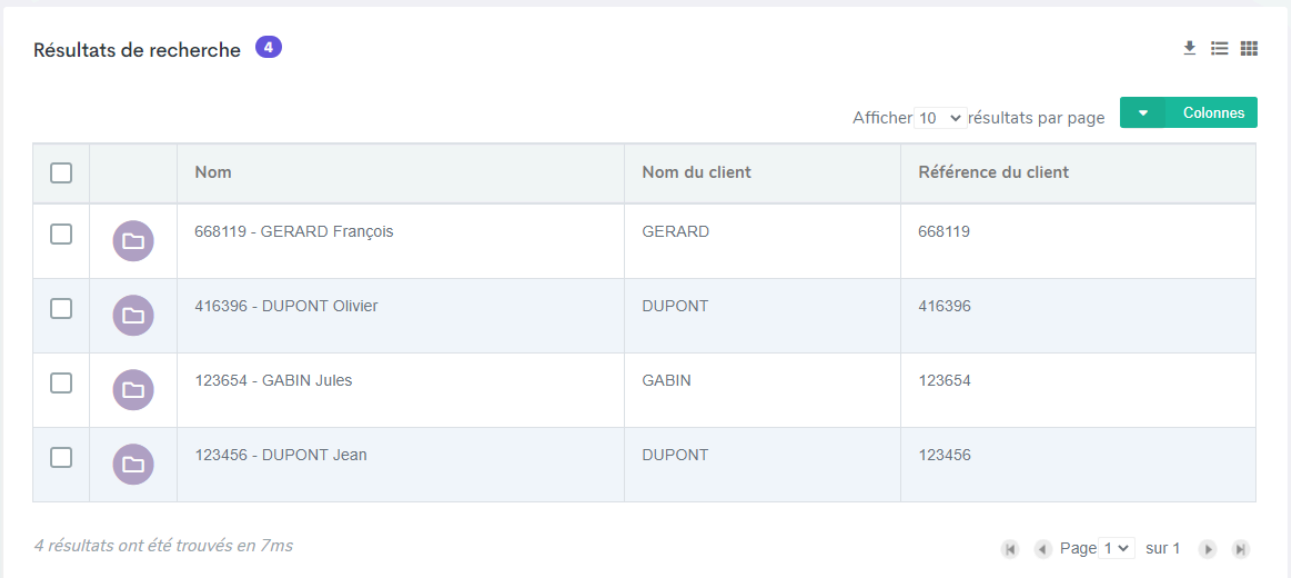
```
<property name="hiddenRequest">
        <bean class="com.flower.docs.domain.search.SearchRequest">
                <!-- Ajoutez ici les différentes parties décrites
dans la suite -->
        </bean>
</property>
```

Columns to be displayed

Place the `SelectClause` property in the `SearchRequest` bean. Set this property as below to display the table columns:

XML CODE: DEFINITION OF COLUMNS TO BE DISPLAYED

```
<property name="selectClause">
        <bean
class="com.flower.docs.domain.search.SelectClause">
                <property name="fields">
                        <list>
                                <!-- Colonnes à afficher -->
                                <value>name</value>
                                <value>NomClient</value>
                                <value>RefClient</value>
                        </list>
                </property>
        </bean>
</property>
```



Filter

Place the `filterClauses` property after the `selectClause` property. Set the `filterClauses` property as below to search client folders:

- The `STRING` type is used to specify that the search criterion will be a character string,
- The `EQUALS_TO` operator will be responsible for client folder feedback,
- the customer files to be searched will be of the `ClientFile` type,

XML CODE: FILTER DEFINITION

```
<property name="filterClauses">
        <list>
                <bean
class="com.flower.docs.domain.search.AndClause">
                        <property name="criteria">
                                <list>
                                        <bean
class="com.flower.docs.domain.search.Criterion">
                                                <property
name="name" value="classid" />
                                                <property
name="type">
                                                        <value
type="com.flower.docs.domain.search.Types">STRING</value>
                                                </property>
                                                <property
name="operator">
                                                        <value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>
                                                </property>
                                                <property
name="values">
                                                        <list>

<value>DossierClient</value>
                                                        </list>
                                                </property>
                                        </bean>
                                </list>
                        </property>
                </bean>
        </list>
</property>
```

Sorting

FlowerDocs allows results to be sorted according to techniques and tags.

In this case, you will sort the result according to the `creationDate` property, which is the date on which the customer files were created.

Use the `orderClauses` property to define the sorting of customer files from most recent to oldest:

XML CODE: DEFINITION OF CUSTOMER FILE SORTING

```
<property name="orderClauses">
        <list>
                <bean
class="com.flower.docs.domain.search.OrderClause">
                        <property name="name"
value="creationDate" />
                        <property name="ascending"
value="false" />
                </bean>
        </list>
</property>
```

The number of results returned by the search. This property allows you to set a default number of results to be displayed per result page:

```
<property name="max" value="25" />
```

FlowerDocs Academy / Create your first search form / Create / Hidden columns

Add this XML block at the same level as the `hiddenRequest`. You can also hide certain results columns to make your search results more elegant:

```
<property name="hiddenColumns">
        <list>
                <value>status</value>
                <value>classid</value>
        </list>
</property>
```

FlowerDocs Academy / Create your first search form / Association

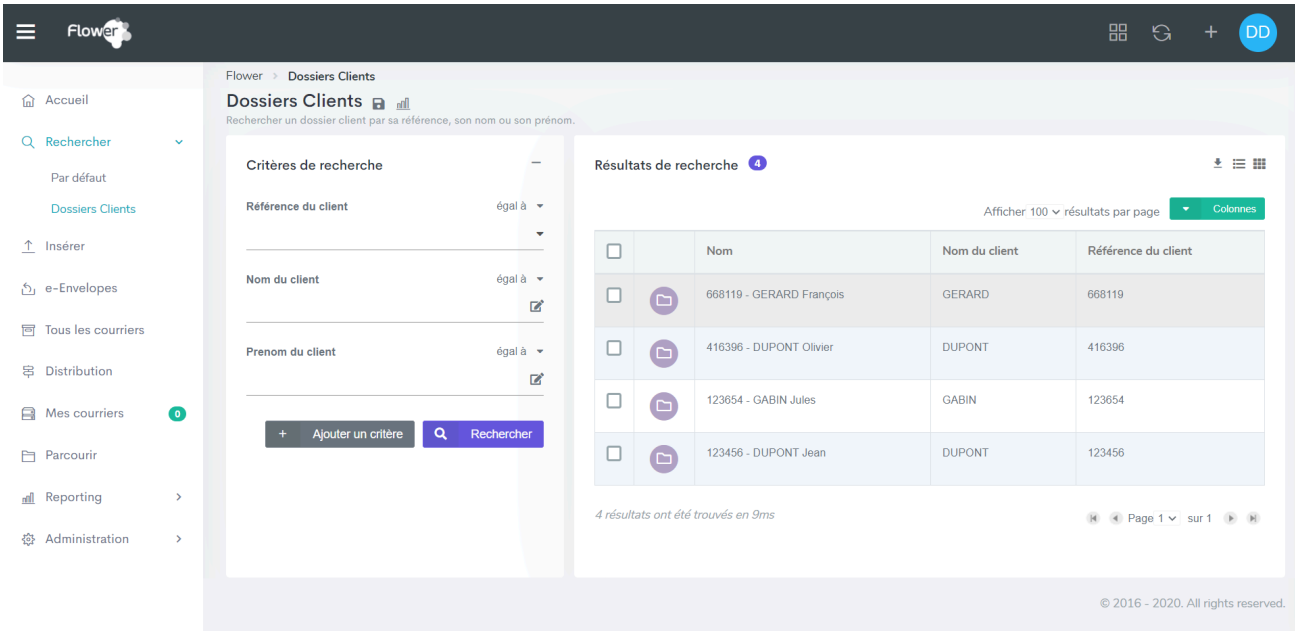
> **INFO**
>
> Link a search to a team of users to make it usable.

To do this:

- Go to the **Administration > Identities > Team**menu,
- Select the user team to which the form should be linked (here `ALL_USERS`),
- In the **Properties**tab, click on the **+** button (at the top of the property list) to create the new property at the bottom of the page,
- Choose **Search template** in the drop-down list,
- Enter the `value` field with **dossierClientSearch(fr=Dossiers Client,en=Client folders)**,
- Click on the **Save**button.

Note: The `Value` field respects a format: `<identifiant>(<langue>=<libellé>,<langue2>=<libellé2>)`. The identifier is the id and the label is its name in the given language.

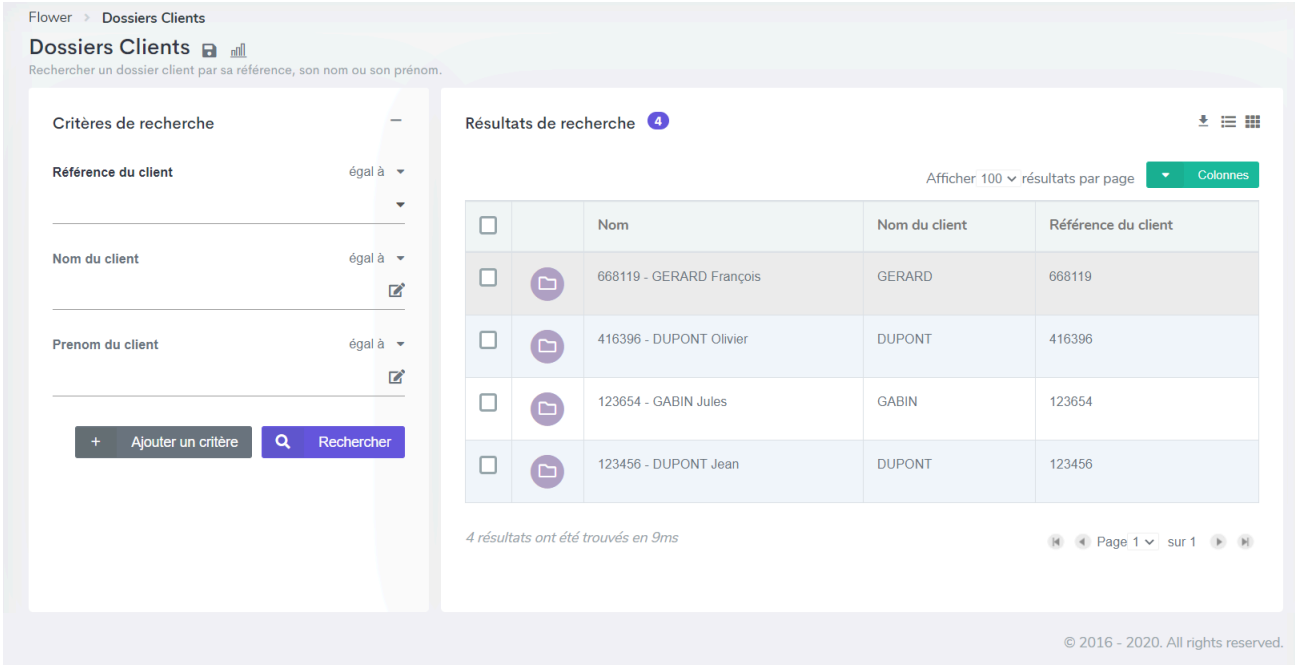
Advanced search is now accessible and usable via the **Search > Client Folders** menu:



FlowerDocs Academy / Create your first search form / Full example

Reminder: Access the editor via the **Administration > Configuration > XML** menu:

- Click on the **+** to create a new XML file,
- In the first field, enter **Search for client folders** to name the XML file
- Copy the XML code below,
- Paste the code into the editor,
- Click on the **Save**button.



FULL EXAMPLE

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema
       http://www.springframework.org/schema/beans/spring-beans.x
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-conte
        <bean id="NomAdherentCriterion"

class="com.flower.docs.gui.client.search.criterion.SimpleCriterio
               <property name="model">
                       <bean class="com.flower.docs.domain.searc
                               <property name="name" value="NomC
                               <property name="type">
                                       <value
type="com.flower.docs.domain.search.Types">STRING</value>
                               </property>
                               <property name="operator">
                                       <value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>
                               </property>
                       </bean>
               </property>
        </bean>
        <bean id="PrenomClientCriterion"

class="com.flower.docs.gui.client.search.criterion.SimpleCriterio
               <property name="model">
                       <bean class="com.flower.docs.domain.searc
                               <property name="name" value="Pren
                               <property name="type">
                                       <value
type="com.flower.docs.domain.search.Types">STRING</value>
                               </property>
                               <property name="operator">
```

```
                                        <value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>
                                </property>
                        </bean>
                </property>
        </bean>
                <bean id="RefClientCriterion"

class="com.flower.docs.gui.client.search.criterion.SimpleCriterio
                <property name="model">
                        <bean class="com.flower.docs.domain.searc
                                <property name="name" value="RefC
                                <property name="type">
                                        <value
type="com.flower.docs.domain.search.Types">STRING</value>
                                </property>
                                <property name="operator">
                                        <value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>
                                </property>
                        </bean>
                </property>
        </bean>
        <bean id="dossierClientSearch"
                class="com.flower.docs.gui.client.search.Componen
                scope="prototype">
                <property name="title">
                        <list>
                                <bean class="com.flower.docs.doma
                                        <property name="language"
                                        <property name="value" va
folders"/>
                                </bean>
                                <bean class="com.flower.docs.doma
                                        <property name="language"
                                        <property name="value" va
Clients"/>
                                </bean>
```

```
                        </list>
                </property>
                <property name="description">
                        <list>
                                <bean class="com.flower.docs.doma
                                        <property name="language"
                                        <property name="value" va
folder by reference, name or first name."/>
                                </bean>
                                <bean class="com.flower.docs.doma
                                        <property name="language"
                                        <property name="value" va
dossier client par sa référence, son nom ou son prénom."/>
                                </bean>
                        </list>
                </property>
                <property name="categorySelectorPresenter">
                        <bean

class="com.flower.docs.gui.client.search.criteria.item.FakeCatego
                                <property name="value">
                                        <value
type="com.flower.docs.domain.component.Category">VIRTUAL_FOLDER</
                                </property>
                        </bean>
                </property>
        <property name="keywordCriteriaPresenter">
                        <bean

class="com.flower.docs.gui.client.search.criteria.KeywordCriteria
                                <property name="enabled" value="f
                        </bean>
                </property>
                <property name="advancedCriteriaPresenter">
                        <bean

class="com.flower.docs.gui.client.search.criteria.advanced.Advanc
">
```

```
                                <property name="enabled" value="t
                                <property name="forceOpen" value=
                                <property name="displayClassSelec
/>
                                <property name="addEmptyCriterion
                                <property name="showSearchButton"
                                <property name="fixedCriterionPre
                                        <list>
                        <ref bean="RefClientCriterion" />
                                                <ref bean="NomAdh
                        <ref bean="PrenomClientCriterion" />
                                        </list>
                                </property>
                        </bean>
                </property>
                <property name="hiddenColumns">
                        <list>
                                <value>status</value>
                                <value>classid</value>
                        </list>
                </property>
                <property name="hiddenRequest">
                        <bean class="com.flower.docs.domain.searc
                                <property name="selectClause">
                                        <bean
class="com.flower.docs.domain.search.SelectClause">
                                                <property name="f
                                                        <list>
                                                                <

<value>NomClient</value>

<value>RefClient</value>
                                                        </list>
                                                </property>
                                        </bean>
                                </property>
                                <property name="filterClauses">
```

```
                        <list>
                                <bean
class="com.flower.docs.domain.search.AndClause">
                                        <property name="criteria"
                                                <list>
                                                        <bean
class="com.flower.docs.domain.search.Criterion">
                                                                <
value="classid" />
                                                                <
name="type">

type="com.flower.docs.domain.search.Types">STRING</value>
                                                                <
                                                                <
name="operator">

type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>
                                                                <
                                                                <
name="values">


<value>DossierClient</value>

                                                                <
                                                        </bean>
                                                </list>
                                        </property>
                                </bean>
                        </list>
                </property>
        <property name="orderClauses">
                        <list>
                                <bean
class="com.flower.docs.domain.search.OrderClause">
                                        <property name="name" val
                                        <property name="ascending
```

```
                                    </bean>
                            </list>
                    </property>
                    <property name="max" value="100" />
                            </bean>
                    </property>
            </bean>
</beans>
```

FlowerDocs Academy / Discovering concepts / Getting Started

> **INFO**
>
> When you have completed this training module, you will be able to use the concepts related to the FlowerDocs data model designed to facilitate:
>
> - understanding,
> - reusability of defined objects,
> - administration of the solution.

The objects manipulated through the interface and APIs are components. Each component is classified according to business or technical criteria using a component class.

> **INFO**
>
> Find out more about the different component categories:
>
> - Documents
> - Folders
> - Virtual folders
> - Tasks

FlowerDocs Academy / Discovering concepts / Component classes

A component class defines the common characteristics of a logical set of components. These sets are characterized by their own tags (or metadata), security, and business or technical rules.

This section defines the notion of component class, used to characterize the components (documents, folders, tasks, etc.) handled within the application. In this way, every component refers to a class of components via its identifier.

Linking a tag to a component

When a tag is referenced on a component class, it can then be characterized with the following parameters:

- Mandatory: indicates whether a value is mandatory for validation
- Technical: indicates whether the user has access to this tag
- Read-only: indicates whether or not the user can modify the tag value
- Multivalued: indicates whether or not the tag can have several values
- Default value: the default value when the tag is not filled (variables can be used for dates, such as `$`dayDate`)
- A validation mask (regular expression): if defined, overrides the one defined in the tag class
- Display order
- A description to display a tooltip

In practice

Each FlowerDocs user can create one or more components according to their needs, for example:

- Create one or more documents.
- Create folders and store documents in them.
- Create tasks for your own needs, and/or assign them to collaborators.

FlowerDocs Academy / Discovering concepts / What is it for?

A class defines a type or family of objects. This notion facilitates the design of FlowerDocs-based solutions by defining a typology of objects that can be reused in a FlowerDocs scope.

Tag classes

`Tags` are metadata that can be defined on a component. They can be used to characterize the product and find it more easily, thanks to filters on these tags.

The tags that can be defined on a component are determined by the existing tag classes. This type of class ensures the consistency of data stored in FlowerDocs. Typically, the amount of an invoice `Amount` is a decimal number; each time this tag is used on a component, the value entered must be a decimal number.

FlowerDocs offers several types of values for a tag:

- String (e.g. reference, contractor) on which a validation mask (or pattern) can be defined
- Decimal number (e.g. amount, tax)
- Integer (e.g. Number of supporting documents)
- Currency
- Boolean (True/False)
- User
- Date (e.g. due date, effective date)
- Choice list (e.g. invoice types)
- Text field (e.g. description, comment)
- Conditional value list
- Choice list

INFO

Add tags to your components by referencing a tag class at component class level.

Component classes

A component class defines the common characteristics of a logical set of components. These sets are characterized by tags (or metadata), security, business or technical rules that are specific to them.

This section defines the notion of component class, used to characterize the components (documents, folders, tasks, etc.) handled within the application. In this way, every component refers to a class of components via its identifier.

A component class defines a type of component:

- tags that can be linked to a component
- the default security to be applied by defining an ACL identifier to be applied
- tag categories to visually group tags into functional blocks
- internationalized labels to provide a multilingual application
- technical nature

Depending on the category of the component class, specifications can be added.

Linking a tag to a component

When a tag is referenced on a component class, it can then be characterized with the following parameters:

- Mandatory: indicates whether a value is mandatory for validation
- Technical: indicates whether the user has access to this tag
- Read-only: indicates whether or not the user can modify the tag value
- Multivalued: indicates whether or not the tag can have several values
- Default value: the default value when the tag is not filled (variables can be used for dates, such as ``$`dayDate```)
- A validation mask (regular expression): if defined, overrides the one defined in the tag class
- Display order
- A description to display a tooltip

> **INFO**
>
> Add tags to your components by referencing a tag class at component class level.
>
> ComponentClass ---→ TagReference ---→ TagClass

FlowerDocs Academy / Discovering concepts / Document content

A document is a component in its own right, with the distinctive feature of having one or more contents. It is therefore a component with a set of tags (or metadata) and a set of content (or files).

These contents can be viewed in the ARender document viewer.

Document permissions

In order to restrict access to, or modifications of, a document's content, a number of permissions are available to control possible operations:

- View content: `READ_CONTENT`
- Add or delete document content: `UPDATE_CONTENT`
- Download document content: `DOWNLOAD_CONTENT`
- Enable printing in the viewer: `PRINT`
- View document annotations: `READ_ANNOTATION`
- Annotate document content: `CREATE_ANNOTATION`
- Document composition: `BUILD_NEW_DOCUMENT`

> **INFO**
>
> Organize your documents by folder for easy retrieval.

FlowerDocs Academy / Discovering concepts / Folder contents

A folder is a component with children. These children can be documents or other folders.

Constraints can be defined at folder class level to restrict the children that can be added to a folder. To do this, it is possible to reference the authorized component classes.

Folder permissions

In order to restrict access to, or modifications of, a document's content, a number of permissions are available to control possible operations:

- Add or delete a folder child from a folder : `UPDATE_CONTENT`
- Detach a document from a folder: `DETACH` (action possible from document indexing)
- Download the contents of a folder as an archive: `DOWNLOAD_CONTENT` Downloaded content contains only the documentary content of the folder and does not include subfolders.

Some actions are only available if the user has write access to the document. To do this, it must have the `UPDATE` permission and have reserved the folder (see Reservation.

- Attach the folder to another folder: `ATTACH`

> **INFO**
>
> To go further, consult the Javadoc:
>
> - Folder
> - Folder classes

INFO

FlowerDocs can be integrated with any workflow engine on the market.

In FlowerDocs terms, a workflow is a logical sequence of several classes of tasks where each step corresponds to a type of task (a class of tasks).

Example: `1. Invoice import` --> `2. Validation comptable` --> `3. Validation frais généraux` --> `4. Payment`

These three stages correspond to three classes of tasks, each of which has two types of response:

- Confirm
- Refuse

FlowerDocs Academy / Discovering concepts / Attachments

When processing a task, it may be necessary for the user to add one or more components. These documents are called task attachments and are globally defined by task class.

In order to perform actions on attachments, the following permissions may be required:

- Add attachment: `CREATE` for the component class to be created
- Update the content of an attachment: `UPDATE_CONTENT` for the component class to be created
- Delete an attachment: `DELETE_CONTENT`

Note: These actions are not possible if the attachment definition is read-only.

Answers

When a user has finished filling in the necessary information for a task, they can select an answer from among those proposed. These answers enable a workflow to move in one of the proposed directions.

To apply a response to a task, the `APPLY_ANSWER` permission is required.

Assignment

To edit a task, the user must:

- have the right to update the task: `UPDATE`
- be assigned to processing the task

In order to assign a user to process a task, several means are available. From a technical point of view, all you need to do is fill in the task's `assignee` field with the identifier of the user, group or team to which the task is assigned.

Appropriation

Appropriating a task means assigning it to oneself. A task cannot be appropriated if it has not been assigned to a user, or to one of the logged-in user's teams. This type of assignment requires the `APPROPRIATE` permission.

It is also possible to assign a task to yourself that has already been assigned to another user if you have the `APPROPRIATE_ALREADY_ASSIGNED` permission.

Assignment

Assigning a task to a user consists in assigning a task from user A to user B. This type of assignment can be made even if the task is already assigned to another user.

This type of assignment requires the `ASSIGN` permission.

INFO

To find out more, have a look at a few references:

- `Task`
- `Task class`

FlowerDocs Academy / Swagger UI / Goal

The aim of this module is to give you the keys to getting to grips with and using Swagger UI to consume the REST web services exposed by **FlowerDocs Core**.

Access

Swagger UI is available from **FlowerDocs Core** at `/swagger-ui/index.html`.
For example, for an online demonstration, Swagger UI is available here.

Authentication

The services exposed by **FlowerDocs Core** require the client to be authenticated, so it is necessary to provide a token for requests executed from Swagger UI.

We will start by generating a user token :

- In the menu bar, select the `Authentication` specification
- Select the `authentication-rest-controller` web service
- Unfold the endpoint `POST /rest/authentication`
- Click on `Try it out`
- In the request editor, fill in the `password`, `scope` and `user` fields

If the information entered is correct, the HTTP response section shows:

- a return code 200
- a JSON token object with its text value and expiry date

Then, to authenticate future requests:

- Copy and paste the text value of the generated token (`value` field)
- Click on the `Authorize` button
- Paste the copied token

Future requests will provide **FlowerDocs Core** with the defined token to authenticate them.

My first request

In this section, we will test our first request using Swagger UI. This request will search for all documents whose name contains the letter `a`.

- In the menu bar, select the `Component` specification
- Select the `document-rest-controller` web service
- Unfold the endpoint `POST /rest/documents/search`
- Click on `Try it out`
- In the request editor, copy and paste the following request:

```
"filterClauses": [{
    "criteria": [{
        "name": "name",
        "operator": "CONTAINS",
        "type": "STRING",
        "values": ["a"]
    }]
}],
"max": 10
```

To execute the search, click on the `Execute` button and consult the results found in the section presenting the HTTP response.

FlowerDocs Academy / Business Folder / Goals

When you have completed this training module, you will be able to:

- Define a virtual folder class,
- Aggregate the contents of a virtual folder from a pivot tag,

Definition

A virtual folder is a folder for which the content (or children) is resolved dynamically. There is no physical link between the virtual file and its children.

These children are determined by one or more searches defined at the class level of the virtual folder. These searches can contain criteria with variables so that search results depend on the tags of the virtual folder or the logged-in user.

Thus two `ClientFolder` class virtual folders can have different contents by adding a `ClientReference` criterion with the `${tags.ClientReference}` value. When a virtual folder is displayed, the `${tags.ClientReference}` variable is replaced by the value of the virtual folder's `ClientReference` tag.

Prerequisites

For this tutorial, a scope with the `learn` module is recommended.
Using the client file as an example, we can test the setting up of a practical example.

FlowerDocs Academy / Business Folder / The client reference

In this example, the client reference is the pivot tag used to make the link between a client file and its contents. To begin, create a string `ClientReference` tag class.

To do this, follow the steps below:

- Go to the admin console,
- Open the `Components > Tag classes` section,
- Click on the `+` button to start creation,
- Select the value type `Character string` then go on to the next step,
- In the form displayed, enter the `ReferenceClient` value as the identifier for this tag class,
- Click on `Create`.

Congratulations! You have created the `ClientReference` tag class, which will be used to classify a client's documents within the same folder.

The client folder

To create instances of client folders, you need to create the virtual folder class `ClientFolder` referencing the `ClientReference` tag class.

To do this, follow the steps below:

- Go to the admin console,
- Open the `Components > Virtual folder classes` section,
- Click on the `+` button to start creation,
- Click on the + button to start creation, In the form displayed, enter the `ClientFolder` value as the class identifier, and select `Security Total Control` as the Security value,
- In the `Tags` tab, click the `Add` button and select the `ClientReference` tag class,
- In the `Search` tab, click `Filters` then `Add` to initiate a new search to link the documents,
- Check that there is an identifier for the search
- Indicate 10 results to be displayed in `Display > Number of results`
- Add a search criterion with the `ClientReference` identifier as its name, the `EQUALS_TO` operator and the variable `${tags.ReferenceClient}` as its value,
- Finally, click `Create`.

Client documents

In order to create instances of client documents, it is necessary to create the `ClientDocument` document class .

To do this, follow the steps below:

- Go to the admin console,
- Open the `Components > Document classes` section,
- Click on the `+` button to start creation,
- In the form displayed, enter the `ClientDocument` value as the identifier for this class and `Security Total control` as the Security value,
- In the `Tags` tab, click the `Add` button and select the `ClientReference` tag class,
- Finally, click `Create`.

FlowerDocs Academy / Business Folder / Validation

To validate the correct operation of what has been set up, start by creating a customer document. To do this, go to the `Insert` tab, insert a file and then click on `Next` to move on to the indexing stage.

In the indexing form, select the `DocumentClient` class and enter a value for the `ReferenceClient` tag (e.g. *1234*).

Next, go to the admin console and create a `ClientFolder` virtual folder class :

- Open the `Components > Virtual folder classes` section,
- Select the `ClientFolder` virtual folder class,
- Click on the `Create an instance` button.

In the indexing form, enter the following information:

- a name for the client folder,
- the value previously entered for the `ClientReference` tag of the document created.

Then click on `Create`.

Your first job file has been created! To validate the data model you have created, open the customer folder you have created and check that it contains

the added document.

> **INFO**
>
> **Resources**:
>
> To facilitate client management between the virtual folder and the documents, use the plugin to link one component to another.

FlowerDocs Academy / Contextualize your administration / Goal

In this module, we will look at how to hide tabs or submenus depending on the current user's profile.

Before getting started

This tutorial is based on retrieving the teams to which the current user belongs via the FlowerDocs JS API.

In this training course we will only use the `ADMIN` or `MANAGER` profiles. It is therefore necessary to have a scope containing these user profiles.

Users belonging to `ADMINs` will have access to all tabs, while `MANAGERs` will not have access to the following tabs:

- Memory diagnostics
- Proxies
- Users
- Directories
- OAuth
- CSS
- XML
- Plugins
- Operations subscriptions
- Historical facts
- Email servers

FlowerDocs Academy / Contextualize your administration / Hiding tabs

The following CSS property hides the element to which it is applied:

```
display: none;
```

We are therefore going to apply the `display: none` style to all the elements we want to hide :

```
.AdminPlace//tools//memory,
        .proxiesTab,
        .usersTab,
        .ldapsTab,
        .oauthTab,
        .stylesheetsTab,
        .xmlsTab,
        .pluginsTab,
        .core,
        .serversTab {
    display: none !important;
}
```

The idea is to add style to the opening of FlowerDocs. To do this, we will add the `canAttach` option to the page:

```
        function hideAdminMenus(styleContent){
                const hideAdminMenusStyle =
document.createElement('style');
                hideAdminMenusStyle.textContent = styleContent;
                document.head.append(hideAdminMenusStyle);
```

Depending on profile

We need to know the profile(s) of the current user to know which tabs will be visible to them. To do this, we are going to use the FlowerDocs JS API:

```
var userAPI = JSAPI.get().getUserAPI();
var profiles = userAPI.getProfiles();
```

Finally, we can hide menus for users with the `MANAGER` profile:

```
function hideAdminMenus(styleContent){
  const hideAdminMenusStyle = document.createElement('style');
  hideAdminMenusStyle.textContent = styleContent;
  document.head.append(hideAdminMenusStyle);

var hiddenMenus = '.AdminPlace\\/tools\\/memory,'
                                        +'.proxiesTab,'
                                        +'.usersTab,'
                                        +'.ldapsTab,'
                                        +'.oauthTab,'
                                        +'.stylesheetsTab,'
                                        +'.xmlsTab,'
                                        +'.pluginsTab,'
                                        +'.core,'
                                        +'.serversTab{display:
none!important;}';
var userAPI = JSAPI.get().getUserAPI();
var profiles = userAPI.getProfiles();
if(profiles.includes('MANAGER')){
  hideAdminMenus(hiddenMenus);
```

FlowerDocs Academy / Customize the application logo / Goal

In this module, you will learn how to configure FlowerDocs in two different ways so that you can use your own logo within the application.

The logo must meet these requirements:

- size: 1000x325 pixels
- format: jpg, png, gif, webp

Before getting started

This tutorial is based on the graphical user interface configuration.

FlowerDocs Academy / Customize the application logo / Set up

Creation of a document containing the logo

For our example, we wi'll use a logo that will be contained within the application. To do this, create a document in the FlowerDocs application and add the logo to the document content.

For the rest of the tutorial, you will keep the ID of the document you have created.

Add customized logo from FlowerDocs administration

The graphical user interface can be configured from the FlowerDocs admin console. Add an XML configuration file containing the `appInfo` bean, which will allow us to override several properties, including our logo.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-
context.xsd">
    <bean id="appInfo"
class="com.flower.docs.gui.client.layout.ApplicationInfo">
        </bean>
</beans>
```

You need to know where your logo is located in order to retrieve it. For the example we have chosen to have it as a document in the application. We can then use FlowerDocs REST services to retrieve it:

```
./rest/documents/`docId`/files/content?index=0
```

We can now add the logoURL property to our bean, setting the value to the location of the desired logo.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-
context.xsd">
    <bean id="appInfo"
class="com.flower.docs.gui.client.layout.ApplicationInfo">
        <property name="logoURL"
value="./rest/documents/`docId`/files/content?index=0" />
        </bean>
</beans>
```

Replace the `docId` with the identifier of the FlowerDocs document containing the logo

> **INFO**
>
> In the same way, you can modify other application properties through this `ApplicationInfo` bean:
>
> - The description with the "description" property
> - The environment with the "environment" property
> - The version with the "environment" property
> - The style with the "styleSheet" property

FlowerDocs Academy / Customize the application logo / Add the property to the application's gui.properties file

In the gui.properties file, add the property `gui.client.app.logo` with the value of the path to the chosen logo.

In order for this modification to be taken into account, it is necessary to restart **FlowerDocs GUI**

> **INFO**
>
> Other properties [defining the deployed instance] can be modified (/documentation/config/gui/getting-started.md)

FlowerDocs Academy / Graphical user interface customization / Goal

The aim of this module is to provide the keys to using the predefined FlowerDocs CSS variables. These variables make it easier to customise FlowerDocs colours.

To improve the accessibility of FlowerDocs and meet RGAA requirements, the colours of the FlowerDocs interface have changed. On the following page, you will learn how to use FlowerDocs variables to revert to the theme prior to these changes.

FlowerDocs versions concerned

From 2.6.0.

Use of variables

CSS variables are defined as follows:

```
--<nom-de-la-variable>: <couleur>;
```

You will need to add this line to the `body` selector in your CSS file.

In a similar way, you can redefine a predefined variable, as in the following example.

Modifying a predefined variable will change the color of all elements using that variable.

```
--btn-primary: #3f3cd6;
```

`btn-primary` is the predefined variable for the color of FlowerDocs' main buttons.

Once a variable has been defined, it can be used with the `var()` function as follows:

```
color: var(--<nom-de-la-variable>)
```

A full example

```
body{
  /*Setting a new variable*/
  --new-variable: #cc3672;
  /*Overriding a predefined variable*/
  --navbar: #6558dd;
  /*Redefining a variable using another predefined variable*/
  --text-sidebar-menu: var(--primary-400);

/*Using the previously defined variable*/
button.btn-primary {
    color: #fff;
    background-color: var(--new-variable);
    border-color: var(--new-variable);
```

FlowerDocs Academy / Graphical user interface customization / Visual change

To meet [RGAA] requirements (https://design.numerique.gouv.fr/accessibilite-numerique/rgaa/), the colors of the FlowerDocs interface have changed.

Based on the use of variables, it is quite possible to get a visual similar to FlowerDocs version 2.5.x. Find out how to do this below.

```
body{
  --btn-success: #1abc9c;
  --btn-success-hover: #159a80;

  --checkbox: #84c7c1;
  --checkbox-after: #26a69a;

  --btn-secondary: #6c757d;
  --btn-secondary-hover: #5a6268;

  --btn-danger: #f1556c;
  --btn-danger-hover: #ee324d;

  --btn-pink: #f672a7;
  --btn-pink-hover: #f44e91;

  --text-sidebar-menu-hover: #00acc1;
  --element-focus: #26a69a;

  --badge-pink: #f672a7;
  --badge-success: #1abc9c;

button.btn-primary.disabled,
button.btn-primary:disabled{
  color: #fff;
  background-color: #6658dd;
  border-color: #6658dd;
  opacity: .65;

button.btn-success.disabled,
button.btn-success:disabled{
  color: #fff;
  background-color: var(--btn-secondary);
  border-color: var(--btn-secondary);
  opacity: .65;

.contextual-menu .menu-item:hover {
```

```
  background-color: #5fbeaa;

.contextual-menu .menu-item:hover .icon:before,
.contextual-menu .menu-item:hover .label {
    color: #FFF!important;
```

FlowerDocs Academy / Component Link / Goal

When you have completed this training module, you will be able to link one component to another using the `LinkFreeListPlugin` plugin.

An example in practice

The example in this module is based on a job application. For application management, it is important that the application references the job offer, so that the purpose of the application can be determined.

Using this plugin, you can use a list of values (`FREE_LIST` tag) to:

- browse all job offers to select the one that matches your requirements,
- access the referenced job offer at any time via an icon.

Data model

To put the example into practice, the following model is required.

The job offer

The starting point for this example, the job offer is a `JobOffer` class task with the following mandatory tags:

- `RH_Service`: list of services concerned
- `RH_Job`: list of job types

The application

The application is an `ApplicationSubmission` class task with the `Lookup` type `RH_OfferLink` tag used to store the link to the job offer to which the candidate is applying.

FlowerDocs Academy / Component Link / Script

To use the component link plugin, add the following script to your scope.

To do this, follow the steps below:

- Go to the admin console,
- Open the `Display > Scripts` section,
- Click on the `+` button to start creation,
- Enter a name for the script,
- Add the following content in the text editor:

```
var suggestOfferLabels = [

        name: "suggestOfferTitle",
                FR: "Consultez l'offre",
                EN: "Consult offer"
        },

        name: "suggestOfferDescription",
                FR: "Accéder aux détails de l'offre d'emploi
publiée.",
                EN: "Access to published job offer's details."

]
var labelsAPI = JSAPI.get().getLabelsAPI();
labelsAPI.setLabels(suggestOfferLabels);

var plugin = new LinkFreeListPlugin({
        toPopulate: 'RH_OfferLink',
        category: 'TASK',
        classId: 'JobOffer',
        tagsToDisplay: ['RH_Service', 'RH_Job'],
        title: labelsAPI.getLabel("suggestOfferTitle"),
        description: labelsAPI.getLabel("suggestOfferDescription")
});
plugin.bind();
```

Results

- To begin with, create a job offer with a department and a position for our applicant to apply for.
- Then create an application: In the indexing form, the `HR_OfferLink` tag contains the job offer created previously, and the values entered for its `HR_Service` and `HR_Job` tags are displayed. You can select the offer and confirm the creation of the application.

Open the application you have created, and you will see that an icon has appeared next to the `HR_OfferLink` tag. Clicking on this icon opens a popup displaying the tags of the referenced (or linked) job offer. The `Open` button takes you to the job offer consultation screen, where you can view both the tags and attachments for this job.

FlowerDocs Academy / Component Link / Enabling creation

When selecting a component to link, the user can be prompted to create a new component. To enable this, the `creation` option accepts indexing phases during which the user can create a new component.

Création d'un composant lié

```
var plugin = new LinkFreeListPlugin({
    toPopulate: "RH_OfferLink",
    category: "TASK",
    classId: "JobOffer",
    tagsToDisplay: ["RH_Service", "RH_Job"],
    title: labelsAPI.getLabel("suggestOfferTitle"),
    description: labelsAPI.getLabel("suggestOfferDescription"),
    creation: { phases: ["INSERT", "MODIFY"] },
});
plugin.bind();
```

Reacting to the selection

When selecting a component offered by this plugin, it may be necessary to react to it in order to access information on the selected component. This information can be used to replicate information about the component being indexed, or to propose additional actions to the user. The `selector` option can be used to provide a function called up when the selected component is selected.

Réagir à la sélection d'un composant

```
var plugin = new LinkFreeListPlugin({
        toPopulate: 'RH_OfferLink',
        category: 'TASK',
        classId: 'JobOffer',
        tagsToDisplay: ['RH_Service', 'RH_Job'],
        title: labelsAPI.getLabel("suggestOfferTitle"),
        description: labelsAPI.getLabel("suggestOfferDescription"),
        selector: function(component){
                console.info('The component '+component.getId()+'
has bee selected!');

});
plugin.bind();
```

FlowerDocs Academy / Processing session / Goals

In this module, you will configure an example of a processing session that can be started from certain virtual folder aggregations. This example allows you to manipulate the processing session so that only tasks with the same priority are opened.

Before getting started

This tutorial is based on the configuration of a processing session via the FlowerDocs JS API

In this tutorial we will be using the following components:

- `Choice list` type `Priority` tag containing the values
  - Urgent
  - Critical
  - Normal
- `Choice list` type `Status` tag containing the values
  - `TOBEPROCESSED`
  - `UNDERWAY`
  - `PROCESSED`
- `Mail` class task containing `Priority` and `Status` tags
- `Mail` virtual folder whose search is configured as follows:
  - Identifier: `SearchMails`
  - Number of results: `100`
  - Aggregation: `Status`
  - Tick: `Priority`
  - Category: `Task`
  - Criterion: `Class equal to Mail`

> **INFO**
>
> Add a tab for easy access to the virtual folder.

FlowerDocs Academy / Processing session / Starting a session based on the context

We will restrict the start of processing sessions. We want only tasks opened from buckets `In process` and `To be processed` of a virtual folder to trigger a processing session.

To do this, we need to know the location when the task is opened, as well as the aggregation selected if the location is a virtual folder location.

DETERMINING THE PLACE AT THE ORIGIN OF THE SESSION

```
var place = session.getPlace();
var aggregation = place.getSelectedLeaf();
// A virtual folder place is a BrowsePlace
if(place.getPlaceType().startsWith('Browse') &&
(aggregation.indexOf('ENCOURS') != -1 ||
aggregation.indexOf('ATRAITER') != -1 )){
    // the aggregation selected when the component was opened
is `In process` or `To be processed`

else{
    // no session should be started
```

We therefore obtain the following script:

STARTING A SESSION RESTRICTED TO A VIRTUAL FOLDER

```
JSAPI.get().batchSession().registerBuilder(function(session,
callback){
        var place = session.getPlace();
    var aggregation = place.getSelectedLeaf();
        // A virtual folder place is a BrowsePlace
if(place.getPlaceType().startsWith('Browse') &&
(aggregation.indexOf('ENCOURS') != -1 ||
aggregation.indexOf('ATRAITER') != -1 )){
                callback.enable(session);
        }else{
                callback.enable(null);

});
```

To go further: manipulate the processing session

We now want to force the session to open only those tasks for which the priority entered is the same as that of the task at the origin of the session. To do this, the user could modify the application search to include this search criterion. However, it is also possible, by configuration, to modify the session request as well as other parameters.

In our example, we will modify our request and add the following priority criterion:

MODIFYING THE SESSION REQUEST

```
var request = session.getRequest();
var filter = new AndClause();
var classCriterion = new Criterion();
classCriterion.setName("Priorite");
classCriterion.setOperator("EQUALS_TO");
classCriterion.addValue(session.getComponentSource().getTagValue(
filter.addCriterion(classCriterion);
request.addFilterClause(filter);
session.setRequest(request);
```

With this script, all the tasks opened in the rest of the processing session will have the same priority:

ACTIVATING THE SESSION WHILE MAINTAINING PRIORITY

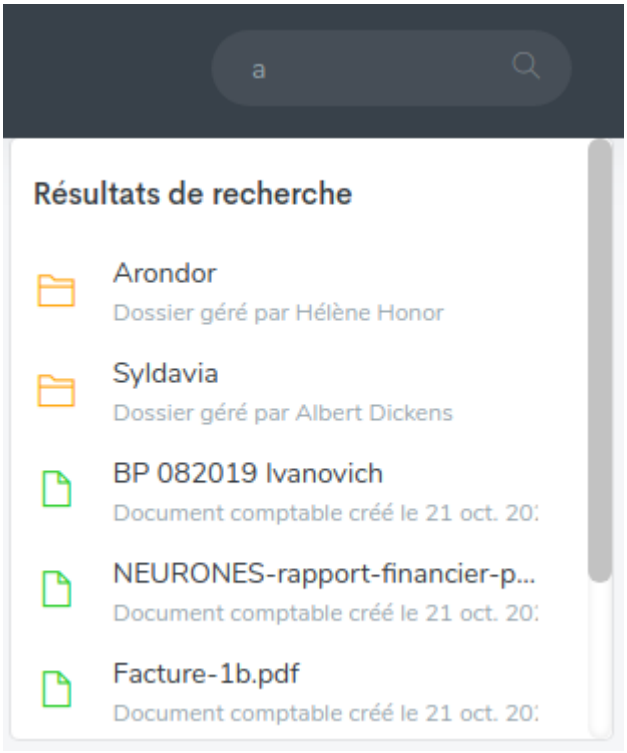
```
JSAPI.get().batchSession().registerBuilder(function(session, call
        var place = session.getPlace();
    var aggregation = place.getSelectedLeaf();
        if(place.getPlaceType().startsWith('Browse') &&
(aggregation.indexOf('ENCOURS') != -1 || aggregation.indexOf('ATR
-1 )){
        var request = session.getRequest();
            var filter = new AndClause();
            var classCriterion = new Criterion();
            classCriterion.setName('Priorite');
            classCriterion.setOperator("EQUALS_TO");

classCriterion.addValue(session.getComponentSource().getTagValue(
            filter.addCriterion(classCriterion);
            request.addFilterClause(filter);
            session.setRequest(request);
                callback.enable(session);
        }else{
                callback.enable(null);

});
```

By modifying the request to add a more restrictive criterion than the original request, when the user processes the last task of that query, the session ends.

FlowerDocs Academy / Search box / Goal



The `SearchBoxPlugin` plugin makes life easier for users by adding a search bar that can be accessed at any time. This search bar can display results from different searches.

An example in practice

In this module, we will configure this plugin to search for a client's folder. To do this, we will use the `ClientFolder` virtual folder class with a *USER* `Accounting` tag.

Set up

In order to display the client folders that match the value entered, we will define a search that finds every virtual folder whose name contains the value entered in the search bar. For each result, we customize the display:

- an orange folder icon is used
- description displays the value of the `Accounting` tag

The `SearchBoxPlugin` plugin can be activated with the script:

```
var searchBox = new SearchBoxPlugin([{
    category: 'VIRTUAL_FOLDER',
    fields: ['name', ‘Accounting'],
    criterion: 'name',
    max: 10,
    icon: function(component, callback){
        callback('orange ti-folder');
    },
    description: function(component, callback){
        var formatter =
JSAPI.get().getHelperFactory().getFieldFormatter('VIRTUAL_FOLDER');
        formatter.format(component.getField(‘Accounting'),
function(formattedValue){
            callback(‘File managed by' + formattedValue);
        });

}]);
searchBox.start();
```

FlowerDocs Academy / Stacked icons / Goal

In this module, we will focus on the notion of stacked icons in order to use more distinctive icons for actions. When you have finished this module, you will be able to create custom icons from several icons in the application.

Before getting started

This tutorial is based on adding actions via the FlowerDocs JS API.

For the purposes of this tutorial, we will be using icons from the FontAwesome icon library, which corresponds to the library available in FlowerDocs.

FlowerDocs Academy / Stacked icons / Goal

For this training module, we want to add a button to the document header menu so that you can write an email to return the document.

Prerequisites

We will use FontAwesome's envelope and pen icons stacked on top of each other for our action

Set up

Creating the stacked icon

Use the following line to combine the two selected icons:

```
stacked(fa fa-envelope-open-text,fa fa-pen)
```

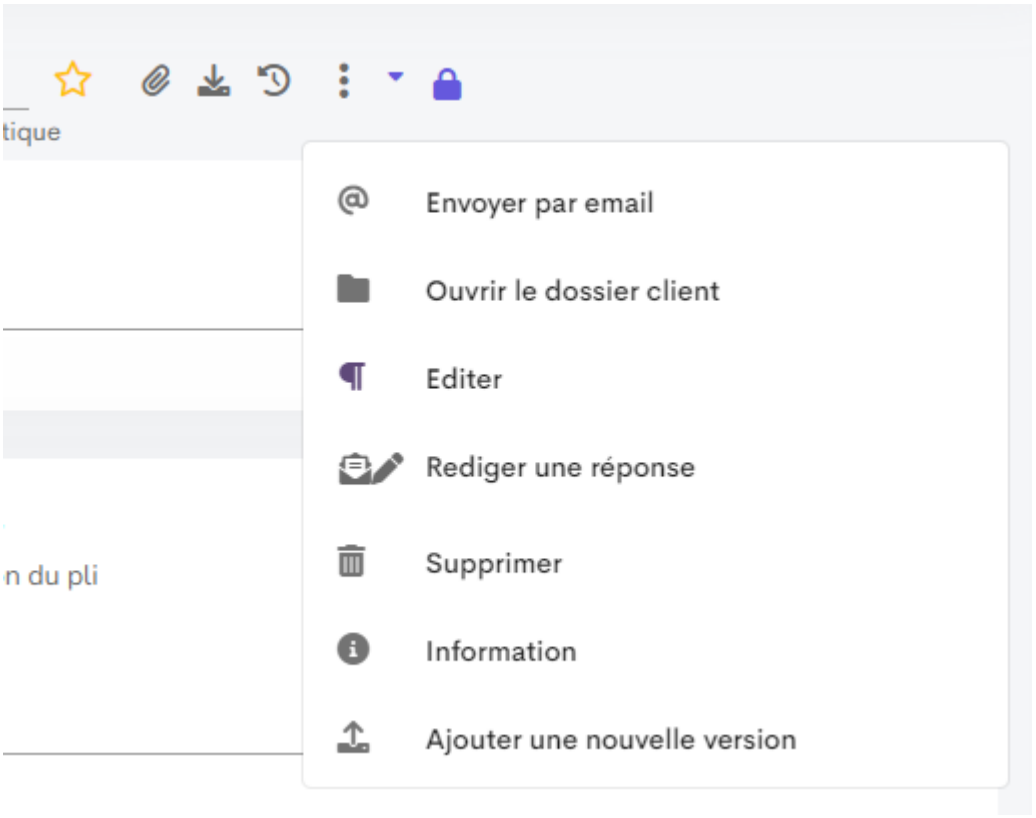
Adding a button to the header containing our icon

Create a script from the **Administration > Display > Scripts** menu and add the following content:

```
JSAPI.get().registerForComponentChange(function(componentFormAPI,
component, phase) {
        if(component.getCategory()=='DOCUMENT'){
                var actionSet =
componentFormAPI.getActions().getHeaderActions();
                var actionAPI = JSAPI.get().getActionFactoryAPI();
                var action = actionAPI.buildResponsive("Rédiger une
réponse", "Rédiger une réponse", "stacked(fa fa-envelope-open-
text,fa fa-pen)",0, function(actionPresenter){
                        //Do custom stuff
                });
                actionSet.add(action)

});
```

You get a new action with an icon like this:



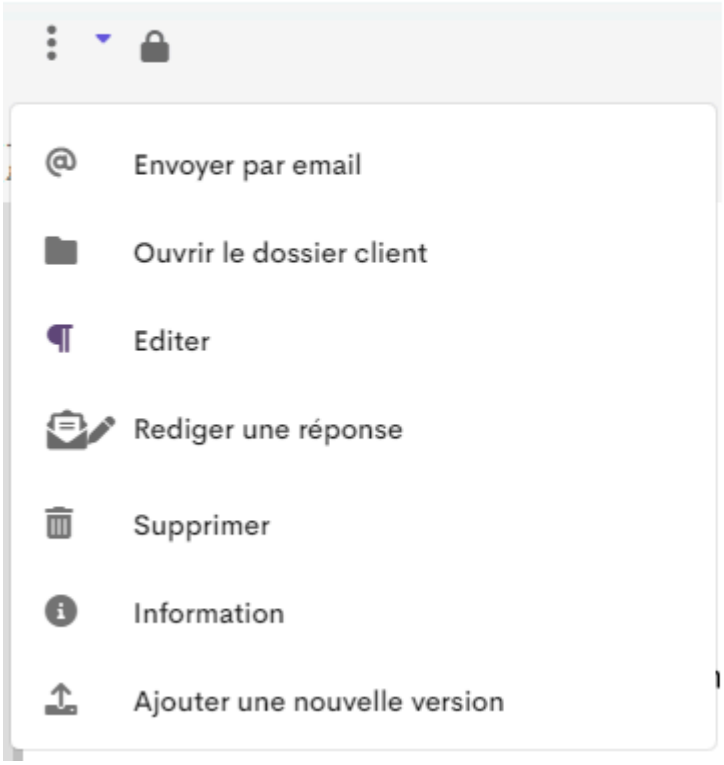
Adapting the size of icons

We now want to reduce the size of the pen in relation to the envelope. To do this, we are going to modify our stacked icon by specifying the size of the different icons:

```
stacked(fa fa-envelope-open-text fa-lg,fa fa-pen fa-sm)
```

> **INFO**
>
> Different sizes are provided by FontAwesome and documented here

All we need to do now is position the pen in the top right-hand corner.

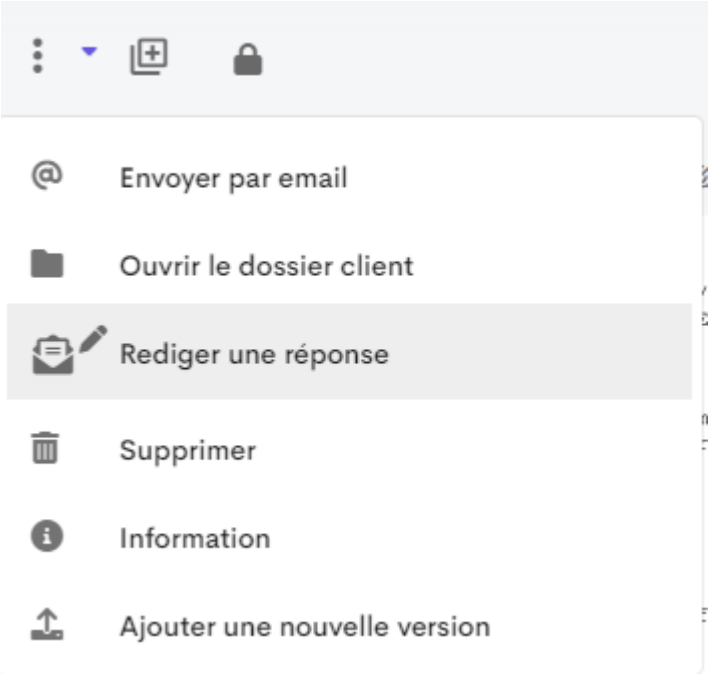


Relative positioning of icons

We are going to add a detail to our stacked icon concerning the positioning of the pen:

```
stacked(fa fa-envelope-open-text fa-lg,fa fa-pen fa-sm align-text-top ml-0.5)
```

We obtain the expected result:



Bonus: a touch of color

Different colours can be set for each icon:

```
stacked(fa fa-envelope-open-text fa-lg green,fa fa-pen fa-sm align-text-top ml-0.5 red)
```

FlowerDocs Academy / GUI Plugin / Goal

The aim of this training module is to lay the foundations for the development of a secured GUI plugin. This plugin can be consumed from and through **FlowerDocs GUI**. It will expose a service consuming the services exposed by **FlowerDocs Core**.

Prerequisites

This tutorial is based on Maven and requires the use of the Arondor Artifactory in which the FlowerDocs libraries are implemented. If you are extern to Arondor, please ask the FlowerDocs support to get the mentioned librairies.

Set up

Project creation

Using your favorite IDE, start by creating a new Maven project with the following POM:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        &lt;modelVersion&gt;4.0.0</modelVersion>

    &lt;groupId&gt;com.flower.docs.samples</groupId>
    &lt;artifactId&gt;secured-gui-plugin</artifactId>
    &lt;version&gt;0.0.1-SNAPSHOT</version>

        &lt;dependencies&gt;
                &lt;dependency&gt;
                        &lt;groupId&gt;com.flower.docs</groupId>
                        &lt;artifactId&gt;flower-docs-starter-
client</artifactId>
                        &lt;version&gt;2025.3.0</version>
                </dependency>
        </dependencies>

        &lt;build&gt;
                &lt;plugins&gt;
                        &lt;plugin&gt;

&lt;groupId&gt;org.springframework.boot</groupId>
                                &lt;artifactId&gt;spring-boot-
maven-plugin</artifactId>
                                &lt;version&gt;2.7.18</version>
                                &lt;executions&gt;
                                        &lt;execution&gt;
                                                &lt;goals&gt;

&lt;goal&gt;repackage</goal>

&lt;goal&gt;build-info</goal>
                                                </goals>
                                        </execution>
                                </executions>
```

```
                                &lt;configuration&gt;
&lt;executable&gt;true</executable>
                                </configuration>
                        </plugin>
                </plugins>
        </build>
</project>
```

```
## Spring Boot application

This Spring Boot application is based on the [Spring Boot starter]
(/docs/flowerdocs/apis/core/java) provided by FlowerDocs.

To start with, we need a _main class_ Spring Boot annotated with
the `@SpringBootApplication` annotation:

```java
package com.flower.samples;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

import com.flower.docs.FlowerDocsClient;
import com.flower.docs.SecurityMode;

@SpringBootApplication
@FlowerDocsClient(security = SecurityMode.TOKEN)
public class SecuredGUIPlugin

        public static void main(String[] args)

                SpringApplication.run(SecuredGUIPlugin.class,
args);
```

Here, the `@FlowerDocsClient` annotation automatically configures:

- the Java client for consuming the web services exposed by **FlowerDocs Core**
- secure requests: a token is required to consume the `@RestController` exposed by the application
```

To configure the application, this `application.properties` file in the `src/main/resources` directory of your project.

Configuration

```properties
spring.application.name=secured-gui-plugin
server.port=2802
server.servlet.context-path=/secured
ws.url=http://localhost:8081/core/services
```

````
:::info
The other configuration options offered by the Spring Boot
framework can also be used.
This makes it possible to outsource application configuration.
:::


## Web service development

In this example, the web service will count the number of documents
accessible by the logged-in user.
The web service will be implemented using the `@RestController`
Spring annotation.

```java
package com.flower.samples;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import com.flower.docs.domain.exception.FunctionalException;
import com.flower.docs.domain.exception.TechnicalException;
import com.flower.docs.domain.search.SearchRequest;
import com.flower.docs.service.api.document.DocumentService;

@RestController
public class FlowerRestController

        @Autowired
        private DocumentService documentService;

        @GetMapping("/count")
        public String count() throws TechnicalException,
FunctionalException

                return "documents: " + documentService.search(new
````
````

```
SearchRequest()).getFound();
```

Here, the `documentService` document management service is used to determine the total number of documents.

FlowerDocs Academy / GUI Plugin / Plugin configuration

Using your favorite browser, open the FlowerDocs admin console, then:

- go to `Configuration` section
- open the `Plugins` menu
- click on the `+` button to start creating a new plugin
- fill in the requested information:
  - Path: `/my-plugin/**`
  - URL: `http://localhost:2802/secured`

Plugin access

Now that your GUI plugin has been configured, you can access the `/count` endpoint through the GUI via the URL: `<gui>/plugins/<scope>/my-plugin/count`.

You can also test the return of the implemented service with different users to observe that the number of documents found depends on the logged-in user.

> **INFO**
>
> Since the plugin requires a token, direct access via the URL `http://localhost:2802/secured/count` is prohibited.

> **INFO**
>
> Not directly linked to the module, this section allows you to go further in implementing a FlowerDocs client.

FlowerDocs Academy / GUI Plugin / Validation type

The `@FlowerDocsClient(security = SecurityMode.TOKEN)` annotation secures requests received by the application by requiring a user token. By default, a received token is validated by querying **FlowerDocs Core** but this behaviour can be changed thanks to the `token.validation.type` property which accepts the values:

- `none`: no token validation at client level. The token supplied must be legible, but its signature is not validated. This mode should therefore be used with caution, for example when all requests received by the FlowerDocs client are forwarded to **FlowerDocs Core**. In this case, the token will be validated directly by **FlowerDocs Core** when requests are received.
- `local`: token validation is carried out by the customer. This mode requires the client to have the secret key (`secret.key`) used to verify the tokens.

Customized safety

In this module, an HTTP filter intercepts each request received and initializes the user context based on the token provided. Custom security mode (`SecurityMode.CUSTOM`) gives developers the freedom to implement their own authentication mechanism.

The security configuration below, for example, allows you to implement your security management by defining an HTTP filter to intercept incoming requests.

```
package com.flower.samples;

import javax.servlet.Filter;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.Order;
import org.springframework.security.config.annotation.web.builders.Ht
import org.springframework.security.config.annotation.web.configurati
import
org.springframework.security.config.annotation.web.configuration.WebS
import org.springframework.security.config.http.SessionCreationPolicy
import org.springframework.security.web.authentication.www.BasicAuthe

import com.flower.docs.security.authentication.TokenAuthenticationFil

@Configuration
@Order(2)
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter

    @Override
    public void configure(HttpSecurity http) throws Exception

        // @formatter:off
        http
            .addFilterBefore(customFilter(), BasicAuthenticationFilte
            .authorizeRequests().anyRequest().authenticated();
        // @formatter:on

    @Bean
    Filter customFilter() throws Exception

        // Return custom HTTP filter
```

FlowerDocs Academy / Customised library / Goal

In this module, we will develop a Java library enabling us to override a native **FlowerDocs Core** behavior: user token generation during authentication.

In fact, we are going to add to each user token a custom attribute named `custom`.

Library development

Maven project

To start developing the custom library, first open your favorite IDE, then create a new Maven project from the following POM. In this module, we will need the following two dependencies:

- Spring Boot: to take advantage of the framework's configuration mechanisms, so that the library can be used by **FlowerDocs Core**
- FlowerDocs Core Services to override the default token generator used by the service layer

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        &lt;modelVersion&gt;4.0.0</modelVersion>
    &lt;groupId&gt;com.flower.docs.samples</groupId>
        &lt;artifactId&gt;flower-docs-lib-sample</artifactId>
    &lt;version&gt;0.0.1-SNAPSHOT</version>
        &lt;dependencies&gt;
                &lt;dependency&gt;

&lt;groupId&gt;org.springframework.boot</groupId>
                        &lt;artifactId&gt;spring-boot-starter-
web</artifactId>
                        &lt;version&gt;2.7.18</version>
                </dependency>
                &lt;dependency&gt;

&lt;groupId&gt;com.flower.docs.core</groupId>
                        &lt;artifactId&gt;flower-docs-
services</artifactId>
                        &lt;version&gt;2025.3.0</version>
                </dependency>
        </dependencies>
</project>
```

````
  </TabItem>
</Tabs>


## Token generator

In this module, we want to add a `custom` attribute to users
authenticating to FlowerDocs.
To do this, we will override the `JWTTokenHelper` token generator
used by **FlowerDocs Core** by default.

<Tabs>
  <TabItem value="customtokenhelper-java"
label="CustomTokenHelper.java">

```java
package com.flower.samples;

import com.flower.docs.domain.security.AuthenticatedUser;
import com.flower.docs.domain.security.IdentityAttribute;
import com.flower.docs.domain.security.Token;
import com.flower.docs.security.token.JWTTokenHelper;
import com.google.common.collect.Lists;

public class CustomTokenHelper extends JWTTokenHelper

    @Override
    public Token generate(AuthenticatedUser user, long
validityTime)

        user.getAttributes().add(new IdentityAttribute("custom",
Lists.newArrayList("value")));
        return super.generate(user, validityTime);
````
````

Auto-configuration

In order for the library to define the previously added token generator, we are going to use Spring Boot's Auto-configuration mechanism. Our configuration class will automatically instantiate a `CustomTokenHelper` object and supply it to the Spring Beans context.

SampleLibAutoConfiguration.java

```
package com.flower.samples;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration
public class SampleLibAutoConfiguration

    @Bean
    @Primary
    CustomTokenHelper customTokenHelpder()

        return new CustomTokenHelper();
```

For this configuration class to be loaded automatically by the Auto-configuration mechanism, it must be referenced in a `META-META-INF/spring.factories` file. With Maven, this file must be placed in the `src/main/resources` directory of your project.

src/main/resources/META-INF/spring.factories

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.flower.samples.SampleLibAutoConfiguration
```

Compilation

To compile the library, use your IDE with the `clean` and `install` goals or run the `mvn clean install` command at the root of your project.

Modification of the classpath

Linux service

When the Core is started as a Linux service (as shown here), the `libs` folder is added to the JVM classpath launched by the service. All JARs included in this directory will be added to the JVM classpath.

All you need to do is copy the library into this directory with the same user as the one with whom the service is launched (or who has read rights on the file).

Executable JAR

When the Core is started as an executable JAR with a command such as `java -jar`, the custom library must be manually added to the JVM classpath. To do this, use the `-cp` parameter.

Test

- Generate a token (e.g. using Swagger `/core/swagger-ui/index.html`) using the:
  - Scope
  - User identifier
  - Password
- From the generated token:
  - Open a browser and go to jwt.io
  - Paste the generated token into the *Encoded* field

The `custom` attribute added by the `CustomTokenHelper` object is indeed present in the attributes added to the token.

> **INFO**
>
> This tutorial is based on Maven and requires the use of the Arondor Artifactory in which the FlowerDocs libraries are implemented. If you are extern to Arondor, please ask the FlowerDocs support to get the mentioned librairies.
>
> When you have completed this training module, you will be able to develop a secure `OperationHook` to respond to the creation of documents and rename them automatically.

FlowerDocs Academy / Operation Hook / Project creation

Using your favorite IDE, start by creating a new Maven project with the following POM:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        &lt;modelVersion&gt;4.0.0</modelVersion>

    &lt;groupId&gt;com.flower.docs.samples</groupId>
    &lt;artifactId&gt;modify-operation-hook</artifactId>
    &lt;version&gt;0.0.1-SNAPSHOT</version>

        &lt;dependencies&gt;
                &lt;dependency&gt;
                        &lt;groupId&gt;com.flower.docs</groupId>
                        &lt;artifactId&gt;flower-docs-starter-
client</artifactId>
                        &lt;version&gt;2025.3.0</version>
                </dependency>
        </dependencies>

        &lt;build&gt;
                &lt;plugins&gt;
                        &lt;plugin&gt;

&lt;groupId&gt;org.springframework.boot</groupId>
                                &lt;artifactId&gt;spring-boot-
maven-plugin</artifactId>
                                &lt;version&gt;2.7.18</version>
                                &lt;executions&gt;
                                        &lt;execution&gt;
                                                &lt;goals&gt;

&lt;goal&gt;repackage</goal>

&lt;goal&gt;build-info</goal>
                                                </goals>
                                        </execution>
                                </executions>
```

```
                                &lt;configuration&gt;

&lt;executable&gt;true</executable>
                                </configuration>
                        </plugin>
                </plugins>
        </build>
</project>
```

```
# Spring Boot application

This Spring Boot application is based on the [Spring Boot starter]
(/docs/flowerdocs/apis/core/java) provided by FlowerDocs.

* To start with, we need a _main class_ Spring Boot annotated with
the `@SpringBootApplication` annotation:


package com.flower.samples;

```javascript
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import com.flower.docs.FlowerDocsClient;
import com.flower.docs.SecurityMode;
```

@SpringBootApplication @FlowerDocsClient(security = SecurityMode.USER)
public class ModifyHookApplication
```

```
    public static void main(String[] args)

        SpringApplication.run(ModifyHookApplication.class, args);
```

- Then add the `application.properties` file to your project's `src/main/resources` directory to configure the application:

```
spring.application.name=modify-hook
server.port=7777
server.servlet.context-path=/modify
```

ws.url=http://localhost:8081/core/services

```
internal.realm.users[0].id=<user>
internal.realm.users[0].password=<password>
```

1 Spring Boot application name

2 Port used to expose WEB application

3 Application path. The application can be accessed via the basic URL *http://localhost:7777/modify*

5 URL for accessing the web services exposed by FlowerDocs Core.

7 User identifier for accessing the `OperationHook`.

8 User password for accessing the `OperationHook`

Hook development

Now we move on to `Operation Hook`! To implement your first hook, create a `ModifyOperationHook` class such as:

package com.flower.samples;

```
import org.springframework.web.bind.annotation.RestController;
import org.terracotta.statistics.Time;
```

```
import com.flower.docs.domain.component.Component;
import com.flower.docs.domain.exception.FunctionalException;
import com.flower.docs.domain.exception.TechnicalException;
import
com.flower.docs.operation.api.DefaultComponentOperationContext;
import com.flower.docs.operation.api.OperationContext;
import com.flower.docs.operation.api.OperationHook;
```

@RestController public class ModifyOperationHook extends OperationHook

```
    @Override
    public void process(OperationContext context) throws
TechnicalException, FunctionalException

        if (context instanceof DefaultComponentOperationContext)

            DefaultComponentOperationContext componentContext =
(DefaultComponentOperationContext) context;
            for (Component component :
componentContext.getComponents())

                component.setName(component.getName() + "_" +
Time.absoluteTime());
```

13 The `@RestController` annotation defines the hook as a REST web service exposed on `/`

14 The `ModifyOperationHook` class extends the `OperationHook` class, making it easier to implement a hook using the Spring Boot framework

17 The `process` method must be implemented to define the behavior following the execution of an operation within FlowerDocs Core

17 The `OperationHook` reacts only to `DefaultComponentOperationContext` class contexts

24 Each component of the operating context is renamed using the current date

FlowerDocs Academy / Operation Hook / Subscription

In order for the `OperationHook` to be notified each time a document is created, a subscription must be defined:

- Go to the **Administration > Configuration > Operations subscriptions section**
- Click on the **+** button to create a new subscription

In the subscription creation form, fill in the following fields:

- `OperationHandler`: enter the hook's HTTP URL (e.g. *http://localhost:7777/modify*).
- `Execution phase`: select the `Before` phase so that the name change is applied before the document is persisted
- `Action`: to react to a creation, select the `Create` action
- `Object ype`: choose the `Document` type
- `Activated`: tick the checkbox to activate your subscription
- `Authorization`: fill in the BASIC authorisation string using Blitter from the user defined in the *application.properties* file (example: Basic dG90bzoxMjM0)

Finally, confirm the creation of the subscription.

Validation

In order to test that the `OperationHook` implemented is operational:

- create a document by the means of your choice (e.g. via the admin console)
- open the created document

If `OperationHook` has been configured correctly, the creation timestamp has been added to the name of the document created.

> **INFO**
>
> If the name has not been changed, check that `OperationHook` has been started and is accessible from FlowerDocs Core. If this is the case, check the logs for more details.

FlowerDocs Academy / Viewing attachment metadata / Goals

This module enables you to configure the plugin for viewing attachment metadata.

Before getting started

This tutorial is based on the configuration of attachment plugins via the FlowerDocs JS API

In this tutorial we will use the `GEC_Step2_ATraiter` tasks in the GEC module.

FlowerDocs Academy / Viewing attachment metadata / Visualize metadata for all job attachments

To begin with, we'll simply configure the plugin to allow us to view the metadata of each attachment to a `GEC_Step2_ATraiter` class task.

Add the following script:

```
new MetadataVisualizationAttachmentPlugin({
    classId: "GEC_Step2_ATraiter",
}).bind();
```

The user can now view the metadata of each attachment of a `GEC_Step2_ATraiter` task, in an Off Menu, by clicking on the `eye` icon of the attachments.

Restricting the action to a single attachment

We will now restrict the view of this action. We want only the attachment identified as `Mail` to have the visualization action. To do this, we add the `attachmentId` option to our plugin configuration to specify which attachment the action should be positioned on.

RESTRICTION ON THE ATTACHMENT

```
attachmentId: ‘Mail'
```

We therefore obtain the following script:

VISUALIZATION ACTION ON ATTACHMENT INCOMING MAIL FROM A TASK MAIL TO BE PROCESSED

```
new MetadataVisualizationAttachmentPlugin({
        classId: 'GEC_Step2_ATraiter',
        attachmentId: ‘Mail'
}).bind();
```

To go further: customize your action

Now we want to customize the visualization action. We are going to add the options for customizing the title and icon of the action to our script:

OPTIONS FOR CUSTOMIZING THE VISUALIZATION ACTION

```
        title: ‘Visualizing incoming mail data',
        icon:'fa-up-right-from-square'
```

With this script, the visualization action is fully customized:

CUSTOMIZING THE METADATA VISUALIZATION ACTION

```
new MetadataVisualizationAttachmentPlugin({
        classId: 'GEC_Step2_ATraiter',
        attachmentId: ‘Mail'
        title: ‘Visualizing incoming mail data',
        icon:'fa-up-right-from-square'
}).bind();
```

FlowerDocs Academy / Word document generation plugin / Goal

This module explains how to configure the plugin for generating a document from a template. We will then look at how to use it to generate an attachment from a template in FlowerDocs. We will also see how this attachment can be edited directly in Word.
When you have finished this module, you will be able to:

- configure the creation of documents from a template,
- use this system to create context-specific attachments.

Before getting started

This tutorial requires the use of the GEC scope, as well as the FlowerDocs JavaScript API. It is based on the TemplateAttachmentPlugin.

FlowerDocs Academy / Word document generation plugin / Goal

This section explains how to configure the plugin and implement it on a scope integrating the GEC module.
We want to apply it to the `Response` attachment of the `GEC_Step2_ToBeProcessed` tasks. Here we want a new document to be created directly from a Word template and associated with the `Response` attachment.

Prerequisites

You need a FlowerDocs environment with a scope that includes the GEC module.
Check that `TemplateAttachmentPlugin` is present on the scope. It can be found in the administration, in the **Display > Scripts**tab.

Configuration

The plugin must be configured for a specific element before it can be used. To do this, follow these steps:

- Create a new JavaScript script in the **Display > Scripts** tab of FlowerDocs administration. You can name this script TemplateAttachmentConfiguration.
- The contents of this script will be as follows:

```
new TemplateAttachmentPlugin({
        attachmentId: ‘Response',
        classId: 'GEC_Step2_ATraiter',
        instantiator:
                function(fileId) {
                        var doc = new Document();
                        doc.addFile(fileId);
                        doc.setClassId('CourrierSortant');

TagOracle.predict(this.formAPI.getComponent(), doc);
                        return doc;
                },
}).bind();
```

- Save the script, then purge the application cache. This takes into account any changes made to the scope configuration.

SCRIPT EXPLANATION

1 We use the `TemplateAttachmentPlugin` as expected.

2-3 The plugin is only applied to `Response` attachments for `GEC_Step2_ToBeProcessed` tasks.

4 The `instantiator` function generates a document from a file whose id is supplied.

6 The document is instantiated.

7-8 The document receives its file and the desired document class.

9 The document is returned when called by the API.

You can add options to this script:

SCRIPT OPTIONS

```
        canAttach:
                function(card, definition, formAPI) {
                        return
formAPI.getComponent().getAssignee() ==
JSAPI.get().getUserAPI().getId();
                },
```

Restricts plugin use to the user to whom the task is assigned.

```
        nameBuilder:
                function() {
                        return 'Réponse-' + new
Date().toLocaleDateString() + '.docx';
                },
```

This function gives the document the name of the added file. It must be called in the `instantiator` function as follows (after document instantiation):

```
                        var name = this.nameBuilder();
                        doc.setName(name);
```

FlowerDocs Academy / Word document generation plugin / Goal

In this section, we will look at how to use the plugin configured in the previous step.

Prerequisites

To use `TemplateAttachmentPlugin`, you need to have completed the steps in the previous section.

Choice of document template

First of all, you need to have a `GEC_Step2_ToBeProcessed` task assigned to you. If it does not already have a document attached to the `Response` attachment, then you will see the following icon appear in the area provided for the attachment:

You will then have a choice of templates. Select the template that meets your needs, then confirm your choice.

Modifying a document in Microsoft Word

When you validate the desired document template, a new folder is created in FlowerDocs. The content of this is a file in Word format generated according to the selected template.

This file is then saved locally and can be edited directly in Word and then updated on FlowerDocs. If the Microsoft Word plugin is installed on your computer, the file will be opened in Word and you can update it on FlowerDocs directly from Word.

The only thing left to do is to save the task or apply a reply to it for the attachment to be taken into account.

Guides / Graphical user interface / Introduction

The graphical user interface is configured using XML files. These XML files are stored in FlowerDocs as `GUIConfiguration` class documents.

These files must be in Spring Beans format. The various XML fragments described in this documentation must be added to the `beans` tag:

```
        <?xml version="1.0" encoding="UTF-8"?>
        <beans xmlns="http://www.springframework.org/schema/beans"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans
               http://www.springframework.org/schema/beans/spring-
beans.xsd
               http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

                <!-- Add here your custom configuration -->

        </beans>
```

To make editing these configuration files easier, they can be modified directly from the administration console.

Instance description

The following properties are used to define the deployed instance, and must be added to the `gui.properties` file:

- `gui.client.app.desc`: defines the instance description
- `gui.client.app.logo`: defines the instance logo
- `gui.client.app.env`: defines the environment on which the instance is deployed
- `gui.client.app.style`: defines a custom CSS style sheet to be applied
- `gui.password.change.enabled`: enables/disables the action of changing a user's password

> **INFO**
>
> Find out how to quickly access the different FlowerDocs regions

Guides / Graphical user interface / Accessibility

To improve the accessibility of FlowerDocs and meet [RGAA] requirements (https://design.numerique.gouv.fr/accessibilite-numerique/rgaa/), keyboard navigation in FlowerDocs has been improved.

A menu has been set up to make it easier to navigate FlowerDocs. It allows a user to reduce the number of **Tabs** needed to reach a point of interest on the current page. This menu is called the Accessibility Menu.

Accessibility menu

Open

The accessibility menu is opened using the following keyboard shortcut: **Ctrl + Alt + N**

This shortcut can be accessed from any part of the FlowerDocs application.

Navigation

Once the menu is open, use the tab key and the Enter key to choose the region you want to move to from the list.

> **INFO**
>
> The list of choices in the menu varies according to the current page and the regions it contains.

FlowerDocs versions affected

Version 2.6.1 and later.

Guides / Graphical user interface / Login page

The display of the login form can be configured using properties defined in the `gui.properties` file:

Property name	Default value	Description
`gui.password.change.enabled`	`true`	Display the link for changing the user's password
`scope.edit`	`false`	Display the scope selector on which to connect
`gui.sign.in.form.enabled`	`true`	Display the user/password connection form

Guides / Graphical user interface / Home page / Overview

Widgets on the home page are defined by team using the `home.widget.catalog=<identifiant du catalogue>` property from widget catalogs.

> **INFO**
>
> Hide the home page by adding the `home.hide=true` property at team level and redefine the default page as not being the home page.

Guides / Graphical user interface / Home page / Search

This type of widget displays the concatenation of fields present in the `selectClause` search for each component retrieved by the configured search.

Users can click on one of the displayed results to access the component.

In the example below, we start by defining the request that will populate the widget:

```
<bean id="tenLastBillRequest"
class="com.flower.docs.domain.search.SearchRequest">
        <property name="selectClause">
                <bean
class="com.flower.docs.domain.search.SelectClause">
                        <property name="fields">
                                <list>
                                        <value>name</value>
                                </list>
                        </property>
                </bean>
        </property>
        <property name="max" value="10" />
        <property name="filterClauses">
                <list>
                        <bean
class="com.flower.docs.domain.search.AndClause">
                                <property name="criteria">
                                        <list>
                                                <bean
class="com.flower.docs.domain.search.Criterion">
                                                        <property
name="name" value="classid" />
                                                        <property
name="type">

<value type="com.flower.docs.domain.search.Types">STRING</value>
                                                        </property>
                                                        <property
name="operator">

<value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>
                                                        </property>
                                                        <property
name="values">
```

```
<list>

<value>ProviderBill</value>

</list>
                                                                </property>
                                                        </bean>
                                                </list>
                                        </property>
                                </bean>
                        </list>
                </property>
                <property name="orderClauses">
                        <list>
                                <bean
class="com.flower.docs.domain.search.OrderClause">
                                        <property name="name"
value="creationDate" />
                                        <property name="ascending"
value="false" />
                                </bean>
                        </list>
                </property>
</bean>
```

Next, we build the `tenLastBillWidget` widget to display the search. In this example, we define:

- the type of component required, the category: here `DOCUMENT`, the available categories are `DOCUMENT`, `TASK`, `FOLDER`, `VIRTUAL_FOLDER`.
- internationalized widget title and description
- the previously defined request `tenLastBillRequest`

```
<bean id="tenLastBillWidget"
class="com.flower.docs.gui.client.home.HomeSearchPresenter">
        <property name="description" >
                <list>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                        <property name="language"
value="EN" />
                                        <property name="value"
value="My first search" />
                        </bean>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                        <property name="language"
value="FR" />
                                        <property name="value"
value="Ma première recherche configurée" />
                        </bean>
                </list>
        </property>
        <property name="search">
                <bean class="com.flower.docs.domain.search.Search">
                        <property name="category">
                                <value
type="com.flower.docs.domain.component.Category">DOCUMENT</value>
                        </property>
                        <property name="request"
ref="tenLastBillRequest" />
                        <property name="displayNames">
                                <list>
                                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                <property
name="language" value="EN" />
                                                <property
name="value" value="My last bills" />
                                        </bean>
```

```
                                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                <property
name="language" value="FR" />
                                                <property
name="value" value="Mes 10 dernières factures" />
                                        </bean>
                                </list>
                        </property>
                </bean>
        </property>
</bean>
```

Finally, we associate the widget `tenLastBillWidget` with a catalog named `homeWidgets`

```
<bean id="homeWidgets"
class="com.flower.docs.gui.client.util.SimpleWidgetCatalog">
        <property name="widgets">
                <list>
                        <ref bean="tenLastBillWidget" />
                </list>
        </property>
</bean>
```

Guides / Graphical user interface / Home page / Favorites

This type of widget displays the user's favorites based on a configured search.

Users can click on the results displayed to access the component.

In the example below, we start by defining the request that will populate the widget:

```
<bean id="favoriteRequest"
class="com.flower.docs.domain.search.SearchRequest">
        <property name="selectClause">
                <bean
class="com.flower.docs.domain.search.SelectClause">
                        <property name="fields">
                                <list>
                                        <value>name</value>
                                </list>
                        </property>
                </bean>
        </property>
        <property name="max" value="10" />
</bean>
```

Note that the `id` property used in this bean represents the bean identifier for a parameterized search.

Next, we build the `favoriteDocumentsWidget` widget to display the search. In this example, we define:

- the type of component required, the category: here `DOCUMENT`, the available categories are `DOCUMENT`, `TASK`, `FOLDER`, `VIRTUAL_FOLDER`.
- the widget's internationalized title and description
- the request defined previously `favoriteRequest`

```
<bean id="favoriteDocumentsWidget"
class="com.flower.docs.gui.client.preferences.favorite.HomeFavoritesP
        <property name="search">
                <bean class="com.flower.docs.domain.search.Search">
                        <property name="id">
                            <bean class="com.flower.docs.domain.commo
                                <property name="value" value="documen
/>
                            </bean>
                        </property>
                        <property name="category">
                                <value
type="com.flower.docs.domain.component.Category">DOCUMENT</value>
                        </property>
                        <property name="request" ref="favoriteRequest
                        <property name="displayNames">
                                <list>
                                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                <property name="langu
value="EN"/>
                                                <property name="value
value="My favorites"/>
                                        </bean>
                                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                <property name="langu
value="FR"/>
                                                <property name="value
value="Mes favoris"/>
                                        </bean>
                                </list>
                        </property>
                </bean>
        </property>
</bean>
```

And finally, we associate the `favoriteDocumentWidget` widget with a catalog named `homeWidgets`

```
<bean id="homeWidgets"
class="com.flower.docs.gui.client.util.SimpleWidgetCatalog">
        <property name="widgets">
                <list>
                        <ref bean="favoriteDocumentsWidget" />
                </list>
        </property>
</bean>
```

The widget title is clickable, allowing users to directly access the associated search results by pressing it.

Guides / Graphical user interface / Home page / History

This section details the display of widgets based on components modified, created or viewed by the current user.

Components are determined from information stored in the browser cache. This feature therefore requires the user's browser to have the `Local storage` function activated. In addition, once the browser cache has been purged, the information is deleted.

In this example, in addition to the properties described above, we define the `property` property with the value:

- `OPENED`: the latest opened components
- `CREATED`: the latest created components
- `UPDATED`: the latest updated components

```
<bean id="tenLastConsultedDocumentWidget"
class="com.flower.docs.gui.client.home.HomeUserSearchPresenter">
        <property name="property">
                <value
type="com.flower.docs.gui.client.localstorage.LocalPersistentCacheKey
        </property>
        <property name="description">
                <list>
                        <bean class="com.flower.docs.domain.i18n.I18N
                                <property name="language" value="EN"
                                <property name="value" value="10 last
documents" />
                        </bean>
                        <bean class="com.flower.docs.domain.i18n.I18N
                                <property name="language" value="FR"
                                <property name="value" value="10 last
documents" />
                        </bean>
                </list>
        </property>
        <property name="search">
                <bean class="com.flower.docs.domain.search.Search">
                        <property name="category">
                                <value
type="com.flower.docs.domain.component.Category">DOCUMENT</value>
                        </property>
                        <property name="request">
                                <bean
class="com.flower.docs.domain.search.SearchRequest">
                                        <property name="selectClause"
                                                <bean
class="com.flower.docs.domain.search.SelectClause">
                                                        <property nam
                                                                <list

<value>name</value>
                                                        </lis
```

```
                                                                </property>
                                                        </bean>
                                                </property>
                                                <property name="max" value="1
                                        </bean>
                                </property>
                                <property name="displayNames">
                                        <list>
                                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                        <property name="langu
/>
                                                        <property name="value
last opened documents" />
                                                </bean>
                                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                        <property name="langu
/>
                                                        <property name="value
derniers documents consultés" />
                                                </bean>
                                        </list>
                                </property>
                        </bean>
                </property>
</bean>
```

Guides / Graphical user interface / Home page / Miniatures

There is another type of search catalogue that is not based on a list of widgets like the `WidgetCatalog` but based on a search, the `SearchWidgetCatalog`. This provides a map widget for each search.

The properties to be defined are as follows:

- `category`: the component category to be searched
- `titleTemplate`: the search template used for the title
- `headingTemplate`: the search template used for the header
- `contentTemplate`: the search template used for content
- `search`: the search

```
<bean id="incidentCatalog"
class="com.flower.docs.gui.client.home.SearchWidgetCatalog">
        <property name="category">
                <value
type="com.flower.docs.domain.component.Category">TASK</value>
        </property>
        <property name="titleTemplate">
                <list>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="EN" />
                                <property name="value"
value="${IncidentType}" />
                        </bean>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="FR" />
                                <property name="value"
value="${IncidentType}" />
                        </bean>
                </list>
        </property>
        <property name="headingTemplate">
                <list>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="EN" />
                                <property name="value"
value="Declared on : ${creationDate} by ${owner}" />
                        </bean>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="FR" />
```

```
                                <property name="value"
value="Déclaré le : ${creationDate} par ${owner}" />
                        </bean>
                </list>
        </property>
        <property name="contentTemplate">
                <list>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="EN" />
                                <property name="value"
                                        value="&lt;b/&gt;
${classid} &lt;/b&gt; &lt;br/&gt; Status :
${StatusIncident}&lt;br/&gt; Assigned to: ${assignee}&lt;br/&gt;
${IncidentType} on : ${IncidentLocation} for building '${Building}'
&lt;br/&gt;&lt;br/&gt; &lt;i&gt; Last modification :
${lastUpdateDate} &lt;/i&gt; &lt;br/&gt;" />
                        </bean>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="FR" />
                                <property name="value"
                                        value="&lt;b/&gt;
${classid} &lt;/b&gt; &lt;br/&gt; Statut :
${StatusIncident}&lt;br/&gt;  Assigné à : ${assignee}&lt;br/&gt;
${IncidentType} au niveau de : ${IncidentLocation} pour l'immeuble
: '${Building}' &lt;br/&gt;&lt;br/&gt; &lt;i&gt; Dernière
modification : ${lastUpdateDate} &lt;/i&gt; &lt;br/&gt;" />
                        </bean>
                </list>
        </property>
        <property name="search">
                <bean
class="com.flower.docs.domain.search.SearchRequest">
                        <property name="selectClause">
                                <bean
```

```
class="com.flower.docs.domain.search.SelectClause">
                                        <property name="fields">
                                                <list>

<value>classid</value>

<value>IncidentType</value>

<value>IncidentLocation</value>

<value>Building</value>

<value>owner</value>

<value>name</value>

<value>creationDate</value>

<value>lastUpdateDate</value>

<value>StatusIncident</value>
                                                </list>
                                        </property>
                                </bean>
                        </property>
                        <property name="filterClauses">
                                <list>
                                        <bean
class="com.flower.docs.domain.search.AndClause">
                                                <property
name="criteria">
                                                        <list>

<bean class="com.flower.docs.domain.search.Criterion">

<property name="name" value="classid" />

<property name="type">
```

```
<value type="com.flower.docs.domain.search.Types">STRING</value>

</property>

<property name="operator">

<value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>

</property>

<property name="values">

<list>

<value>NouvelIncident</value>

</list>

</property>

</bean>
                                                        </list>
                                                    </property>
                                                </bean>
                                        </list>
                                </property>
                                <property name="orderClauses">
                                        <list>
                                                <bean
class="com.flower.docs.domain.search.OrderClause">
                                                    <property
name="name">

<value>lastUpdateDate</value>
                                                    </property>
                                                    <property
```

```
name="type">
                                                        <value
type="com.flower.docs.domain.search.Types">TIMESTAMP</value>
                                                    </property>
                                                    <property
name="ascending">

<value>false</value>
                                                    </property>
                                            </bean>
                                    </list>
                            </property>
                            <property name="max" value="10"/>
                    </bean>
            </property>
</bean>
```

Guides / Graphical user interface / Home page / Counter

A counter displays the number of search results in a widget on the home page.

```
<bean id="myCounter"
class="com.flower.docs.gui.client.home.SearchCountPresenter">
        <property name="header">
                <list>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                        <property name="language"
value="EN" />
                                        <property name="value"
value="New incidents" />
                        </bean>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                        <property name="language"
value="FR" />
                                        <property name="value"
value="Nouveaux incidents" />
                        </bean>
                </list>
        </property>
        <property name="title">
                <list>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="EN" />
                                <property name="value"
value="Currently : " />
                        </bean>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="FR" />
                                <property name="value"
value="Actuellement : " />
                        </bean>
                </list>
```

```
        </property>
        <property name="icon" value="cardCount-icon far fa-file-
archive flat-red" />
        <property name="search">
                <bean class="com.flower.docs.domain.search.Search">
                        <property name="category">
                                <value
type="com.flower.docs.domain.component.Category">TASK</value>
                        </property>
                        <property name="request">
                                <bean
class="com.flower.docs.domain.search.SearchRequest" />
                        </property>
                </bean>
        </property>
</bean>
```

On the home page, graphs defined by searches can be displayed. The chart type is automatically determined from the number of aggregations:

- one level of aggregation: donut
- two levels of aggregation: histogram

Guides / Graphical user interface / Home page / Donut

Example: Donut of my mails according to their status

```
<bean id="myMailsGraph"
class="com.flower.docs.gui.client.home.graph.HomeGraphPresenter">
        <property name="search">
                <bean class="com.flower.docs.domain.search.Search">
                        <property name="id">
                            <bean
class="com.flower.docs.domain.common.Id">
                                <property name="value"
value="pliSearch" />
                            </bean>
                        </property>
                        <property name="category">
                                <value
type="com.flower.docs.domain.component.Category">TASK</value>
                        </property>
                        <property name="request">
                                <bean
class="com.flower.docs.domain.search.SearchRequest">
                                        <property
name="selectClause">
                                                <bean
class="com.flower.docs.domain.search.SelectClause">
                                                        <property
name="fields">

<list>

<value>name</value>

</list>
                                                        </property>
                                                </bean>
                                        </property>
                                        <property
name="filterClauses">
                                                <list>
                                                        <bean
```

```
class="com.flower.docs.domain.search.AndClause">

<property name="criteria">

<list>

<bean class="com.flower.docs.domain.search.Criterion">

<property name="name" value="assignee" />

<property name="type">

<value type="com.flower.docs.domain.search.Types">STRING</value>

</property>

<property name="operator">

<value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>

</property>

<property name="values">

<list>

<value>${user.id}</value>

</list>

</property>

</bean>

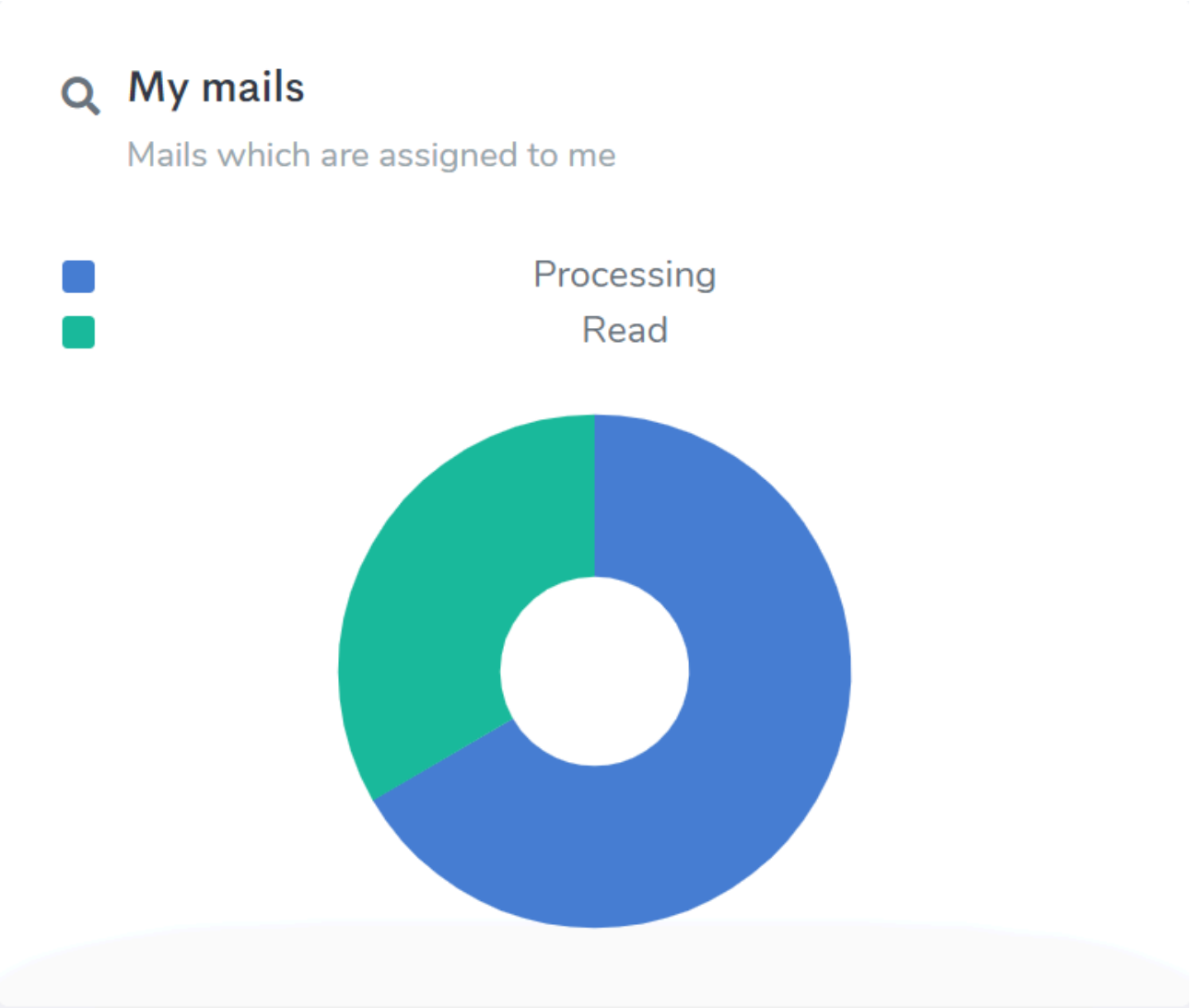
</list>

</property>
```

```
                                                                </bean>
                                                        </list>
                                                </property>
                                                <property
name="aggregation">
                                                        <bean
class="com.flower.docs.domain.search.FieldAggregation">
                                                                <property
name="field" value="Statut" />
                                                        </bean>
                                                </property>
                                        </bean>
                                </property>
                                <property name="displayNames">
                                        <list>
                                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                        <property
name="language" value="EN"/>
                                                        <property
name="value" value="My mails"/>
                                                </bean>
                                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                        <property
name="language" value="FR"/>
                                                        <property
name="value" value="Mes courriers"/>
                                                </bean>
                                        </list>
                                </property>
                        </bean>
                </property>
                <property name="description">
                        <list>
                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                        <property name="language"
```

```
value="EN" />
                                <property name="value" value="Mails
which are assigned to me" />
                        </bean>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="FR" />
                                <property name="value" value="Les
courriers qui me sont assignés" />
                        </bean>
                </list>
        </property>
</bean>
```

The following result on the home page:



INFO

To redirect the user when clicking on a series, override the `handleDonutClick(selector, fieldName, fieldValue, label)` JavaScript method.

Exemple : Redirect to PersonelleTab with filter on selected Statut value

```
        function handleDonutClick(selector, fieldName, fieldValue,
label){
                if (selector.indexOf("#pliSearch") != -1 &&
fieldName==="Statut") {
                var place = new
flower.docs.PlaceBuilder.build("browse(id=PersonnelleTab,leaf=" +
fieldValue + ")");
                JSAPI.get().getNavigationAPI().goTo(place);
```

Histogram

With this type of graph, the first level of aggregation is used to display vertical columns. The second level defines the series.

Example: Histogram of all correspondence by department and status

```
<bean id="allMailsGraph"
class="com.flower.docs.gui.client.home.graph.HomeGraphPresenter">
        <property name="search">
                <bean class="com.flower.docs.domain.search.Search">
                        <property name="id">
                            <bean
class="com.flower.docs.domain.common.Id">
                                <property name="value"
value="allSearch" />
                            </bean>
                        </property>
                        <property name="category">
                                <value
type="com.flower.docs.domain.component.Category">TASK</value>
                        </property>
                        <property name="request">
                                <bean
class="com.flower.docs.domain.search.SearchRequest">
                                        <property
name="selectClause">
                                                <bean
class="com.flower.docs.domain.search.SelectClause">
                                                        <property
name="fields">

<list>

<value>name</value>

</list>
                                                        </property>
                                                </bean>
                                        </property>
                                        <property
name="filterClauses">
                                                <list>
                                                        <bean
```

```
class="com.flower.docs.domain.search.AndClause">

<property name="criteria">

<list>

<bean class="com.flower.docs.domain.search.Criterion">

<property name="name" value="assignee" />

<property name="type">

<value type="com.flower.docs.domain.search.Types">STRING</value>

</property>

<property name="operator">

<value
type="com.flower.docs.domain.search.Operators">DISPLAY</value>

</property>

<property name="values">

<list>

<value>${user.id}</value>

</list>

</property>

</bean>

</list>

</property>
```

```
                                                                </bean>
                                                        </list>
                                                </property>
                                                <property
name="aggregation">
                                                        <bean
class="com.flower.docs.domain.search.FieldAggregation">
                                                                <property
name="field" value="ServiceDestinataire" />
                                                                <property
name="nested">

<list>

<bean class="com.flower.docs.domain.search.FieldAggregation">

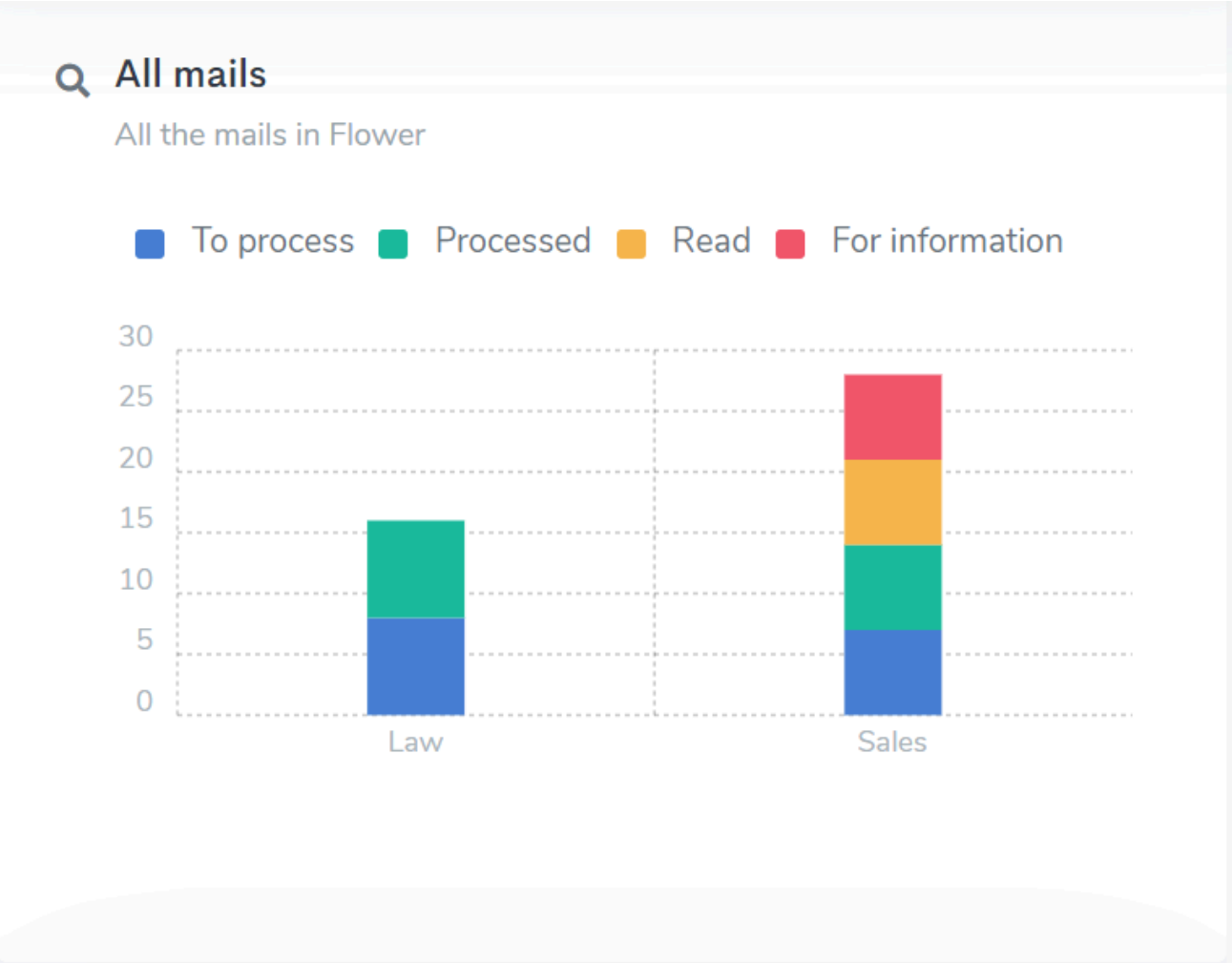
<property name="field" value="Statut" />

</bean>

</list>
                                                                </property>
                                                        </bean>
                                                </property>
                                        </bean>
                                </property>
                                <property name="displayNames">
                                        <list>
                                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                        <property
name="language" value="EN"/>
                                                        <property
name="value" value="All mails"/>
                                                </bean>
                                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                        <property
```

```
name="language" value="FR"/>
                                                    <property
name="value" value="Tous les courriers"/>
                                            </bean>
                                </list>
                        </property>
                </bean>
        </property>
        <property name="description">
                <list>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="EN" />
                                <property name="value" value="All
the mails in FlowerDocs" />
                        </bean>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language"
value="FR" />
                                <property name="value" value="Tous
les courriers dans FlowerDocs" />
                        </bean>
                </list>
        </property>
</bean>
```

The following result on the home page:



INFO

To redirect the user when clicking on a series, override the `handleStackedBarClick(selector, bar, serie)` JavaScript method.

Exemple : Redirect to CollectiveTab tab with filter on selected series and Status value

```
function handleStackedBarClick(selector, barName,barValue,
serieName, serieValue){
        if (selector.indexOf("#allSearch") != -1) {
                var place = new
flower.docs.PlaceBuilder.build("browse(id=CollectiveTab,leaf=" +
barValue + "__"+ serieValue + ")");
                JSAPI.get().getNavigationAPI().goTo(place);
```

Guides / Graphical user interface / Home page / Link to a search form

Search-based widgets can be coupled with a search form. So by clicking on the widget title, the user will be redirected to the search form defined with pre-filled criteria if they are displayed or authorized.

To define the search form to be opened, it is necessary to add the following property to the search bean:

```
<property name="id">
        <bean class="com.flower.docs.domain.common.Id">
                <property name="value" value="AgentSearch" />
        </bean>
</property>
```

Guides / Graphical user interface / Menu / The different types of tabs

URL-based tabs

To enhance integration with other information system applications, FlowerDocs lets you add URL-based tabs to the graphical user interface.

To add a new tab of this type, simply add the following property to the relevant user profile: `tab.url=<label>(<URL>,<icon>)`

> **INFO**
>
> Configure an icon and tab name by setting a property such as: `Viewer(https://arender.io,fa fa-map)`

Folder-based tabs

FlowerDocs lets you define new tabs based on folders (physical or virtual).

The principle consists in displaying a screen divided in two:

- sidebar: presents the various hierarchical aggregations (e.g. documents by validation status)
- content: presents referenced components in sortable tabular form (e.g. documents of a given status)

To add a new tab, simply add the following property to the relevant user profile:

- `tab.folder=<folder identifier>`
- `tab.virtualfolder=<virtual folder identifier>`

The name of the tab displayed in the HMI corresponds to the name of the folder.

The security of these tabs is ensured by the ACL carried by the folder.

> **INFO**
>
> Configure an icon and the name of the internationalized tab by setting a property such as: `MyTab(icon=fa fa-thermometer-2,fr=Dossier,en=Folder)`

Search-based tabs

Search-based tabs allow you to dynamically define tabs corresponding to a physical or virtual folder. After configuring a search for a user team, users see the folders found displayed as a tab.

The search mentioned above must be defined in a `GUIConfiguration` class document and then associated with a team by adding the `tab.component.resolve= <search id>` property.

SEARCH EXAMPLE

```
<bean id="agentFolderSearch"
class="com.flower.docs.domain.search.Search">
  <property name="category">
    <value
type="com.flower.docs.domain.component.Category">FOLDER</value>
  </property>
  <property name="request">
    <bean class="com.flower.docs.domain.search.SearchRequest">
      <property name="selectClause">
        <bean
class="com.flower.docs.domain.search.SelectClause">
          <property name="fields">
            <list>
              <value>name</value>
            </list>
          </property>
        </bean>
      </property>
      <property name="filterClauses">
        <list>
          <bean class="com.flower.docs.domain.search.AndClause">
            <property name="criteria">
              <list>
                <bean
class="com.flower.docs.domain.search.Criterion">
                  <property name="name" value="classid" />
                  <property name="type">
                    <value
type="com.flower.docs.domain.search.Types">STRING</value>
                  </property>
                  <property name="operator">
                    <value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>
                  </property>
                  <property name="values">
                    <list>
                      <value>Folder</value>
```

```
                    </list>
                  </property>
                </bean>
              </list>
            </property>
          </bean>
        </list>
      </property>
      <property name="max" value="1" />
    </bean>
  </property>
</bean>
```

Tab order

The order of all tabs can be redefined for each team. To define the order of a team's tabs, add the `tabs.order` property.

The value of this property must be defined as follows: `${TabType1}(${TabName1});${TabType2}(${TabName2});` where `${TabType#}` = tab type and `${TabName#}` = tab name

The tab type can take on the following values:

Value	Description
NATIVE	FlowerDocs native tab
FOLDER	Folder-based tab
VIRTUALFOLDER	Virtual folder-based tab
URL	URL-based tab
RESOLVED_COMPONENT	Search-based tab(s)

The value of a tab name depends on its type:

Type	Name
FOLDER	Folder identifier
VIRTUALFOLDER	Virtual folder identifier

Type	Name
URL	URL of the tab defined in the tab.url property
RESOLVED_COMPONENT	Search identifier defined in the tab.component.resolve property

For native tabs, the name can take on the following values:

Value	Description
ADMIN	Administration tab
HOME	Home tab
INSERT	Insert tab
SEARCH	Search tab

Please note: All tabs to be displayed must be defined in advance.

Example: The following example displays the tabs in the following order Administration, URL tab "https://flowerdocs.com/documentation", Virtual folder tab "root", Folder tab "Arondor-2020".

The last three tabs must first be defined in the properties `tab.url`, `tab.virtualfolder` and `tab.folder`:

```
NATIVE(ADMIN);URL(https://flowerdocs.com/documentation);VIRTUALFOLDER(root);FOLDER(Arondor-2020);
```

Guides / Graphical user interface / Menu / Default page

The default page is the one displayed at login.

To configure the default page, simply add a property to the corresponding profile, for example: `place.default=home`

The different values supported are:

Values	Description
home	Home page
search(templateName)	Search screen with optional search template name
store	File insertion screen
storeTask(id)	Task creation screen
admin	Admin console
browse(id)	Virtual folder consultation tab
componentResolve(templateName)	Search-based component tab

Two special cases:

- Search: the `search` keyword can take one argument: `search(search_template_id)`
- Virtual folders: it is necessary to specify the virtual folder identifier as a parameter

Guides / Graphical user interface / Menu / Creation shortcuts

The `Create` button in the menu bar facilitates access to the creation of a component of a given class.

This button is configured using a `ComponentReference` object to define:

- `category`: the type of object to be created (document, folder, task, etc.)
- `id`: class identifier

Example : Referencing a `DocumentAgent` class document

```
<bean id="documentAgentRef"
class="com.flower.docs.domain.component.ComponentReference">
        <property name="category">
                <value
type="com.flower.docs.domain.component.Category">DOCUMENT</value>
        </property>
        <property name="id">
                <bean class="com.flower.docs.domain.common.Id">
                        <property name="value"
value="DocumentAgent" />
                </bean>
        </property>
</bean>
```

This reference is then added to the shortcut button. To do this, edit the `componentCreationShortcuts` bean configuration (or define it if it does not

exist) such that:

```
<bean id="componentCreationShortcuts" class="java.util.ArrayList">
   <constructor-arg>
     <list>
       <ref bean="documentAgentRef"/>
     </list>
   </constructor-arg>
</bean>
```

By default, document creation is carried out via the document insertion place and then the indexing place, so that the document content is displayed during indexing.

It is also possible to have a similar behavior to other types of components and create them as popups by setting the `shortcut.document.creation.popup` team property to `true`.
This feature enables users to delegate their work during an absence.

Guides / Graphical user interface / Principle

A delegation is defined by:

- the user on leave
- the delegate
- the vacation period
- a description (optional)

All users access the delegation functionality via the Avatar menu and can create delegations. They also have access to the delegations given to them.

Administrators with the `DELEGATION_MANAGER` team can also create/modify or delete delegations via the administration interface.

How it works

During the defined period, the authorizations of the user on leave are added to those of the delegate:

- access to searches and menus
- access to tasks and documents

All this is necessary to enable the delegate to carry out the absent person ' s work , even if the delegate does not normally have the necessary rights.

In the history, it is always the user who actually performed the action who is recorded.

If an administrator user delegates to a non-administrator user, the latter will access the administration interface during this period.

Guides / Graphical user interface / Indexing screen / Contexts

To configure an indexing screen, you first need to identify the components for which you wish to apply the configuration. This identification is based on context.

These contexts will be used in the following sections to apply a particular configuration in a specific context.

By default, the following generic contexts are provided:

Identifier	Description
`documentInsertContext`	Create document
`documentModifyContext`	Modify document
`documentReadOnlyContext`	Read-only document
`folderInsertContext`	Create folder
`folderModifyContext`	Modify folder
`folderReadOnlyContext`	Read-only folder

Identifier	Description
`virtualFolderInsertContext`	Create virtual folder
`virtualFolderModifyContext`	Modify virtual folder
`virtualFolderReadOnlyContext`	Read-only virtual folder
`taskInsertContext`	Create task
`taskModifyContext`	Modify task
`taskReadOnlyContext`	Read-only task

To identify components more precisely, a context can be defined in terms of a component class. To do this, you need to define it manually using the `IdBasedContext` class, which allows you to define:

- component category
- component class
- phase

The different phases available are as follows:

Value	Description
`INSERT`	Create component
`MODIFY`	Modify component

Value	Description
`READONLY`	Open a component as read-only

The different categories available are as follows:

Value	Description
`DOCUMENT`	Document
`FOLDER`	Folder
`VIRTUAL_FOLDER`	Virtual folder
`TASK`	Task

** Example:** Defining a context used to configure the `IncomingMail` document creation screen.

```
<bean class="com.flower.docs.domain.componentclass.IdBasedContext">
        <property name="id">
                <bean class="com.flower.docs.domain.common.Id">
                        <property name="value"
value="CourrierEntrant" />
                </bean>
        </property>
        <property name="category">
                <value
type="com.flower.docs.domain.component.Category">DOCUMENT</value>
        </property>
        <property name="phase">
                <value
type="com.flower.docs.domain.componentclass.Phase">INSERT</value>
        </property>
</bean>
```

Indexing views are composed of two parts:

- component details (metadata, etc.)
- component content (ARender viewer, results table, etc.)

To add a configuration, first define the `componentActivityConfigurations` bean and save the configuration, contained in the `ComponentActivityConfiguration` object, according to the context:

Example:

```
<bean id="componentActivityConfigurations"
class="com.flower.docs.gui.client.component.activity.ComponentActivit
        <property name="activityConfigurations">
                <map>
                        <entry>
                                <key>
                                        <ref bean="documentModifyCont
                                </key>
                                <bean
class="com.flower.docs.gui.client.component.activity.ComponentActivit
                                        <!-- Custom configuration for
modifications -->
                                </bean>
                        </entry>
                </map>
        </property>
</bean>
```

Guides / Graphical user interface / Indexing screen / General configuration

- `leftPanelWidthRatio`: defines the ratio occupied by the left panel (value between `0.1f` and `1.0f`)
- `goBackAfterSave`: determines whether or not the user should be redirected after saving a component (including when applying a response to a task)

Task creation

- `saveBeforeTaskCreation`: determines whether the component should be saved before a task is created from it
- `saveAfterTaskCreation`: determines whether the component should be saved after a task has been created from it
- `allowTaskCreationIfInvalid`: determines whether it is possible to create a task from an invalid component (if this is the case and save before/after is enabled, then the status is set to `INVALID`)

Confirmations

- `showSaveConfirmationBeforeGo`: defines the display of a save confirmation popup when a modified component is removed from the index.
  If the value is `false`: - the confirmation popup displayed is a confirmation popup to cancel modifications - `showCancelConfirmation` is `false`, no confirmation popup is displayed

Save

- `showSaveConfirmationBeforeGo`: defines the display of a save confirmation popup when a modified component is removed from the index.
- `excludedClassesFromSaveConfirmRule`: excludes component classes from the rule defined by `showSaveConfirmation`
  - If the `showSaveConfirmation` value is `true`, then modifying a component of a class present in this list will not result in the display of a confirmation popup.

Cancel

- `showCancelConfirmation`: defines whether or not to display a confirmation popup if modifications are cancelled when indexing a component
- `excludedClassesFromCancelConfirmRule`: excludes component classes from the rule defined by `showCancelConfirmation`
  - If the `showCancelConfirmation` value is `true`, then cancelling modifications to a component of a class present in this list will not result in the display of a confirmation popup.

Specific configuration

Document

- `minFilesUpload`: the minimum number of files to upload for a document
- `maxFilesUpload`: maximum number of files to upload for a document

Virtual folder

- `useAlternativeView`: allows results to be displayed as a list rather than in the viewer

Customize the virtual folder search

The search associated with a class of virtual folders can be overridden to hide columns or add criteria.

The identifier associated with the search bean of a `DossierVirtuel` virtual folder class is `contentDossiervirtualVirtualFolder`.

Task

- `allowInvalidSaving`: defines whether the component can be saved despite the invalidity of its data

Example:

```
<bean id="componentActivityConfigurations"
class="com.flower.docs.gui.client.component.activity.ComponentActivit
        <property name="activityConfigurations">
                <map>
                        <entry>
                                <key>
                                        <ref bean="documentModifyCont
                                </key>
                                <bean
class="com.flower.docs.gui.client.component.activity.ComponentActivit
                                        <property name="leftPanelWidt
value="0.65f" />
                                        <property name="showSaveConfi
value="true"/>
                                        <property
name="excludedClassesFromSaveConfirmRule">
                                                <list>
                                                        <bean
class="com.flower.docs.domain.common.Id">
                                                                <prop
name="value" value="MyClassId"/>
                                                        </bean>
                                                </list>
                                        </property>
                                </bean>
                        </entry>
                </map>
        </property>
</bean>
```

Guides / Graphical user interface / Indexing screen / System metadata

> **INFO**
>
> System metadata is the metadata natively present on all components (name, class, creation date, etc.). Their configuration on the indexing screens can be modified by overriding the basic configuration.

To create a configuration of this metadata, we need to create a new `ComponentPropertiesConfiguration` bean (containing a list of field definitions) linked to a context.

These associations must be added to the following object:

```
<bean id="componentProperties"
class="com.flower.docs.gui.client.componentclass.ComponentPropertiesC
        <property name="propertiesConfiguration" />
</bean>
```

Field definition

To define a configuration, you need to create a new `ComponentPropertiesConfiguration` bean and assign it field definitions (`FieldDefinition`) to define a particular metadata.

The following field definitions are provided by default:

Identifier	Description
Name	Component name
Class	Component class
Class-ReadOnly	Component class
creationDateData	Date the component was created
lastUpdateDate	Date of last update

They can also be set manually:

```
<bean id="NameDefinition"
class="com.flower.docs.gui.api.model.fields.FieldDefinition">
        <property name="name" value="name" />
        <property name="type">
                <value
type="com.flower.docs.gui.api.model.fields.FieldType">String</value>
        </property>
        <property name="mandatory" value="false" />
        <property name="label" ref="NameLabel" />
</bean>

<bean id="NameLabel"
class="com.flower.docs.gui.api.model.i18n.I18NLabel">
        <property name="languageMap">
                <map>
                        <entry>
                                <key>
                                        <ref bean="EN" />
                                </key>
                                <value>Name</value>
                        </entry>
                        <entry>
                                <key>
                                        <ref bean="FR" />
                                </key>
                                <value>Nom</value>
                        </entry>
                </map>
        </property>
</bean>
```

Configuration

In order to associate field definitions with a context, they must be grouped together in a `ComponentPropertiesConfiguration` object:

```
<bean id="insertDocumentConfiguration"
class="com.flower.docs.gui.client.componentclass.ComponentPropertiesC
    <property name="propertiesConfiguration">
        <list>
            <ref bean="NameDefinition" />
            <ref bean="Class" />
        </list>
    </property>
</bean>
```

All the associations (context / configuration) must then be defined in the following object:

```
<bean id="componentProperties"
class="com.flower.docs.gui.client.componentclass.ComponentPropertiesC
        <property name="propertiesConfiguration">
                <map>
                  <entry>
                        <key>
                          <ref bean="documentInsertContext" />
                        </key>
                        <ref bean="insertDocumentConfiguration" />
                   </entry>
                </map>
        </property>
</bean>
```

INFO

Note : only one `componentProperties` bean can be defined per scope. Otherwise, the last load will override the others.

Guides / Graphical user interface / Indexing screen / Component tracking

Component tracking allows you to view the various tasks linked to a component. This `task tracking` can be configured using the following XML fragment, by setting the desired category via the `category` property of a `taskTrackingConfiguration` bean.

Example: Keeping track of related documents

```
<bean id="taskTrackingConfiguration"
class="com.flower.docs.gui.client.task.tracking.TaskTrackingConfigura
        <property name="category">
                <value
type="com.flower.docs.domain.component.Category">DOCUMENT</value>
        </property>
        <property name="title">
                <list>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language" value="FR"
                                <property name="value" value="Suivi d
copies" />
                        </bean>
                </list>
        </property>
</bean>
```

Note: *This configuration is global for a given scope.*

Guides / Graphical user interface / Folder contents

The `folderSearchCatalog` bean is used to modify the display of data for documents, folders and virtual folders contained in a folder. It is made up of 3 properties:

- `documents`, which lets you configure the display of children documents
- `folders`, which lets you configure the display of children folders
- `virtualFolders`, which configures the display of children virtual folders

These 3 properties work in the same way:

- a `request` request
- a `hiddenColumns` list of hidden columns

By default, document identifiers are retrieved and then hidden in order to retrieve only the data requested during the search

By default in the `flower-docs-gui-client.xml` file, the `folderSearchRequest` catalog is used to display document names, classes, creation date and last modification date.

Example:

```
<bean id="folderSearchCatalog"
class="com.flower.docs.gui.client.search.folder.FolderChildrenTableCa
scope="prototype">
        <property name="documents">
                <bean
class="com.flower.docs.gui.client.search.document.DocumentTablePresen
                        <property name="request" ref="folderSearchReq
                        <property name="hiddenColumns">
                                <list>
                                        <value>id_value</value>
                                        <value>status</value>
                                </list>
                        </property>
                </bean>
        </property>
        <property name="folders">
                <bean
class="com.flower.docs.gui.client.search.folder.FolderTablePresenter"
                        <property name="request" ref="folderSearchReq
                        <property name="hiddenColumns">
                                <list>
                                        <value>id_value</value>
                                        <value>status</value>
                                </list>
                        </property>
                </bean>
        </property>
        <property name="virtualFolders">
                <bean
class="com.flower.docs.gui.client.search.virtualfolder.VirtualFolderT
                        <property name="request" ref="folderSearchReq
                        <property name="hiddenColumns">
                                <list>
                                        <value>id_value</value>
                                        <value>status</value>
                                </list>
                        </property>
```

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

<bean name="folderSearchRequest"
class="com.flower.docs.domain.search.SearchRequest">
        <property name="selectClause">
                <bean class="com.flower.docs.domain.search.SelectClau
                        <property name="fields">
                                <list>
                                        <value>name</value>
                                        <value>classid</value>
                                        <value>creationDate</value>
                                        <value>lastUpdateDate</value>
                                </list>
                        </property>
                </bean>
        </property>
</bean>
```

INFO

Search forms are pre-configured templates made available to end users.

Guides / Graphical user interface / Search / Presentation

A search form allows you to fully configure a search screen. These are associated with scope teams to personalize the display presented to the user.

Example: A basic `AgentSearch` search form with the title Agent file search with description

```
<bean id="RechercheAgent"
class="com.flower.docs.gui.client.search.ComponentSearchPresenter"
scope="prototype">
        <property name="title">
                        <list>
                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                        <property name="language"
value="EN"/>
                                        <property name="value"
value="Agent folder search"/>
                                </bean>
                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                        <property name="language"
value="FR"/>
                                        <property name="value"
value="Rechercher de dossier agent"/>
                                </bean>
                        </list>
                </property>
                <property name="description">
                        <list>
                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                        <property name="language"
value="EN"/>
                                        <property name="value"
value="Do not fill criteria to find all agent folders"/>
                                </bean>
                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                        <property name="language"
value="FR"/>
                                        <property name="value"
value="Ne remplissez aucun critère pour retrouver l'ensemble des
dossiers agents"/>
```

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

Define access

Link a search form to a user team via:

- the CLM and the `scope.xml` file:

  > **TEAM EXAMPLE**

```
<profiles>
        <name>ALL_USERS</name>
        <Description>User profile</Description>
        <principals>ou=users,dc=arondor,dc=com</principals>
        <properties>
                <ns3:name>search.template</ns3:name>
                <ns3:value>RechercheAgent(fr=Agent search,en=Agent
search)</ns3:value>
        </properties>
</profiles>
```

:::

- the administration console: in the `Security > Teams > Properties` section, add a `Search form` property

This property has the following value: `<identifiant>(<language>=<label>,<language2>=<label2>)` for example `template_id(fr=Label de formulaire,en=form label)`

> **INFO**
>
> If no label is entered, the title from the search form will be used.

Guides / Graphical user interface / Search / Quick reference

ComponentSearchPresenter

Property	Type	Description
`title`	`List<I18NLabel>`	Search form title
`description`	`List<I18NLabel>`	Description displayed below the title
`categorySelectorPresenter`	Bean	Component category (document, folder, etc.)
`keywordCriteriaPresenter`	Bean	Keyword search bar configuration
`advancedCriteriaPresenter`	Bean	Advanced criteria panel configuration
`hiddenRequest`	`SearchRequest`	Hidden search request (details)

Property	Type	Description
`hiddenColumns`	`List<String>`	Columns to hide in results (details)
`responsePresenterProvider`	Bean	Result display mode (details)
`actions`	`List<Bean>`	Actions available on the form (details)
`enableActionsIfDirty`	`boolean`	Show actions before search is executed (default: `true`)
`enableIfInvalid`	`boolean`	Allow search when form is invalid (default: `false`)
`emptyResultsMessages`	`List<I18NLabel>`	Custom message when no results are found

KeywordCriteriaPresenter

Property	Type	Description
`enabled`	`boolean`	Enable/disable keyword search (default: `true`)

Property	Type	Description
`fields`	`List<String>`	Tags to search (default: all tags)
`showSearchButton`	`boolean`	Show a dedicated search button (default: `false`)
`searchButtonTitle`	`List<I18NLabel>`	Custom label for the search button

AdvancedCriteriaPresenter

Property	Type	Description
`enabled`	`boolean`	Enable/disable advanced search (default: `true`)
`forceOpen`	`boolean`	Open advanced search and hide the collapse button (default: `false`)
`displayClassSelector`	`boolean`	Show class selector (default: `true`)
`addEmptyCriterion`	`boolean`	Allow adding free criteria with the `+` button (default: `true`)

Property	Type	Description
`showSearchButton`	`boolean`	Show search button (default: `false`)
`searchButtonTitle`	`List<I18NLabel>`	Custom label for the search button (default: `Search`)
`fixedCriterionPresenters`	`List<Bean>`	Pre-configured criteria displayed by default
`searchableCriteria`	`List<String>`	Restrict available criteria to this list (overrides `unsearchableCriteria`)
`unsearchableCriteria`	`List<String>`	Hide specific tags from available criteria
`nonUniqueCriteria`	`List<String>`	Allow a tag to appear as multiple criteria
`activateUniqueCriteria`	`boolean`	Enforce unique criteria (default: `true`)
`classCriterionPresenter`	Bean	Custom class selector

Category selector

Each search targets a component category (documents, folders, etc.). A category selector is available by default. To hide it, use `FakeCategorySelectorPresenter` to set the category directly:

```
<property name="categorySelectorPresenter">
        <bean
class="com.flower.docs.gui.client.search.criteria.item.FakeCategorySe
            <property name="value">
                <value
type="com.flower.docs.domain.component.Category">VIRTUAL_FOLDER</valu
            </property>
        </bean>
</property>
```

Keyword search

Enable keyword search:

```
<property name="keywordCriteriaPresenter">
        <bean
class="com.flower.docs.gui.client.search.criteria.KeywordCriteriaPres
/>
</property>
```

Disable it:

```
<property name="enabled" value="false" />
```

Restrict keyword search to specific tags:

```
<property name="fields">
    <list>
        <value>Matricule</value>
        <value>Contractor</value>
    </list>
</property>
```

Show a dedicated search button:

```
<property name="showSearchButton" value="true" />
```

The button label can be customized with the `searchButtonTitle` property using I18N labels.

Advanced search

Enable advanced search:

```
<property name="advancedCriteriaPresenter">
    <bean
class="com.flower.docs.gui.client.search.criteria.advanced.AdvancedCr
/>
</property>
```

Disable it:

```
<property name="enabled" value="false" />
```

Hide the class selector:

```
<property name="displayClassSelector" value="false" />
```

Open the advanced search and hide the collapse button:

```
<property name="forceOpen" value="true" />
```

Override the default `Search` button label:

```
<property name="searchButtonTitle">
        <list>
                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                        <property name="language" value="EN"/>
                        <property name="value" value="Verify"/>
                </bean>
                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                        <property name="language" value="FR"/>
                        <property name="value" value="Vérifier"/>
                </bean>
        </list>
</property>
```

> **INFO**
>
> When using a search criterion on a `CONDITIONAL` tag, if the tag on which it depends is not filled in, all the values of the `CONDITIONAL` tag are displayed.

Free criteria

By default, all tag classes defined as **searchable** can be added as search criteria. To restrict the scope of your searches, you can:

- Prevent users from adding a free criterion using the `+` button:

```
<property name="addEmptyCriterion" value="false" />
```

- Hide a tag from the list of available criteria:

```
<property name="unsearchableCriteria">
        <list>
                <value>ServiceName</value>
                <value>Assignee</value>
        </list>
</property>
```

- Force which tags can be used as criteria (overrides `unsearchableCriteria`):

```
<property name="searchableCriteria">
        <list>
                <value>Nature</value>
        </list>
</property>
```

Fixed criteria

Fixed criteria are pre-configured fields displayed by default. The user fills them in before running the search.

Each fixed criterion is defined as a `FilterCriterionPresenter` bean with a `model` property containing a `Criterion` object. The criterion defines the tag name, the data type and the default operator.

The configurable properties are:

Property	Type	Description
`model`	`Criterion`	**(required)** The search criterion defining `name`, `type`, and `operator`

Property	Type	Description
`description`	`List<I18NLabel>`	Placeholder text displayed in the input field
`displayOperatorSelector`	`boolean`	Show or hide the operator selector dropdown (default: `true`)
`forceMonovalued`	`boolean`	Force the user to select only one value for a normally multivalued criterion (default: `false`)
`mandatory`	`boolean`	Make the field mandatory before the search can be executed (default: `false`)

Criterion types and operators

Each criterion must declare a `type` and an `operator`. The available operators depend on the type:

Type	Description	Available operators
`STRING`	Text field	`CONTAINS`, `EQUALS_TO`, `STARTS_WITH`, `ENDS_WITH`, `DIFFERENT`
`TIMESTAMP`	Date picker	`BETWEEN`

Type	Description	Available operators
`BOOLEAN`	Checkbox	`EQUALS_TO`
`INTEGER`	Numeric field	`EQUALS_TO`, `LESS_THAN`, `GREATER_THAN`, `DIFFERENT`
`CURRENCY`	Currency field	`EQUALS_TO`, `LESS_THAN`, `GREATER_THAN`, `DIFFERENT`

> **INFO**
>
> For choice list tags and conditional tags, the available operators are `EQUALS_TO` and `DIFFERENT`.

Examples

Only `name`, `type`, and `operator` change between criterion types -- see the types table above for valid combinations.

String criterion with a description (placeholder):

```
<bean id="FirstnameCriterionPresenter"
class="com.flower.docs.gui.client.search.criterion.FilterCriterionPre
    <property name="model">
        <bean class="com.flower.docs.domain.search.Criterion">
            <property name="name" value="Firstname" />
            <property name="type">
                <value
type="com.flower.docs.domain.search.Types">STRING</value>
            </property>
            <property name="operator">
                <value
type="com.flower.docs.domain.search.Operators">CONTAINS</value>
            </property>
        </bean>
    </property>
    <property name="description">
        <list>
            <bean class="com.flower.docs.domain.i18n.I18NLabel">
                <property name="language" value="EN"/>
                <property name="value" value="Collaborator firstname"
            </bean>
            <bean class="com.flower.docs.domain.i18n.I18NLabel">
                <property name="language" value="FR"/>
                <property name="value" value="Prénom du collaborateur
            </bean>
        </list>
    </property>
</bean>
```

Mandatory criterion with hidden operator selector:

```
<bean id="ValidationStatusCriterionPresenter"
class="com.flower.docs.gui.client.search.criterion.FilterCriterionPre
    <property name="description">
        <list>
            <bean class="com.flower.docs.domain.i18n.I18NLabel">
                <property name="language" value="EN"/>
                <property name="value" value="Enter a value"/>
            </bean>
            <bean class="com.flower.docs.domain.i18n.I18NLabel">
                <property name="language" value="FR"/>
                <property name="value" value="Saisissez une valeur"/>
            </bean>
        </list>
    </property>
    <property name="displayOperatorSelector" value="false" />
    <property name="forceMonovalued" value="true" />
    <property name="mandatory" value="true" />
    <property name="model">
        <bean class="com.flower.docs.domain.search.Criterion">
            <property name="name" value="ValidationStatus" />
            <property name="type">
                <value
type="com.flower.docs.domain.search.Types">STRING</value>
            </property>
            <property name="operator">
                <value
type="com.flower.docs.domain.search.Operators">STARTS_WITH</value>
            </property>
        </bean>
    </property>
</bean>
```

Assigning criteria to the advanced search

To assign fixed criteria to an advanced search:

```
<property name="fixedCriterionPresenters">
    <list>
        <ref bean="FirstnameCriterionPresenter" />
        <ref bean="creationDateCriterion" />
        <ref bean="ConfidentialCriterion" />
    </list>
</property>
```

Multiple criteria:

By default, each tag in the list of criteria can only be used once. To allow a tag to appear as multiple criteria, use the `nonUniqueCriteria` property:

```
<property name="nonUniqueCriteria">
        <list>
                <value>name</value>
                <value>TIAmount</value>
                <value>creationDate</value>
        </list>
</property>
```

Disable unique criteria enforcement entirely:

```
<property name="activateUniqueCriteria" value="false" />
```

Class selector

Customize the class selector with a `ComponentClassCriterionSelectorPresenter` bean. It supports the same properties as filter criteria:

- `displayOperatorSelector`: hide the operator selector
- `forceMonovalued`: force single-value selection
- `mandatory`: make the field required

Example:

```
<bean id="classIdCriterionPresenter"
class="com.flower.docs.gui.client.search.criteria.clazz.ComponentClas
        <property name="displayOperatorSelector" value="false" />
        <property name="forceMonovalued" value="true" />
        <property name="mandatory" value="true" />
        <property name="model">
                <bean class="com.flower.docs.domain.search.Criterion"
                        <property name="name" value="classid" />
                        <property name="type">
                                <value type="com.flower.docs.domain.s
                        </property>
                        <property name="operator">
                                <value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>
                        </property>
                </bean>
        </property>
</bean>
```

The `CreatableTaskClassCriterionSelectorPresenter` class selector only displays jobs without mandatory or technical attachments, and where the user has creation rights.

Assign a class criterion to an advanced search with the `classCriterionPresenter` property:

```
<property name="advancedCriteriaPresenter">
                <bean
class="com.flower.docs.gui.client.search.criteria.advanced.AdvancedCr
                        <property name="enabled" value="true" />
                        <property name="displayClassSelector" value="
                        <property name="showSearchButton" value="true
                        <property name="classCriterionPresenter"
ref="classIdCriterionPresenter" />
                </bean>
</property>
```

Filters

When an aggregation is defined for the hidden request of the search form, results are displayed in a tree structure. When a bucket is selected, the search is run using the criteria corresponding to the bucket.

Technical data

Technical information positioned by FlowerDocs can also be used as a search criterion or filter:

- `name`: component title
- `id`: component identifier
- `classid`: component class identifier
- `owner`: login of the user who created the component
- `creationDate`: component creation date
- `lastUpdateDate`: date the component was last modified
- `workflow`: workflow identifier, applies only to `TASK` components
- `assigned`: login of the user to whom the task is assigned, applies only to `TASK` components
- `content`: file content, only concerns `DOCUMENT` components

> **INFO**
>
> To be able to search this data, you need to define a `.com.flower.docs.gui.client.search.SearchableFieldCatalog` type `dataCriteriaCatalog` bean.

Validation

Control whether the search button is active when the form is invalid with the `enableIfInvalid` property:

- `true`: allow search even if the form is invalid
- `false`: require a valid form to activate the search button

```
<property name="enableIfInvalid" value="true" />
```

Empty results message

Display a custom message when the search returns no results with the `emptyResultsMessages` property:

```
<property name="emptyResultsMessages">
    <list>
        <bean class="com.flower.docs.domain.i18n.I18NLabel">
            <property name="language" value="EN" />
            <property name="value" value="No document was found" />
        </bean>
        <bean class="com.flower.docs.domain.i18n.I18NLabel">
            <property name="language" value="FR" />
            <property name="value" value="Aucun document n'a été
trouvé" />
        </bean>
    </list>
</property>
```

Advanced configuration of virtual folders

Filters can be configured within a virtual folder and the search used can be overloaded to add or hide columns in the list of results.

Tab

Follow the naming convention for the search bean:

Scope	Bean name pattern	Example (cla
All searches in a VF	`content{ClassId}VirtualFolder`	`contentCourrierc`
One search in a VF	`content{ClassId}VirtualFolder{SearchId}`	`contentCourrierc`
One search across all VFs	`contentVirtualFolder{SearchId}`	`contentVirtualFo`

> **WARNING**
>
> `{ClassId}` and `{SearchId}` are lowercased with the first letter capitalized. If the ID contains underscores (`_`), remove them and capitalize the following letter. Example: `Courrier_Mail` becomes `CourrierMail`.

Example

```
        <bean id="contentCourriercollectiveVirtualFolderCourriersearc
                class="com.flower.docs.gui.client.search.ComponentSea
                scope="prototype">
                <property name="responsePresenterProvider">
                        <bean

class="com.flower.docs.gui.client.search.response.TableSearchResponse
/>
                </property>
                <property name="categorySelectorPresenter">
                        <bean

class="com.flower.docs.gui.client.search.criteria.item.FakeCategorySe
                                <property name="value">
                                        <value
type="com.flower.docs.domain.component.Category">TASK</value>
                                </property>
                        </bean>
                </property>
                <property name="keywordCriteriaPresenter">
                        <bean

class="com.flower.docs.gui.client.search.criteria.KeywordCriteriaPres
                                <property name="enabled" value="false
                        </bean>
                </property>
                <property name="hiddenColumns">
                        <list>
                                <value>status</value>
                                <value>classid</value>
                                <value>B_DirectionDestinataire</value
                        </list>
                </property>
                <property name="advancedCriteriaPresenter">
                        <bean

class="com.flower.docs.gui.client.search.criteria.advanced.AdvancedCr
```

```
                                <property name="enabled" value="true"
                                <property name="forceOpen" value="tru
                                <property name="inline" value="false"
                                <property name="displayClassSelector"
                                <property name="addEmptyCriterion" va
                                <property name="showSearchButton" val
                                <property name="searchButtonTitle">
                                        <list>
                                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                        <property nam
value="EN"/>
                                                        <property nam
value="Filter"/>
                                                </bean>
                                                <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                                        <property nam
value="FR"/>
                                                        <property nam
value="Filtrer"/>
                                                </bean>
                                        </list>
                                </property>
                                <property name="fixedCriterionPresent
                                        <list>
                                                <ref bean="ObjetCrite
                                                <ref bean="DateCriter
                                                <ref bean="NomAdheren
                                        </list>
                                </property>
                                <property name="searchableCriteria">
                                        <list>
                                                <value>B_DateCourrier
                                                <value>B_NomClient</v
                                                <value>B_ObjetCourrie
                                        </list>
                                </property>
```

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

Indexation

Virtual folder search forms in indexation mode follow the same pattern as tabs, with a `{Phase}` suffix:

Scope	Bean name pattern	Example (cla `Courr`
All searches in a VF	`content{ClassId}VirtualFolder{Phase}`	`contentCourriercolle`
One search in a VF	`content{ClassId}VirtualFolder{Phase}{SearchId}`	`contentCourriercolle`
One search across all indexed VFs	`contentVirtualFolder{Phase}{SearchId}`	`contentVirtualFolde`

Available phases: `Modify` and `ReadOnly`.

> **WARNING**
>
> The same underscore rule applies: remove underscores and capitalize the following letter. Example: `Courrier_Mail` becomes `CourrierMail`, so the bean name is `contentCourrierMailVirtualFolderModify`.

Define a hidden search request on a form to apply criteria, columns, sorting, or aggregations invisible to users.

The `ComponentSearchPresenter` object accepts a `hiddenRequest` property with a `com.flower.docs.domain.search.SearchRequest` class bean. This request allows you to:

- add hidden criteria
- configure the columns to be displayed
- set default sorting
- set the number of results to be displayed
- configure aggregations for tree-view display

Example

```
<bean id="monFormulaire"
class="com.flower.docs.gui.client.search.ComponentSearchPresenter"
                scope="prototype">
                <!-- ... -->
                <property name="hiddenRequest">
                        <bean
class="com.flower.docs.domain.search.SearchRequest">
                                <property name="selectClause">
                                        <bean
class="com.flower.docs.domain.search.SelectClause">
                                                <property
name="fields">
                                                        <list>
                                                                <!-
- Columns to display-->

<value>name</value>

<value>TypeCourrier</value>
                                                        </list>
                                                </property>
                                        </bean>
                                </property>
                                <property name="filterClauses">
                                        <list>
                                                <bean
class="com.flower.docs.domain.search.AndClause">
                                                        <!-- Hidden
criteria -->
                                                        <property
name="criteria">

<list>

<bean class="com.flower.docs.domain.search.Criterion">

<property name="name" value="classid" />
```

```
<property name="type">

<value type="com.flower.docs.domain.search.Types">STRING</value>

</property>

<property name="operator">

<value
type="com.flower.docs.domain.search.Operators">EQUALS_TO</value>

</property>

<property name="values">

<list>

<value>CourrierEntrant</value>

</list>

</property>

</bean>

</list>
                                                        </property>
                                                </bean>
                                        </list>
                                </property>
                                <!-- Search results to display -->
                                <property name="max" value="42" />
                        </bean>
                </property>
                <!-- ... -->
        </bean>
```

Guides / Graphical user interface / Search / Sorting

The default sort order of results is configured using the `orderClauses` property on the `SearchRequest` object. Each `OrderClause` defines a field name and a sort direction.

```
<property name="orderClauses">
    <list>
        <bean class="com.flower.docs.domain.search.OrderClause">
            <property name="name" value="creationDate" />
            <property name="ascending" value="false" />
        </bean>
    </list>
</property>
```

Multiple sort clauses can be combined. They are applied in order of declaration.

Aggregation

An aggregation groups search results by field values, displaying them as a tree structure. When a bucket is selected, the search is filtered to match that bucket's criteria.

Aggregations are configured using the `aggregation` property on the `SearchRequest` object with a `FieldAggregation` bean. Nested aggregations create multi-level grouping.

```
<property name="aggregation">
    <bean class="com.flower.docs.domain.search.FieldAggregation">
        <property name="nested">
            <list>
                <bean
class="com.flower.docs.domain.search.FieldAggregation">
                    <property name="field" value="TypeCourrier" />
                </bean>
                <bean
class="com.flower.docs.domain.search.FieldAggregation">
                    <property name="field" value="CanalEntree" />
                </bean>
            </list>
        </property>
    </bean>
</property>
```

Hidden columns

To retrieve tag values via the `selectClause` without displaying them as visible columns, use the `hiddenColumns` property on the `ComponentSearchPresenter`. See Search results for details.

INFO

- We recommend using the notion of hidden requests to simplify access to components, rather than to secure access to them.
- Add the `ADD_FILTERS_TO_SELECT` criterion with the `true` value in the request contexts to display the criteria filled in as columns.

Guides / Graphical user interface / Search / Hiding columns

Hide columns in the results table with the `hiddenColumns` property:

Example

```
<property name="hiddenColumns">
        <list>
                <value>TypeCourrier</value>
        </list>
</property>
```

Display modes

FlowerDocs supports two display modes for search results:

- **Tabular** -- classic table format
- **Thumbnail** -- ARender generates a thumbnail of the component

Tabular and thumbnails (switcher)

The default display lets the user switch between tabular and thumbnail modes. Tabular is shown first by default.

To default to thumbnails, set `tableByDefault` to `false`:

```
<property name="responsePresenterProvider">
        <bean
class="com.flower.docs.gui.client.search.response.SwitcherSearchRespo
                <property name="tableByDefault" value="false" />
        </bean>
</property>
```

To configure thumbnail card content, add the `cardPresenter` property:

```
<property name="cardPresenter">
        <bean
class="com.flower.docs.gui.client.search.response.CardSearchResponseP
                <property name="titleTemplate">
                        <list>
                                <bean class="com.flower.docs.domain.i
                                        <property name="language" val
                                        <property name="value" value=
                                </bean>
                                <bean class="com.flower.docs.domain.i
                                        <property name="language" val
                                        <property name="value" value=
                                </bean>
                        </list>
                </property>
                <property name="headingTemplate">
                        <list>
                                <bean class="com.flower.docs.domain.i
                                        <property name="language" val
                                        <property name="value" value=
${creationDate}, by ${owner}" />
                                </bean>
                                <bean class="com.flower.docs.domain.i
                                        <property name="language" val
                                        <property name="value" value=
${creationDate}, par ${owner}" />
                                </bean>
                        </list>
                </property>
        </bean>
</property>
```

The `CardSearchResponsePresenterProvider` supports three template properties. Each uses `${fieldName}` placeholders that are replaced with actual values from the search results:

Property	Description
`titleTemplate`	Main title of the card
`headingTemplate`	Subtitle displayed below the title
`contentTemplate`	Body content of the card

Tabular only

Force tabular-only display:

```
<property name="responsePresenterProvider">
        <bean
class="com.flower.docs.gui.client.search.response.TableSearchResponse
/>
</property>
```

Thumbnail only

Use `CardSearchResponsePresenterProvider` directly as the `responsePresenterProvider`. It supports the same `titleTemplate`, `headingTemplate`, and `contentTemplate` properties as described above.

```
<property name="responsePresenterProvider">
        <bean
class="com.flower.docs.gui.client.search.response.CardSearchResponseP
                <!-- Configure titleTemplate, headingTemplate, conten
needed -->
        </bean>
</property>
```

Export

Search results can be exported as a CSV file. The export is limited to the first 200 results for performance.
When you right-click on one or more search results, a contextual menu appears. It features both:

- Native actions (open, view, etc.)
- Tasks that can be created from selected results

These actions are also added to the header of the search results table.

To disable a native action, you can add the `menu.contextual.${category}.hidden.actions` property where `${category}` is the component category concerned. If you want to disable the `Download` action in the default document contextual menu, add the `menu.contextual.document.hidden.actions=DOWNLOAD_CONTENT` property.

Guides / Graphical user interface / Search / Determining the Spring bean identifier

To further customize the contextual menu, you can define the actions it contains.

To do this, you first need to define which contextual menu to configure:

- for which component category?
- for what context? A search? A folder? A virtual folder?

Here are some examples of keys to use depending on the context:

- `contextualMenuDocument`: to manually set the default contextual menu
- `contextualMenuDefaultsearchDocument`: to define the contextual menu of a specific `DefaultSearch` search form
- `contextualMenuDossierAgentDocument`: to define the context menu for documents contained in an `agentfolder` virtual folder

Determining the Spring bean identifier

Depending on the component category, use the Java class indicated:

- document: `com.flower.docs.gui.client.search.document.DocumentContextualMenuPresenter`
- task: `com.flower.docs.gui.client.search.task.TaskContextualMenuPresenter`
- folder: `com.flower.docs.gui.client.search.folder.FolderContextualMenuPresenter`
- virtual folder: `com.flower.docs.gui.client.search.virtualfolder.VirtualFolderContextualMenuPresenter`

Example: Defining the context menu for the `DefaultSearch` document search form

```
<bean id="contextualMenuDefaultsearchDocument"
class="com.flower.docs.gui.client.search.document.DocumentContextualM
scope="prototype">
</bean>
```

Defining customized actions

To define actions, it is necessary to:

- define them in the Spring context. To do this, refer to this part of the documentation.
- references in the appropriate context menu.

Example :__ Adding an `exportToCRM` JavaScript action to the context menu of the `DefaultSearch` document search form

```
<bean id="contextualMenuDefaultsearchDocument"
class="com.flower.docs.gui.client.search.document.DocumentContextualM
scope="prototype">
        <property name="actions">
                <list>
                        <ref bean="exportToCRM" />
                </list>
        </property>
</bean>
```

Define actions available on a search form with the `actions` property on the `ComponentSearchPresenter` object:

EXAMPLE

```
<bean id="EnvelopeSearch"
class="com.flower.docs.gui.client.search.ComponentSearchPresenter
scope="prototype">
        <property name="actions">
                <list>
                        <ref bean="myAction1" />
                        <ref bean="myAction2" />
                </list>
        </property>
</bean>
```

By default, these actions are only visible once the search has been executed. To change this behaviour, you can add the following property:

```
<property name="enableActionsIfDirty" value="false" />
```

Guides / Graphical user interface / Search / Types of action

Component creation

An action can be added to allow users to create a component from a search form. This type of action pre-indexes the component to be created based on the search criteria.

For example, if a user searches for a document in the `Invoice` class with a `status` tag whose value is `paid`, then the action will create a document with the same information.

EXAMPLE: FOLDER CREATION

```
<bean id="EnvelopeSearch"
class="com.flower.docs.gui.client.search.ComponentSearchPresenter
scope="prototype">
        <property name="actions">
                <list>
                        <ref bean="createComponentFromSearchActio
                </list>
        </property>
</bean>

<bean id="createComponentFromSearchAction"
class="com.flower.docs.gui.client.search.action.ComponentCreation
        <property name="componentRef">
                <bean
class="com.flower.docs.domain.component.ComponentReference">
                        <property name="category">
                                <value
type="com.flower.docs.domain.component.Category">FOLDER</value>
                        </property>
                        <property name="id">
                                <bean class="com.flower.docs.doma
                                        <property name="value" va
                                </bean>
                        </property>
                </bean>
        </property>
</bean>
```

Screen change

`GoToPlaceActionPresenter` type actions are used to switch screens.

EXAMPLE: GO TO A TASK CREATION SCREEN WITH VERIFICATION

```
<bean
class="com.flower.docs.gui.client.search.action.GoToPlaceActionPr
        <constructor-arg type="java.util.List">
                <list>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language" value="
                                <property name="value" value="Cre
envelope"/>
                        </bean>
                        <bean
class="com.flower.docs.domain.i18n.I18NLabel">
                                <property name="language" value="
                                <property name="value" value="Cré
enveloppe"/>
                        </bean>
                </list>
        </constructor-arg>
        <property name="place">
                <bean
class="com.flower.docs.gui.client.search.CreateWithVerificationPl
                        <property name="id">
                                <bean
class="com.flower.docs.domain.common.Id">
                                        <property name="value"
value="EEnvelopeSearch" />
                                </bean>
                        </property>
                        <property name="category">
                                <value
type="com.flower.docs.domain.component.Category">TASK</value>
                        </property>
                </bean>
        </property>
</bean>
```

Activation

Control when an action is available with the `enablingStrategy` property. Built-in strategies:

- Whatever the search results (default): `AnyResultEnablingStrategy`

```
<bean
class="com.flower.docs.gui.client.search.action.AnyResultEnablingStra
/>
```

- Only if the search leads to no results: `NoResultEnablingStrategy`

```
<bean
class="com.flower.docs.gui.client.search.action.NoResultEnablingStrat
/>
```

- If the user has selected a certain number of components based on an operator: `MultipleComponentsEnablingStrategy` This activation strategy has two properties:
  - `operator`: The operator used to evaluate the number of components selected. The following operators are available: `EQUALS_TO`, `GREATER_THAN`, `LESS_THAN` and `DIFFERENT`.
  - `componentsCount`: The number of components

  Example: Activating the action, if the number of selected components is equal or more than 2.

```
<bean
class="com.flower.docs.gui.client.search.action.MultipleComponents
        <property name="componentsCount" value="2"/>
        <property name="operator">
                <value
type="com.flower.docs.domain.search.Operators">GREATER_THAN</value
        </property>
</bean>
```

Guides / Graphical user interface / Search / Full example

```
<bean id="AgentSearch"
class="com.flower.docs.gui.client.search.ComponentSearchPresenter" sc
        <!-- Title -->
        <property name="title">
                <list>
                        <bean class="com.flower.docs.domain.i18n.I18N
                                <property name="language" value="EN"/
                                <property name="value" value="Agent f
                        </bean>
                        <bean class="com.flower.docs.domain.i18n.I18N
                                <property name="language" value="FR"/
                                <property name="value" value="Recherc
agents"/>
                        </bean>
                </list>
        </property>
        <property name="description">
                <list>
                        <bean class="com.flower.docs.domain.i18n.I18N
                                <property name="language" value="EN"/
                                <property name="value" value="Fill cr
find desired folder"/>
                        </bean>
                        <bean class="com.flower.docs.domain.i18n.I18N
                                <property name="language" value="FR"/
                                <property name="value" value="Remplis
pour retrouver facilement le dossier désiré"/>
                        </bean>
                </list>
        </property>
        <!-- Category: search virtual folders only -->
        <property name="categorySelectorPresenter">
                <bean
class="com.flower.docs.gui.client.search.criteria.item.FakeCategorySe
                        <property name="value">
                                <value
type="com.flower.docs.domain.component.Category">VIRTUAL_FOLDER</valu
```

```
                        </property>
                </bean>
        </property>
        <!-- Keyword search: disabled -->
        <property name="keywordCriteriaPresenter">
                <bean
class="com.flower.docs.gui.client.search.criteria.KeywordCriteriaPres
                        <property name="enabled" value="false" />
                </bean>
        </property>
        <!-- Advanced search: fixed criteria, no class selector -->
        <property name="advancedCriteriaPresenter">
                <bean
class="com.flower.docs.gui.client.search.criteria.advanced.AdvancedCr
                        <property name="enabled" value="true" />
                        <property name="showSearchButton" value="true
                        <property name="displayClassSelector" value="
                        <property name="fixedCriterionPresenters">
                                <list>
                                        <ref bean="MatriculeCriterion
                                        <ref bean="creationDateCriter
                                </list>
                        </property>
                </bean>
        </property>
        <!-- Hidden columns -->
        <property name="hiddenColumns">
                <list>
                        <value>status</value>
                        <value>Matricule</value>
                </list>
        </property>
        <!-- Hidden request: columns, filter, max results -->
        <property name="hiddenRequest">
                <bean class="com.flower.docs.domain.search.SearchRequ
                        <property name="selectClause">
                                <bean
class="com.flower.docs.domain.search.SelectClause">
```

```
                                        <property name="fields">
                                                <list>
                                                        <value>name</
                                                        <value>classi
                                                        <value>creati
                                                        <value>lastUp
                                                </list>
                                        </property>
                                </bean>
                        </property>
                        <property name="filterClauses">
                                <list>
                                        <bean
class="com.flower.docs.domain.search.AndClause">
                                                <property name="crite
                                                        <list>
                                                                <bean
class="com.flower.docs.domain.search.Criterion">

name="name" value="name" />

name="type">

type="com.flower.docs.domain.search.Types">STRING</value>


name="operator">

type="com.flower.docs.domain.search.Operators">STARTS_WITH</value>


name="values">


<value>Agent</value>

</list>
```

```
                                                                    </bea
                                                            </list>
                                                    </property>
                                            </bean>
                                    </list>
                            </property>
                    </bean>
            </property>
            <!-- Empty results message -->
            <property name="emptyResultsMessages">
                    <list>
                            <bean class="com.flower.docs.domain.i18n.I18N
                                    <property name="language" value="EN"
                                    <property name="value" value="No agen
found" />
                            </bean>
                            <bean class="com.flower.docs.domain.i18n.I18N
                                    <property name="language" value="FR"
                                    <property name="value" value="Aucun d
été trouvé" />
                            </bean>
                    </list>
            </property>
</bean>

<!-- Criterion: EDS (string, contains) -->
<bean id="EDSCriterionPresenter"
class="com.flower.docs.gui.client.search.criterion.FilterCriterionPre
    <property name="model">
        <bean class="com.flower.docs.domain.search.Criterion">
            <property name="name" value="EDS" />
            <property name="type">
                <value type="com.flower.docs.domain.search.Types">STR
            </property>
            <property name="operator">
                <value
type="com.flower.docs.domain.search.Operators">CONTAINS</value>
```

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

<!-- Criterion: Matricule (string, starts with) -->
<bean id="MatriculeCriterionPresenter"
class="com.flower.docs.gui.client.search.criterion.FilterCriterionPre
    <property name="model">
        <bean class="com.flower.docs.domain.search.Criterion">
            <property name="name" value="Matricule" />
            <property name="type">
                <value type="com.flower.docs.domain.search.Types">STR
            </property>
            <property name="operator">
                <value
type="com.flower.docs.domain.search.Operators">STARTS_WITH</value>
            </property>
        </bean>
    </property>
</bean>

<!-- Criterion: creationDate (timestamp, between) -->
<bean id="creationDateCriterion"
class="com.flower.docs.gui.client.search.criterion.FilterCriterionPre
        <property name="model">
                <bean class="com.flower.docs.domain.search.Criterion"
                        <property name="name" value="creationDate" />
                        <property name="type">
                                <value
type="com.flower.docs.domain.search.Types">TIMESTAMP</value>
                        </property>
                        <property name="operator">
                                <value
type="com.flower.docs.domain.search.Operators">BETWEEN</value>
                        </property>
                </bean>
```

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

Guides / Graphical user interface / The viewer / ARender at a glance

Users view their documents directly in their browser, using ARender.

ARender is a document viewer that offers:

- support for numerous file formats (see complete list
- high performance
- annotation management (text notes, highlighting, circles, etc.)
- definition of buffer models

Guides / Graphical user interface / The viewer / Profile override

Global

It is possible to override the ARender profile (and its parameters) by defining the property: `gui.client.arender.profile`. The value of this property will be added to the URL calling ARender.

By scope

In the context of a shared FlowerDocs platform, it is possible to have different parameter requirements for each scope. In order to override the ARender profile per scope, a `GUIConfiguration` document must be created, containing the following `arenderConfig` bean:

```
<bean id="arenderConfig"
class="com.flower.docs.gui.client.files.viewer.ARenderConfiguration"
scope="singleton">
                <property name="profil"
value="arender&amp;annotation.stickyNote.default.color=%23000000"
/>
</bean>
```

This example lets you change the default color of sticky notes.

It is not recommended to modify Arender properties by setting parameters in properties files. Properties that are not defined in the documentation are not qualified by FlowerDocs: the correct operation of the application is therefore not guaranteed with these modifications.

Guides / Graphical user interface / The viewer / Principle

Since version 2025.1.0, FlowerDocs included ARender 2023.8.0 (and more). In this version and the following, ARender has included a new feature powered by Mixpanel called Product Analytics. With this feature, the Uxopian team can collect information to better understand how ARender is used in real-world scenarios.

Users can be assured that their sensitive information remains private and secure. The collected data are mainly counts metrics related to the use of some specific functionalities.

> **INFO**
>
> More information about this new feature and the collected information are available in the ARender documentation.

Configuration

This feature is activated by default but can be easily disabled.

To disable Product Analytics, add the property `arender.data.analytics.enabled=false` into the profile.

Documentation about the profile overriding can be found here.

Guides / Graphical user interface / The viewer / Principle

Stamp templates are used to define the stamps a user can create. They can be of two types:

- textual: based on a character string
- image: based on an image *(encoded using the base64 format)*

A stamp template is stored as a catalogue in a document of class `UserPreference` with the following tags:

- `UserPreferenceType` with `STAMP` value
- `User` with the value of the identity to which the stamp template is shared

By user

Users create their own stamp templates via their preferences *(accessible by clicking on their avatar)*. New stamp templates are stored and can be accessed after refreshing the viewer in the annotations menu.

The `User` tag is filled in with the identifier of the user creating the template. Only this user can access the template.

By team

Administrators set up stamp templates that are accessible to a team of users. The *Configuration* > *Stamps* section in the administration console lets them add the XML configuration needed to create stamp templates.

The `User` tag is filled in with the identifier of the team of users who can use the template created.

Textual **Image**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

&lt;annotationTemplateCatalog&gt;
    &lt;annotationTemplates&gt;
        &lt;annotationStyle&gt;
            &lt;arrowWidth&gt;0.0</arrowWidth>
            &lt;backgroundColor&gt;none</backgroundColor>
            &lt;bold&gt;false</bold>
            &lt;borderBackmode&gt;0</borderBackmode>
            &lt;borderColor&gt;green</borderColor>
            &lt;borderStyle&gt;1</borderStyle>
            &lt;borderWidth&gt;1</borderWidth>
            &lt;fontColor&gt;green</fontColor>
            &lt;fontName&gt;arial</fontName>
            &lt;fontSize&gt;20</fontSize>
            &lt;hasBorder&gt;false</hasBorder>
            &lt;heightHead&gt;0.0</heightHead>
            &lt;italic&gt;false</italic>
            &lt;lineBackMode&gt;2</lineBackMode>
            &lt;lineStyle&gt;0</lineStyle>
            &lt;opacity&gt;1.0</opacity>
            &lt;rotation&gt;30</rotation>
            &lt;strikethrough&gt;false</strikethrough>
            &lt;textBackMode&gt;1</textBackMode>
            &lt;underline&gt;false</underline>
        </annotationStyle>
        &lt;annotationType&gt;Stamp</annotationType>
        &lt;contentTemplate&gt;Approved on: ${date.now#dd-MM-yyyy}
</contentTemplate>
        &lt;name&gt;Approved</name>
    </annotationTemplates>
</annotationTemplateCatalog>
```

Guides / Graphical user interface / The viewer / Principle

Obfuscation is used to hide data within the content of a document. This feature leverages the annotation mechanism of the ARender viewer to:

- select the areas or text to be hidden
- restore hidden areas

This feature is particularly useful in contexts where documents contain sensitive data. This data can be hidden from certain users, depending on their document permissions.

Data obfuscation

This section describes how to initiate the obfuscation process.

Security

The obfuscation feature can be used if the user has the `OBFUSCATE` permission on the document, in addition to the `CREATE_ANNOTATION` permission.

From the graphical user interface

From the document viewer, obfuscation can be performed by selecting:

- text
- an area of the document being viewed

The document viewer's advanced search also allows you to obfuscate any results found by using the `search and strikeout` action.

> **INFO**
>
> When downloaded, the document is converted to PDF with masking annotations if the user does not have the `OBFUSCATE` permission.

REST API

String obfuscation can be automated using a REST web service. This web service allows you to define:

- the character string (or regular expression) to be obfuscated

- if the string to be obfuscated is a regular expression
- if the search is case-sensitive
- if the search is sensitive to accents

For each occurrence found, an obfuscation annotation is automatically created.

REST

```
POST <core>/rest/documents/{id}/obfuscations HTTP/1.1
Host: {core}
token: {token}
Content-Type: application/json

  "text": "string",
  "accentSensitive": true,
  "caseSensitive": true,
  "regex": false
```

Access to document content

Access concerned

Obfuscation management is handled cross-functionally by **FlowerDocs Core**. All access points to the content of a document with obfuscations provide the content, taking into account the obfuscations and their security:

- view
- download via **FlowerDocs GUI**
- access via web services

The `OBFUSCATE` permission on an obfuscation annotation allows authorized users to view obfuscated data. If content data is obfuscated for a user, a PDF version is generated on the fly for each access.

The obfuscation process involves no modification of the stored content.

Content generation with obfuscation

In certain situations, it is necessary to have access to the contents of a document with all the obfuscations, even if the user has the right to see the obfuscated data. To achieve this, the REST web service for accessing document content can be used to specify whether obfuscations are to be included.

REST

```
GET: <core>/rest/documents/{id}/files/{file}/content?
includeObfuscations=true HTTP/1.1
Host: {core}
token: {token}
```

If the content obtained includes obfuscations, the HTTP header `X-ALTERED HTTP` with the `true` value is added to the HTTP response.

Guides / Graphical user interface / The viewer / Introduction

FlowerDocs natively embeds ARender. ARender can be configured using a number of parameters, not all of which have been qualified within FlowerDocs. You can find below the list of customisations that have been tested and validated in FlowerDocs.

You can follow this documentation page to learn how to use these customisations here

Annotation styles

Sticky note	
Minimum width	annotation.stickyNote.minimum.width
Minimum height	annotation.stickyNote.minimum.height
Background color	annotation.stickyNote.default.color
Font size	annotation.stickyNote.default.fontColor
Font size	annotation.stickyNote.default.fontSize
Default font	annotation.stickyNote.default.font

Sticky note	
Hide ‘reply’ button when text is empty	annotation.stickyNote.can.hide.reply.button

FreeText	
Opacity	annotation.freetext.opacity
Background color	annotation.freetext.default.color
Border color	annotation.freetext.default.border.color
Border width	annotation.freetext.default.border.width
Font size	annotation.freetext.default.font.size

Rectangle	
Opacity	annotation.rectangle.opacity
Minimum width	annotation.rectangle.minimum.width
Minimum height	annotation.rectangle.minimum.height
Font color	annotation.rectangle.default.color
Border color	annotation.rectangle.default.border.color

Rectangle	
Border size	annotation.rectangle.default.border.width

Highlight, strikeout and underline text	
Opacity	annotation.highligthtext.opacity
Background color	annotation.highlighttext.default.color

Toaster management

Parameter	**Description**	**Default Value**
toaster.log.info.enabled	Display information toasts	`false`

You can find more details in the documentation ARender online

Guides / Graphical user interface / The viewer / Principle

FlowerDocs, through the ARender viewer, offers a document composition function that allows users to **redefine document content** :

- change page order
- delete unwanted pages
- merge two document contents
- create a new document from the pages of an existing document
- rename the file before displaying the indexing page

Guides / Graphical user interface / Global configuration

Within the FlowerDocs GUI, several date formats can be defined:

- In forms using the `gui.date.form` property
- In table columns using the `gui.date.table` property
- For technical information popups `gui.date.technical`
- For other locations, use the `gui.date.display` property

For more information on the different formats supported, please consult this.

Example:

In a form, to obtain dates of the type 01/12/2016, the `gui.date.form=dd/MM/yyyy` property must be defined. This type of date makes it easier to enter dates manually without using the DatePicker object.

Date format configuration by tag class or tag reference

In a `Date` tag class or a `Date` tag reference, it is possible to use a custom date format from supported date formats. Format internationalization is managed by the application.
This section explains how to configure the FlowerDocs GUI labels. The application supports French and English natively. These native labels can be defined using:

- property files
- the JS API (see documentation

Guides / Graphical user interface / Defining labels

The graphical user interface supports French and English natively. Native labels can be overwritten or new languages added. Labels can be defined using property files located in a folder called `labels` in the configuration directory.

The property file containing the labels must be named such as `<locale>.properties` (for example `en.properties`).

Example: Overriding the Home tab label.

fr.properties en.properties

```
home=Tableau de bord
```

Determining the locale

The user's locale is used to determine the language of the labels to be used. This locale is determined by the locale defined in the browser. It can also be overridden by adding the `locale` parameter to the GUI URL (for example: https://www.demo.flowerdocs.cloud/flower-docs-gui?locale=EN).

When no label is defined for the user locale, English (`en`) is used by default.

> **INFO**
>
> Labels can also be determined dynamically using the JS API.

Guides / Graphical user interface / What is a GUI plugin?

A plugin can be used to redirect an HTTP stream received by **FlowerDocs GUI** to another URL. To achieve this, **FlowerDocs GUI** includes a `reverse proxy` module based on Netflix's Zuul open source product. A plugin is exposed under the `/plugins/` path according to the routes defined.

A GUI plugin is configured using two types of information:

- The path of the requests to be intercepted (`Path`)
- The URL to which intercepted requests are redirected (`URL`)

To access a route defined by a plugin, the client must be authenticated to the GUI.

Defining a plugin

A GUI plugin can be configured for a specific scope or for all the scopes of a FlowerDocs instance.

Scope-specific plugin

A plugin can be configured through a `Route` class document with `Path` and `URL` tags.

The configuration of these plugins is described in the *Administration > Configuration > Plugins* section of the administration console.

By defining a plugin with the `/my-route/**` path and the `https://flowerdocs.com/my-plugin` for scope `<scope>`, request is executed on `/plugins/<scope>/my-route/test` and redirected to `https://flowerdocs.com/my-plugin/test` URL.

Global plugin

A global plugin can be accessed from any scope. It must be configured through the `gui.properties` property file. To define a new global plugin, it is necessary to define the corresponding route as:

```
zuul.routes.<plugin-id>.path=<plugin path>
zuul.routes.<plugin-id>.url=<external plugin URL>
```

Example: Defining a plugin named *myplugin*

```
zuul.routes.myplugin.path=/plugins/sample/**
zuul.routes.myplugin.url=http://localhost:3006/sample
```

With this example, requests issued on `<gui>/plugins/sample` are redirected to `http://localhost:3006/sample`.

The timeout on plugins can be configured using the `zuul.host.connect-timeout-millis` and `zuul.host.socket-timeout-millis` properties.

Default plugins

By default, several plugins are added to consume FlowerDocs resources. They are listed in the table below.

Path	Target URL	Description
`/plugins/plume`	`${gui.client.plume.url}`	Redirects to plume
`/plugins/rest`	`${core}/rest`	Redirects to REST services exposed by **FlowerDocs Core**
`/plugins/external`	`${core}/external`	Redirects to plugins exposed by **FlowerDocs Core**

Security

To access a plugin via **FlowerDocs GUI**, the client making the request must be authenticated (via the `SESSION` cookie). This authentication is transmitted to the plugin in the form of an HTTP header `token` whose value is the token of the user making the request.

With the Spring MVC framework, this token can be retrieved by adding the following parameter to the input method: `@RequestHeader("token") String token`.

Guides / Administration / Activation

A FlowerDocs stack made up of several instances requires certain data to be synchronized in order to function at both the **FlowerDocs Core** and **FlowerDocs GUI** levels. To achieve this, FlowerDocs uses the [Redis] key-value database (https://redis.io/) to manage:

- session persistence between different GUI instances
- a distributed cache for maintaining component reservations
- renewal of "static" caches (component classes, ACLs, GUI configurations, etc.) between the various **FlowerDocs Core** and **FlowerDocs GUI** instances
- persistence and distribution of asynchronous `OperationHandler`

This database offers high performance as the data is stored in memory.

> **INFO**
>
> To activate this connector, you need to add the `redis.enabled=true` property.

Access

Depending on the architecture chosen, the configuration for accessing Redis needs to be adapted.

Simple

This communication mode allows the use of a single Redis instance. This can be configured by specifying the `spring.redis.host` and `spring.redis.port` properties.

Note: *This type of configuration is not recommended for a production environment.*

Sentinel

The Redis Sentinel mode enables Redis to be set up in high-availability mode. This mode is activated by entering the name of the Redis master via the `spring.redis.sentinel.master` property, and the set of Redis nodes via `spring.redis.sentinel.nodes` (list of *host:port* separated by commas).

Configuration

The properties listed below are used to configure communication with Redis.

Property	Description
spring.redis.password	Password

Property	Description
spring.redis.timeout	Timeout for Redis exchanges
spring.redis.database	Redis database index
spring.redis.timeout	Maximum connection time

INFO

- It is recommended to follow Redis recommendations to guarantee the security of this component.
- Setting up an encrypted tunnel between FlowerDocs and Redis can be done with the spiped utility as indicated by Redis.

Guides / Administration / Component identifiers

By default, component identifiers are generated in UUID (2^128) format when they are created.

When injecting components via WS, the identifiers supplied as input are, by default, overloaded in favor of UUIDs generated by **FlowerDocs Core**. This behavior can be modified by adding the `id.generate.force` property with the `false` value.

This property can be used to force **FlowerDocs Core** to use identifiers supplied as input (by a business IS, for example). This can be useful when migrating or recovering data.

FlowerDocs can generate two different types of identifier:

- UUID (default)
- Digital (not recommended for production environments)

To change the default configuration, set the `id.generate.type` property to `UUID` or `Numeric`.
This section describes how to configure the download of search results in CSV format.

Guides / Administration / Presentation

Users can download this list in CSV format from the search results. All tags used for the search, including technical tags that are not visible, are exported to the file.

For `USER` tags, the user identifiers are exported.

Configuration

The following configurations are possible and can be set in the `core.properties` file:

Property name	Description	Default value
`search.export.hiddenColumns`	Columns to exclude from the CSV file	`context,acl,statu`
`search.export.separator`	Separator used between columns	`;`
`search.export.multivalued.separator`	Separator used between multi-valued tag values	`,`
`search.export.fileEncoding`	File encoding	`windows-1252`

INFO

The FlowerDocs platform manages the lifecycle of documents from creation to destruction. Several features are implemented to ensure this lifecycle management.

Guides / Administration / Reservations

A reservation allows you to temporarily give write access to a component in the graphical user interface in order to avoid concurrent modifications.

A component is reserved when it is opened in read/write mode by a user of the graphical user interface. If a reserved component is opened by another user, the form is displayed as read-only.

Reservations are automatically deleted when:

- the user exits the screen from which the component was reserved (by taking an action or closing the browser)
- their session expires
- the user logs out

From the graphical user interface, the current user's reservations can be consulted using:

GET /rest/session/reservations

Version management

A version

During the life of a document, various changes may be made to its tags or content. A version of a document is the image of a document at a given moment. It contains both the tags and the content of a document.

It can be created manually or automatically, depending on the configuration.

To track the different versions, each version has:

- A unique identifier
- A label

Version tracking

Document version tracking allows you to view and manage the different versions of a document.

Creating a version

Creating a version involves saving the current state of a document. Creating a version can also be seen as promoting the document as a version.

Restoring a version

Restoring a version allows you to return to a version of a document. The document, with its tags and content, is then in the same state as when the version was created.

Comparing two versions

You can compare two versions of a document using the document viewer. ARender provides a textual comparison, highlighting differences to make it easier to identify text additions, modifications or deletions.

Tracking modes

Document version tracking is configured using a version tracking mode for each document class. Several modes are available to control how document versions are managed:

- `None`: no version is stored
- `Manual`: document versions are created manually: via the user interface or the exposed APIs
- `Auto`: a version is created automatically each time the content of a document is modified

  Content modification only takes effect when the document is saved, and no longer when content is uploaded

This mode is defined by document class to configure how versions are managed.

Version label

By default, the `NumberedVersionLabelStrategy` strategy is used for each automatic version creation. It automatically defines the label of a new version as a minor version, incremented with each new version.

From the document version consultation popup, you can manually create document versions. This action is accessible only if manual or automatic tracking is enabled and the user has permission to update content. Different strategies are therefore available for naming versions.

Storage

Document versions are stored and retained until the document is permanently deleted.

A purge policy can be defined to reduce the volume of stored data.

Guides / Administration / Security / Directory configuration

This section describes the configuration of the company directory.

An administrator account must be set up to perform the following actions:

- user search
- user recovery
- authentication
- etc.

Directory type

To configure access to the company directory, you need to identify the type of access required:

- `simple`: Simple LDAP such as Apache Directory Server or OpenLDAP
- `ad`: Microsoft Active Directory
- `ad-ds`: Microsoft ADLDS

Directory definition by scope

This section covers the configuration of a scope-specific directory. **This is the most flexible configuration mode.**

Directory configuration is accessible from the administration interface in the `Identities > Directory` section, and is stored as a document in the `LDAPConfiguration` class in FlowerDocs.

The information to be configured is as follows:

Identifier	Display name	Description
`LDAPType`	LDAP type	Directory type
`URL`	URL	Directory access url
`User`	User	User to connect to the directory
`Password`	Password	Password for directory user
`BaseDN`	Base DN	DN configured in the directory
`IdAttribute`	Attribute for identifier	Directory attribute used to store the user identifier
`GroupIdAttribute`	Attribute for group identifier	Directory attribute used to store the group identifier
`DisplayNameAttribute`	Display name attribute	Directory attribute used to store the user's

Identifier	Display name	Description
		display name
`SearchAttribute`	Search attribute	Directory attribute used to search for users/groups
`MembersAttribute`	Members	Directory attribute used to store group members
`EnableLowerCaseOfUserName`	Enable evaluation of lower case identifier	Enables you to force the resolution of user identifiers in lower case

Default directory definition

This section covers the configuration of a default directory for a FlowerDocs instance.

This section is not necessary if the directory is configured via the scope-based administration interface described above. The directory type can then be defined by WEB application:

- For **FlowerDocs GUI** using the property: `gui.ldap.type`
- For **FlowerDocs Core** using the property: `ws.ldap.type`

Example: Configuring an embedded server

```
gui.ldap.type=ad
ws.ldap.type=simple
```

To configure access to the LDAP directory, you need to set the ldap property.

Property	Default value	Description
ldap.bind.url		Directory address
ldap.bind.root		Base node in LDAP structure
ldap.base.dn		User search database

An admnistrator account must be set up.

Property	Description
ldap.bind.dn	User's Distinguished Name
ldap.bind.password	User's password

In order to retrieve (or authenticate) users with the configured directory, you must also define:

Property	Description
ldap.attr.id	Attribute used to retrieve a user's identifier

Examples:

- Microsoft Active Directory : `sAMAcountName`
- Microsoft Active Directory LDS: `uid`
- Apache Directory Server: `uid`

Other attributes used for user mapping can be defined:

Property	Description
ldap.attr.display.name	Attribute used to retrieve a user's identifier
ldap.attr.password	Attribute used to retrieve a user's password
ldap.attr.search	Attribute used for research

Configuration examples

ADLDS

```
ldap.bind.url=ldap://ldap.company.com:389
ldap.bind.root=dc=arondor,dc=dev
ldap.bind.dn=CN=fadmin,OU=Demo,OU=FlowerDocs,DC=arondor,DC=dev
ldap.bind.password=okidoki
ldap.base.dn=OU=Demo,OU=FlowerDocs
ldap.attr.id=CN
ldap.attr.display.name=displayName
```

OpenLDAP

OpenLDAP requires that the base DN used

```
ldap.bind.url=ldap://ldap.company.com:389
ldap.bind.root=
ldap.base.dn=dc=arondor,dc=dev
ldap.attr.id=CN
ldap.bind.dn=CN=admin,DC=arondor,DC=dev
ldap.bind.password=okidoki
ldap.attr.display.name=displayName
```

Directory administration

From the administration interface, users can be created with a default password. For this, the password is not mandatory, but is a global parameter for the FlowerDocs instance.

Property	Description
ldap.default.password	Default password if none set at creation
ldap.user.password.mandatory	Boolean defining if the password field is mandatory

Users and groups can only be created at the root of the directory access node.

Example: For Microsoft Active Directory: `<domaine>/<base DN>`

Guides / Administration / Security / User token

A FlowerDocs client, such as **FlowerDocs GUI**, requires authentication to communicate with **FlowerDocs Core**. This authentication is submitted in the form of a *JWT* token.

A token is generated by **FlowerDocs Core** and is signed using an HMAC calculated from the SHA-256 hashing algorithm and a secret key.

Each time a request is received, **FlowerDocs Core** validates the token supplied with the secret key.

By default, this secret key is randomly generated (32 characters long) at **FlowerDocs Core** startup. It is advisable to define it in the `core.properties` file using the `token.key` parameter.

A token is valid for 3600s (60 minutes) from the moment it is generated. To change this validity period, the `token.expiration.time` parameter (*duration in seconds*) can be defined at the **FlowerDocs Core** level.

In some cases, it may be necessary to include the password of the account used in the generated token. This configuration can be activated using the following parameters:

- Inclusion: `token.password.include` (default value `false`)

- Passphrase: `token.password.passphrase`
- Iv: `token.password.iv`

> **INFO**
>
> The token lifetime must be longer than the user session lifetime, which can be set using `gui.session.timeout` (seconds - default value `1800`)

Guides / Administration / Security / Content security policy (CSP) and HTTP Strict Transport Security (HSTS)

To protect against attacks, FlowerDocs sets the `Content security policy` mechanism to the default value of `frame-ancestors 'self'` and the `HTTP Strict Transport Security` mechanism to a default maximum duration of `10 minutes`. However, these values can be changed using the following parameters:

- CSP: `gui.content.security.policy`
- HTS max-age: `gui.hsts.max.age`

> **INFO**
>
> FlowerDocs-specific users can be defined in **FlowerDocs Core**. These users can be used as service accounts, for administrators, etc.

Guides / Administration / Security / System user

The `system` user is the account used by the various FlowerDocs applications. This information can be configured using the `system.admin.username` and `system.admin.password` parameters.

This account is used by:

- **FlowerDocs GUI** to load its configuration
- **FlowerDocs Core** to run OperationHandlers
- CLM to manage scopes
- The FlowerDocs Java client to simplify authentication to FlowerDocs

For each of these applications, we recommend configuring the account used (a different one for each application).

Other users

FlowerDocs Core allows you to define additional accounts and their information:

- `id`: account identifier
- `password`: the account password
- `profiles`: account profiles (roles, groups, teams)

These additional accounts can be set in the `core.properties` and `gui.properties` files:

```
internal.realm.users[0].id=client1
internal.realm.users[0].password=&lt;password&gt;
internal.realm.users[0].profiles=ADMIN,ALL_USERS,LEGAL,BUSINESS,MARKE
```

> **INFO**
>
> The different accounts defined in **FlowerDocs Core** can be used for **all** existing scopes.

Guides / Administration / Security / Secrets

It is not advisable to store passwords in clear text in configuration files. To avoid storing secrets in clear text in the `core.properties` and `gui.properties` files, FlowerDocs provides a secret encryption mechanism.

To indicate to FlowerDocs that a property value is encrypted, it must be defined as `ENC(<encrypted value>)`. An encrypted property is decrypted at application startup using its main secret (`secret`). In this way, a different cipher can be defined for each application.

The application cannot be started if a property, indicated as encrypted, cannot be decrypted.

Property encryption can be achieved in several ways, starting with a master secret:

Curl

```
curl -X POST \
  <core>/rest/encrypt \
  -H 'token: <token>' \
  -d {{toEncrypt}}
```

> **INFO**
>
> With this method, we recommend setting the properties `token.key` and `system.admin.password` as a minimum.
>
> This recommendation also applies to components developed around the FlowerDocs ecosystem: GUI plugins and operation handlers.

The secret for each application can be defined in different ways:

- as an environment variable: the name is `secret` and the value is `,<secret>`, the machine must be rebooted to take effect
- as a property of the JVM by adding: `--secret=<secret>` when running the application
- in the `core.properties` and `gui.properties` files (*not recommended*)

The secret declared in system variables takes precedence over the one defined in the properties file.

Guides / Administration / Security / Concept

OpenID Connect is a protocol for delegating the authentication of an application to a third-party application called an *Identity Provider* (IDP). Based on the OAuth 2 protocol and its authorisation code, OpenId Connect is used by **FlowerDocs GUI** to provide its users with a **S**ingle **S**ign **O**n.

From the authentication page, users authenticate by selecting one of the *Identity Providers* displayed. Once authenticated at the *Identity Provider*, the user is redirected to a redirect URL `redirect_uri` from **FlowerDocs GUI** with an authorization code generated by the *Identity Provider*.

Based on this authorization code, **FlowerDocs GUI** initializes the user's HTTP session after retrieving the `id_token` and `access_token` from the *Identity Provider*. A user token specific to FlowerDocs is then generated.

Prerequisites

1. FlowerDocs Core 's secret key must be shared with **FlowerDocs GUI** in order to use this mechanism. Sharing the secret key is necessary so that **FlowerDocs GUI** can initialize the user session by generating a valid user token (`token.key` property in the *gui.properties* file).

2. Authorization of the removal URL in the *Identity Provider*

Configuration

An *Identity Provider* OpenId Connect can be configured via the FlowerDocs administration console. This configuration is stored inthe `OAuthClientConfiguration` class technical documents. The various parameters to be entered are stored in tags.

OAUTHCLIENTCONFIGURATION DOCUMENT CLASS

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns5:DocumentClass category="DOCUMENT" active="false"
technical="true" xmlns="http://flower.com/docs/domain/common"
xmlns:ns2="http://flower.com/docs/domain/componentclass"
xmlns:ns4="http://flower.com/docs/domain/tagclass"
xmlns:ns3="http://flower.com/docs/domain/i18n"
xmlns:ns6="http://flower.com/docs/domain/component"
xmlns:ns20="http://flower.com/docs/domain/security"
xmlns:ns5="http://flower.com/docs/domain/documentclass"
xmlns:ns8="http://flower.com/docs/domain/search"
xmlns:ns7="http://flower.com/docs/domain/acl"
xmlns:ns13="http://flower.com/docs/domain/scope"
xmlns:ns9="http://flower.com/docs/domain/file"
xmlns:ns12="http://flower.com/docs/domain/reservation"
xmlns:ns11="http://flower.com/docs/domain/task"
xmlns:ns22="http://flower.com/docs/domain/folderclass"
xmlns:ns10="http://flower.com/docs/domain/taskclass"
xmlns:ns21="http://flower.com/docs/domain/virtualfolderclass"
xmlns:ns17="http://flower.com/docs/domain/folder"
xmlns:ns16="http://flower.com/docs/domain/document"
xmlns:ns15="http://flower.com/docs/domain/report"
xmlns:ns14="http://flower.com/docs/domain/workflow"
xmlns:ns19="http://flower.com/docs/domain/fact"
xmlns:ns18="http://flower.com/docs/domain/virtualFolder">
    <id>OAuthClientConfiguration</id>
    <ns2:data>
        <ACL>acl-admin</ACL>
    </ns2:data>
    <ns2:tagReferences tagName="ClientId" mandatory="true"
multivalued="false" technical="false" readonly="false"
order="0" />
    <ns2:tagReferences tagName="ClientSecret" mandatory="true"
multivalued="false" technical="false" readonly="false"
order="0" />
    <ns2:tagReferences tagName="AuthorizationGrantType"
mandatory="false" multivalued="false" technical="false"
readonly="false" order="0" />
```

```
    <ns2:tagReferences tagName="RedirectUriTemplate"
mandatory="true" multivalued="false" technical="false"
readonly="false" order="0" />
    <ns2:tagReferences tagName="Scope" mandatory="false"
multivalued="true" technical="false" readonly="false" order="0"
/>
    <ns2:tagReferences tagName="AuthorizationUri"
mandatory="true" multivalued="false" technical="false"
readonly="false" order="0" />
    <ns2:tagReferences tagName="TokenUri" mandatory="true"
multivalued="false" technical="false" readonly="false"
order="0" />
    <ns2:tagReferences tagName="JwkSetUri" mandatory="false"
multivalued="false" technical="false" readonly="false"
order="0" />
    <ns2:tagReferences tagName="UserInfoUri" mandatory="true"
multivalued="false" technical="false" readonly="false"
order="0" />
    <ns2:tagReferences tagName="UserNameAttributeName"
mandatory="true" multivalued="false" technical="false"
readonly="false" order="0" />
    <ns2:tagReferences tagName="MemberOfAttribute"
mandatory="false" multivalued="false" technical="false"
readonly="false" order="0" />
    <ns2:tagReferences tagName="RegistrationId"
mandatory="true" multivalued="false" technical="false"
readonly="false" order="0" />
    <ns2:tagReferences tagName="ClientName" mandatory="false"
multivalued="false" technical="false" readonly="false"
order="0" />
    <ns2:tagReferences tagName="Icon" mandatory="false"
multivalued="false" technical="false" readonly="false"
order="0" />
        <ns2:tagReferences tagName="RegistrationOrder"
mandatory="false" multivalued="false" technical="false"
readonly="false" order="10">
        <ns4:descriptions language="EN">
            <ns3:value>Loading order</ns3:value>
```

```
        </ns4:descriptions>
        <ns4:descriptions language="FR">
            <ns3:value>Loading order</ns3:value>
        </ns4:descriptions>
        <ns4:pattern></ns4:pattern>
    </ns2:tagReferences>
        <ns2:displayNames language="EN">
                <ns3:value>oAuthClients
configuration</ns3:value>
        </ns2:displayNames>
        <ns2:displayNames language="FR">
</ns5:DocumentClass>
```

Access account

Most *Identity Providers* require authentication to start the authentication process. To configure the account used by FlowerDocs to contact the *Identity Provider*, it is necessary to value the tags:

- `ClientId`: the identifier representing the client application (or *Relying party*): **FlowerDocs GUI**
- `ClientSecret`: the password associated with the identifier

Link to FlowerDocs

- `RedirectUriTemplate`: template used to generate the parameter `redirect_uri` (value must be set to `baseUrl`*/login/oauth2/code/{registrationId}*)
- `Scope`: OAuth 2.0 scopes (at least *openid* and *email*)
- `UserNameAttributeName`: Name of attribute to be used as user identifier

- `RegistrationId`: Unique identifier for authorization server
- `ClientName`: Customer name
- `Icon`: Font Awesome icon to display on login page
- `MemberOfAttribute`: Attribute name for providing groups

Endpoints

The various endpoints required by the OpenId Connect protocol must be configured using the following tags:

- `AuthorizationUri`: User authorization endpoint
- `TokenUri`: Endpoint for token recovery
- `JwkSetUri`: Endpoint used to access the authorization server's public key (JWK) used to validate information received
- `UserInfoUri`: Endpoint displaying user attributes (or *claims*)

For more details, see specifications OpenId Connect

Examples

Google **Microsoft**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns3:Document category="DOCUMENT" name="Google"
xmlns="http://flower.com/docs/domain/common"
        xmlns:ns2="http://flower.com/docs/domain/component"
xmlns:ns3="http://flower.com/docs/domain/document">
        <id>Google</id>
        <ns2:data>
                <ns2:classId>OAuthClientConfiguration</ns2:classId>
        </ns2:data>
        <ns2:Tags>
                <ns2:tags name="ClientId" readOnly="false">
                        <ns2:value>***</ns2:value>
                </ns2:tags>
                <ns2:tags name="ClientSecret" readOnly="false">
                        <ns2:value>***</ns2:value>
                </ns2:tags>
                <ns2:tags name="RedirectUriTemplate"
readOnly="false">

<ns2:value>`baseUrl`/login/oauth2/code/{registrationId}</ns2:value>
                </ns2:tags>
                <ns2:tags name="Scope" readOnly="false">
                        <ns2:value>openid</ns2:value>
                        <ns2:value>profile</ns2:value>
                        <ns2:value>email</ns2:value>
                        <ns2:value>address</ns2:value>
                        <ns2:value>phone</ns2:value>
                </ns2:tags>
                <ns2:tags name="AuthorizationUri" readOnly="false">

<ns2:value>https://accounts.google.com/o/oauth2/v2/auth</ns2:value>
                </ns2:tags>
                <ns2:tags name="TokenUri" readOnly="false">

<ns2:value>https://www.googleapis.com/oauth2/v4/token</ns2:value>
                </ns2:tags>
                <ns2:tags name="JwkSetUri" readOnly="false">
```

```
<ns2:value>https://www.googleapis.com/oauth2/v3/certs</ns2:value>
                </ns2:tags>
                <ns2:tags name="UserInfoUri" readOnly="false">

<ns2:value>https://www.googleapis.com/oauth2/v3/userinfo</ns2:value>
                </ns2:tags>
                <ns2:tags name="UserNameAttributeName"
readOnly="false">
                        <ns2:value>sub</ns2:value>
                </ns2:tags>
                <ns2:tags name="RegistrationId" readOnly="false">
                        <ns2:value>google</ns2:value>
                </ns2:tags>
                <ns2:tags name="ClientName" readOnly="false">
                        <ns2:value>Google</ns2:value>
                </ns2:tags>
                <ns2:tags name="Icon" readOnly="false">
                        <ns2:value>border-danger text-danger mdi
mdi-google</ns2:value>
                </ns2:tags>
        </ns2:Tags>
</ns3:Document>
```

Automatic connection

To authenticate a user automatically using this authentication mechanism, you can add the parameter `sso=auto` to the URL.

With this parameter, the user is automatically authenticated using OpenId Connect when accessing the login page.

Guides / Administration / Security / Permissions

The following permissions are allowed for all components :

Permission	Description
`CREATE`	Authorises creation
`READ`	Authorizes read access
`UPDATE`	Authorises update
`DELETE`	Authorises deletion
`READ_HISTORY`	Access to history
`READ_TASK_HISTORY`	Access to task history

Document-specific permissions:

Permission	Description
`READ_CONTENT`	Read content
`UPDATE_CONTENT`	Update content

Permission	Description
`DOWNLOAD_CONTENT`	Download content (viewer)
`PRINT`	Print (viewer)
`CREATE_ANNOTATION`	Create annotation (viewer)
`READ_ANNOTATION`	Read existing annotations (viewer)
`BUILD_NEW_DOCUMENT`	Activate document clipping (viewer)
`OBFUSCATE`	Create obfuscation annotations and reading hidden data (viewer)

INFO

Note: annotations are stored as documents in FlowerDocs. To be authorized to create annotations, a user must have :

- `READ_ANNOTATION` and `CREATE_ANNOTATION` rights on the business document class
- `CREATE` right on the `Annotation` document class.

To be able to view annotations, a user must have :

- `READ_ANNOTATION` right on the business document class
- `READ` right on the `Annotation` document class.

Special case for obfuscations: To be authorized to obfuscate documents, a user must have the `CREATE_ANNOTATION` and `OBFUSCATE` permissions on the business document class.

> **INFO**
>
> **Page rotation**:
> All users are allowed to rotate pages. Rotation are automatically save to be accessible for other users.

Task-specific permissions:

Permission	Description
`APPROPRIATE`	Appropriate an unassigned task
`APPROPRIATE_ALREADY_ASSIGNED`	Appropriate an already assigned task
`ASSIGN`	Assign a task to a user
`APPLY_ANSWER`	Apply an answer
`UPDATE_CONTENT`	Update attachments
`DELETE_CONTENT`	Delete attachments
`READ_CONTENT`	View attachments

Virtual folder-specific permissions:

Permission	Description
`DOWNLOAD_CONTENT`	Access to zip export including folder's documents

Identities

For FlowerDocs, an identity is either a user, a group or a team. The team concept has been introduced to centralise and pool the management of authorisations common to one or more identities.

ACL Proxy

This feature is in beta. For any integration requirements using ACl's proxies, please contact the FlowerDocs team to help you find the best solution for your needs.

`ACLProxy` type objects are used to add a business aspect to authorisation management.

A proxy is also a `SecurityObject` used to define the security to be applied to a component. It relies on conditions to determine which ACL to apply to a component.

Example:

For an `Invoice` document class, the following proxy could be used:

- if amount < €100: everyone has read-only permission for the document
- if amount > €100: everyone has view/modify permissions for the document

Diagram

Default setting

Defining an unconditional entry in a proxy allows you to define which ACL should be evaluated to create a component from **FlowerDocs GUI**.

Roles

Roles give access to FlowerDocs features through the team concept.

To assign a role to a user:

- create a team whose identifier is the role name
- add users to a team

Role	Description
`ADMIN`	Administers a scope
`DOCUMENT_CREATOR`	Accesses the Insert tab

Guides / Administration / Security / Encryption

Concept

When encryption is enabled, the files handled by FlowerDocs are protected in the event of an attacker accessing the file system. Files are thus rendered unreadable, and you need the secret key to decrypt them.

FlowerDocs uses symmetrical encryption (default `AES 256` per block) to encrypt both document content and temporary files.

To enable content protection by encryption, you need to add the `file.encrypt.enabled=true` property to the `core.properties` configuration file.

When Amazon S3 is used for file storage, FlowerDocs encryption cannot be activated.

S3 natively supports encryption of stored files.

The encryption/decryption performed by S3 is transparent to FlowerDocs.

With S3, the `file.encrypt.enabled=true` property must not be present in the `core.properties` configuration file.

Key management

Key security is one of the main issues when using encryption. The same secret key is used for each FlowerDocs platform. This key must therefore be configured for each instance of **FlowerDocs Core**. Different management strategies are possible:

Derived key

The secret key used for encryption can be derived from a character string (`file.encrypt.passphrase`). Key derivation is performed using the `PBKDF2` algorithm, the `HMAC` function using the `SHA-256` hash function and 65536 iterations. The default key size is 256bits. Its size can be increased or reduced using the `file.encrypt.key-length` property (size in bits).

Keystore

The secret key used for encryption can be stored in a keystore (or *keystore*) in [PKCS12] format (https://tools.ietf.org/html/rfc7292). To use a keystore, you need to configure it using the following properties:

- `key-store.path`: keystore path (example: `file:/data/flowerdocs.pfx`)
- `key-store.password`: keystore password

The keystore can hold several keys. To identify the one to be used, you need to enter the `file.encrypt.key.alias` and `file.encrypt.key.password` properties. These properties respectively define the alias and password of the key entered when registering in the keystore.

Algorithm

The default algorithm is `AES/CBC/PKCS5Padding`. Another algorithm can be used using the `file.encrypt.algorithm` property.

The algorithm used should only be changed after careful analysis, to maintain and guarantee the level of security provided by the type of encryption chosen.

For example, the *DES* algorithm is not recommended.

File system encryption

The CLM command-line utility encrypts unencrypted files stored on file systems using the same information as for the **FlowerDocs Core** instances used:

- the secret key
- the key size
- the encryption algorithm

For more details, see the section below dedicated to its use.

Integrity

To ensure the integrity of files stored through web services exposed by **FlowerDocs Core**, a fingerprinting mechanism is activated by default.

When copying to the file system, the *hash* of each file is calculated using the configured algorithm. The result and the algorithm used are stored with the file information in OpenSearch.

The default algorithm is `SHA-256`. Another algorithm can be used using the `file.hash.algorithm` property and one of the values supported by the JVM (see MessageDigest).

Utility

Directory analysis

By analyzing a directory before encrypting its contents, you can determine the size of the directory to be encrypted and the number of files it contains. The calculation is performed recursively.

The directory is analyzed using the CLM `dir analyze` job, which takes the directory to be analyzed as the `input` parameter.

```
java -jar flower-docs-clm-2025.3.0-bundle.jar dir analyze --input=
<directory to analyze>
```

> **INFO**
>
> The optional parameter `--output=<path to report>` is used to create a report at the end of the analysis.

Directory encryption

As the aim is to encrypt the directory so that it can be read by the various instances of **FlowerDocs Core**, a configuration identical to these must be used.

The directory is analyzed using the CLM `dir analyze` job. It requires the following parameters, regardless of the type of encryption used:

- `input`: the directory to be encrypted

- `output`: the target directory in which the encrypted copy of the directory provided in `input` parameter will be created
- `conf` : patch of the configuration file containing information about the key store or passphrase

The command to be executed is as follows:

```
java -jar flower-docs-clm-2025.3.0-bundle.jar dir encrypt --input=
<directory to encrypt> --output=<target directory with encrypted
data> --conf=<config file path>
```

Derived key

In order to encrypt a directory using a derived key, the parameter to be supplied is `conf` with the value of the path to a configuration file containing the information mentioned in the Derived key section and enable encryption:

```
#Enable encryption
file.encrypt.enabled=true

#Derived key
file.encrypt.passphrase=MyDerivedKey
```

Keystore

In order to encrypt a directory using a keystore strategy, the parameter to be supplied is `conf` with the value of the path to a configuration file containing the information mentioned in the Keystore section and enable encryption:

```
#Enable encryption
file.encrypt.enabled=true

#Keystore management

```properties
key-store.path=file:PATH
key-store.password=****
file.encrypt.key.alias=my-file-encryption-key-alias
file.encrypt.key.password=****
```

```
<br/>
```
```

Guides / Administration / Templates / Principle

An HTML template is a document with an HTML file. This file is used by the [Thymeleaf] template engine (https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html) to generate a new HTML file from a set of variables (or context). It allows you to generate documents based on the same template, using HTML.

> **INFO**
>
> Use these HTML templates to:
>
> - Add private or public pages

Syntax

This section does not cover all the possibilities offered by the [Thymeleaf] engine (https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html)#using-texts) but is intended to provide an initial overview.

Texts

The `th:text` attribute defines the textual value of an HTML tag. It can thus be used with a context variable to display its value:

```
<span th:text="${customerName}">Mrs. Murielle Palutat</span>
```

When the HTML file is generated, the text of the `span` tag is replaced by that of the context variable. It is advisable to define a default text in this type of tag to facilitate rendering previews.

With the `th:text` attribute, the generated text is escaped. If variables contain HTML (e.g. for hypertext links), it is necessary to use the `th:utext` attribute, which does not escape the generated text.

Conditions

The `th:if` attribute conditions the addition of the tag to which it is added. This attribute can be used, for example, to define conditional text blocks:

```
<p th:if="${customerAge > 18}">...</p>
```

Configuration

These templates can be defined directly from the admin console in the `Templates > HTML` section. They are stored as `Template` class technical documents with the `TemplateType` tag whose value is `HTML`.

> **INFO**
>
> The possibilities for providing variables differ according to the use of templates. See the appropriate documentation for details.

Guides / Administration / Templates / Principle

FlowerDocs generates Microsoft Word documents in DOCX format from a template. This template is also a Microsoft Word document in DOCX format that can contain custom properties (*see below*). The properties supplied as input to the generator, and included in the template, are modified when the document is generated to take account of the values supplied.

Properties

Defining a new property

Before a property can be used in a document, it must first be defined at the document level:

- Open advanced properties via the following menu: *Information* > *Properties* > *Advanced properties*
- In the *Custom*tab, enter the following information:
  - Name: property name
  - Type: text
  - Value: enter a default value
- Click on *Add*

Using a property

To use a property, it must be added to the body of the document. To do this:

- Position the cursor at the desired location
- Press `Ctrl` + `F9` simultaneously
- Enter `DOCPROPERTY <Nom de la variable>`
- Press `F9`

Other actions

Shortcuts	Description
`Alt` + `F9`	Enter/exit edit mode for all custom properties
`Shift` + `F9`	Enter/exit edit mode for the selected property
`F9`	Update selected property

Adhoc use

HTTP Curl

```
POST {core}/rest/template/msoffice/{templateId} HTTP/1.1
token:  <token>
Content-Type: application/json
[

        "name": "variable",
        "value": ["myvalue"]

]
```

Guides / Administration / Principle

Pages allow you to expose HTML pages to your users. They are configured by scope and can be used, for example, to collect information such as documents, or simply to make information available.

These pages can be restricted to certain users according to their authorizations, or made public so that they are accessible to all.

This concept requires two types of configuration:

- a template
- a page

Private page

A page is considered `private` if it has an ACL in its definition. In this case, only authenticated users can access it, and the configured ACL is evaluated to determine whether the user is authorized to consult it.

Private pages can be accessed through a browser at the following URL: `/gui/private/<scope>/pages/<page>`.

Public page

A page is considered `public` if its definition does not reference an ACL. In this case, access is public and a user can access it without being authenticated.

If this page requires actions from **FlowerDocs Core**, a token can be injected to provide authentication and enable these actions to be carried out with authentication.

The token supplied initiates authentication with a technical user. The user identifier is the path to the page. It is also assigned the `#PUBLIC` profile.

Security policies can therefore be defined taking into account this public access to authorise or not actions on components.

Public pages can be accessed through a browser at the following URL: `/gui/public/<scope>/pages/<page>`.

Guides / Administration / Operations / Subscription

An operation subscription is a `OperationHandlerRegistration` class technical document. The tags referenced by this class are used to configure the subscription:

Configuration

Tag	Type	Description
`OperationHandler`	*string*	OperationHandler Java class or OperationHook URL
`ExecutionPhase`	*choicelist*	Operation execution phase
`Action`	*choicelist*	Operation action
`ObjectType`	*choicelist*	Object type to respond to
`Enabled`	*boolean*	Determines whether the subscription is active or inactive
`Asynchronous`	*boolean*	Determines whether the OperationHandler should be executed asynchronously or not

Tag	Type	Description
`RegistrationOrder`	*integer*	Scheduling different subscriptions to the same operation
`StopOnException`	*boolean*	Determines whether operation execution should be stopped in the event of an exception (*only if synchronous*)
`Authorization`	*string*	basic authorization string to be supplied to the hook (generated online using blitter

Filters

Filter evaluation

For each operation subscription, execution filters can be defined through a `com.flower.docs.domain.search.SearchRequest` object. These filters can be used to restrict the contexts in which an operation handler is executed. In this way, a subscription can be limited to components that meet the configured criteria.

> **INFO**
>
> To restrict the triggering of a billing process to invoices, the administrator sets the class filter equal to `Bill`.

Filters are evaluated according to the context in which an operation is performed. The data that can be used to filter operations differs according to the type of operation.

An operation performed on a component exposes the component concerned through its context. In this way, its data (tags, owner, class, etc.) can be used to evaluate the defined filters. Whereas when a search is run, filters are evaluated on its criteria.

Storage

Filters are stored through a `com.flower.docs.domain.search.SearchRequest` object. When saved, this object is serialized (XML) and saved as the document file used to define the subscription. The naming of the file containing the filters is imposed. Its name should be `request`.

Special features

Some operations allow the use of specific criteria.

Assignment

The `Id` criterion corresponds to the identifier of the user to whom the task has been assigned. Whereas the `criterion assigned to` corresponds to the user to whom the task was assigned before the operation.

Task response

The `Id` criterion corresponds to the identifier of the user to whom the task has been assigned.

Folder contents

When performing an operation on the contents of a folder (addition or deletion), the `Id` criterion corresponds to the class identifier of the component added to or deleted from the folder.

Search

During a search, subscription filters can be used to react to the execution of specific searches. The filters are then compared with the search criteria.

Filter operators are interpreted as follows:

- `EQUALS_TO` presence of the criterion in the request with the indicated value.
- `CONTAINS`: presence of the criterion in the request with a value containing the string indicated in the filter.

- `STARTS_WITH`: presence of the criterion in the request with a value starting with the character string indicated in the filter.
- `ENDS_WITH`: presence of the criterion in the request with a value ending in the string indicated in the filter.
- `DIFFERENT` no criterion or present with a value different from that indicated in the filter.

For Date criteria, specific operators are used and resolved, such as :

- `GREATER_THAN` presence of the criterion in the request with a value greater than that indicated in the filter.
- `LESS_THAN` presence of the criterion in the request with a value lower than that indicated in the filter.
- `BETWEEN`: The criterion must be present in the SearchRequest with a value between the two specified.

> **INFO**
>
> - The `EQUALS_TO` and `DIFFERENT` operators are case-sensitive.
> - For criteria representing user identifiers, the `${user.id}` value can be defined. This value indicates the current user ID for the session.

Guides / Administration / Operations / Operations managers / Principle

When an operation is executed, the context in which it was performed is provided as input, and its type differs according to the original action. The variable `context` contains an object inherited from `com.flower.docs.operation.api.OperationContext`.

Action context definition

Depending on the action to which the operation is subscribed, the operation context changes:

-- `CREATE`: `com.flower.docs.operation.api.UpdateComponentOperationContext`.

-- `UPDATE`: `com.flower.docs.operation.api.UpdateComponentOperationContext`.

-- `UPDATE_CONTENT`: `com.flower.docs.operation.api.UpdateContentOperationContext`.

-- `ANSWER`: `com.flower.docs.operation.api.TaskOperationContext`.

-- `ASSIGN`: `com.flower.docs.operation.api.TaskOperationContext`.

Guides / Administration / Operations / Operations managers / Principle

This operation manager relies on the [Drools] rules engine (https://www.drools.org/) to execute a decision table. A decision table is a Microsoft Excel file with two types of columns:

- conditions: determines the cases in which to apply a rule
- actions: the actions to be executed for a given rule

For each defined rule (or line), the defined actions are executed if all conditions are met.

The decision table is stored as the content of the document used to configure subscription to the execution of an operation.

Conditions & Actions

The conditions and actions defined must be Java code that can be compiled within **FlowerDocs Core**'s JVM. To facilitate their development, an object accessible through the variable `util` is made available whose exposed methods are listed here.

In order to contextualize decision-making, the `context` and `component` variables are provided when evaluating a decision table. They can therefore be used in conditions or actions. The `context` variable contains a `com.flower.docs.operation.api.OperationContext` object storing the operation execution context.

It is possible to retrieve the component before the action using the `getOld()` method available on the `UpdateComponentOperationContext` or `TaskOperationContext` type `context` variable.

The `component` variable contains the component (see `com.flower.docs.domain.component.Component`) concerned by the operation (if available).

> **INFO**
>
> To manually define this operation handler, the `com.flower.docs.core.tsp.operation.DroolsOperationHandler` identifier can be used as the value of the `OperationHandler` tag.

Drools-type operation managers do not support certain execution contexts:

- on administration objects (component classes, security objects, etc.)
- on document version modification

Guides / Administration / Operations / Operations managers / Principle

This operation manager allows you to react to the execution of an operation by executing a JavaScript script. The JavaScript script is executed using the Graal engine embedded in the FlowerDocs Core JVM.

The script is stored as the content (attached `.js` file) of the `OperationHandlerRegistration` document used to configure the subscription.

Variables

Context-related

The following variables are injected into the script depending on the type of operation:

Variable	Type	Availability	Description
`context`	`OperationContext`	Always	Operation execution context (scope, user, phase, etc.)
`component`	`Component`	Component operations (CREATE, UPDATE, DELETE, READ, ANSWER, ASSIGN)	The component being operated on
`request`	`SearchRequest`	Search operations	The search request being executed
`response`	`SearchResponse`	Search operations (AFTER phase only)	The search response
`logger`	SLF4J Logger	Always	Logger for writing to the FlowerDocs Core log

Variable	Type	Availability	Description
`util`	ContextUtil	Always	Utility object for service access and component persistence

In **BEFORE** phase scripts, modifications to the `component` object are applied to the operation before it executes. In **AFTER** phase scripts, the component reflects its state after the operation completed.

Utilities

Two utility objects are available — they serve different purposes:

- **`util`** is an **instance** of `RuleContextUtil`. Use it for any operation that **calls FlowerDocs Core services**: accessing services (`util.getDocumentService()`, `util.getFolderService()`, ...), persisting components (`util.create()`, `util.update()`), changing classes (`util.changeClass()`), and creating facts (`util.createFact()`).
- **`RuleUtil`** is a **static class reference**. Use it for **in-memory operations** on the `component` object without making service calls: reading and writing tags (`RuleUtil.getTagValue()`, `RuleUtil.setTagValue()`), class identifiers (`RuleUtil.getClassId()`, `RuleUtil.setClassId()`), and statuses (`RuleUtil.getStatus()`, `RuleUtil.setStatus()`).

The full method reference for both objects is documented here.

> **INFO**
>
> Since `util` is an instance of a class that extends `RuleUtil`, calling static methods through `util` (e.g., `util.setTagValue(...)`) also works. However, the recommended convention is `RuleUtil.method()` for static helpers and `util.method()` for service calls.

Pre-imported classes

To simplify script development, the following classes are available by their short name without needing a full package reference:

Builders | **Domain** | **Search** | **Utilities** | **Exceptions**

Class	Description
`ComponentBuilder`	Build Document, Folder, Task, or VirtualFolder instances
`TagBuilder`	Build Tag instances
`CriterionBuilder`	Build search criteria
`FilterClauseBuilder`	Build AND/OR filter clauses
`SearchRequestBuilder`	Build search requests
`SearchBuilder`	Build virtual folder searches

Class	Description
`FactBuilder`	Build audit facts
`ReferenceBuilder`	Build component references
`ExceptionBuilder`	Build functional or technical exceptions

Examples

Creating a folder when creating a document

AFTER / CREATE / DOCUMENT

```
var folder = ComponentBuilder.folder().classId('Folder').build();
folder.setName("Dossier " + component.getName());
util.getFolderService().create(Lists.newArrayList(folder));
util.getFolderService().addChildren(folder.getId(),
Lists.newArrayList(ReferenceBuilder.from(component)), false);
```

Creating an audit fact

AFTER / CREATE / DOCUMENT — Record a business fact for traceability

```
logger.info("[CreateFactOnCreation] Build facts for " +
component.getId());

var builder =
FactBuilder.objectId(component.getId()).type(ObjectType.DOCUMENT);
builder.action("CREATE").description("Processing has been
started.");
util.createFact(builder.build());
```

Workflow state machine (task class transitions)

BEFORE / ANSWER / TASK — Change task class based on the answer given

```
logger.info("[Workflow_OH] START");
var classId = RuleUtil.getClassId(component);
var answerId = component.getAnswer().getId().getValue();
logger.info("[Workflow_OH] classId=" + classId + ", answerId=" +
answerId);

var start = "WKF_Step0_Creation";
var step1 = "WKF_Step1_Treatment";
var end = "WKF_Step2_End";

switch (classId) {
  case start:
    changeClassOnAnswer(answerId, "Initiate", step1);
    break;
  case step1:
    if (!changeClassOnAnswer(answerId, "Validate", end)) {
      changeClassOnAnswer(answerId, "Adjourn", start);
    }
    break;
  default:
}
logger.info("[Workflow_OH] END");

function changeClassOnAnswer(appliedAnswer, expectedAnswer,
classToApply) {
  if (appliedAnswer === expectedAnswer) {
    util.changeClass(component, classToApply);
    logger.info("[Workflow_OH] Class changed to " + classToApply);
    return true;
  }
  return false;
}
```

Search and conditionally create a virtual folder

AFTER / CREATE / DOCUMENT — Create a business folder if it does not already exist

```
let refValue = RuleUtil.getTagValue(component, "NumReference");
let clientValue = RuleUtil.getTagValue(component, "NomClient");

let refCriterion = CriterionBuilder.name("NumReference").operator(Ope
    .type(Types.STRING).value(refValue).build();
let clientCriterion = CriterionBuilder.name("NomClient").operator(Ope
    .type(Types.STRING).value(clientValue).build();
let request = SearchRequestBuilder.init()

.filter(FilterClauseBuilder.and().criterion(refCriterion).criterion(c
    .build();

let response = util.getVirtualFolderService().search(request);
if (response.getFound() == 0) {
  let vf = ComponentBuilder.virtualFolder().classId("BusinessFolder")
      .name(refValue + " - " + clientValue).build();
  vf.getTags().getTags().add(TagBuilder.name("NomClient").value(clien
  vf.getTags().getTags().add(TagBuilder.name("NumReference").value(re
  util.getVirtualFolderService().create([vf]);
}
```

Modifying tags and ACLs

BEFORE / ANSWER / TASK — Update tags and ACL on workflow step transition

```
var classId = RuleUtil.getClassId(component);
var answerId = component.getAnswer().getId().getValue();

if (classId == "Step0_Creation" && answerId === "Initiate") {
  util.changeClass(component, "Step1_Treatment");
  component.setAssignee("");
  component.getData().setACL("acl-treatment");
  util.setTagValue(component, "Status", "In Progress");
}
```

Throwing exceptions

A BEFORE-phase script can prevent the operation from executing by throwing an exception. This only works when `StopOnException` is set to `true` in the registration.

BEFORE — Reject the operation with a functional error

```
// Throw a functional exception (business rule violation)
throw
ExceptionBuilder.createFunctionalException(FunctionalErrorCode.F00008

// Throw a technical exception (system error)
throw
ExceptionBuilder.createTechnicalException(TechnicalErrorCode.T00008);
```

> **INFO**
>
> To manually define this operation handler, the `com.flower.docs.core.tsp.operation.script.ScriptOperationHandler` identifier can be used as the value of the `OperationHandler` tag.

> **WARNING**
>
> The legacy identifier `com.flower.docs.bpm.core.operation.ScriptOperationHandler` is **deprecated** since version 2025.0. Use the identifier above instead. When migrating scripts from the legacy handler, replace `equals()` with `==` for string comparisons and `Date.now()` with `String.valueOf(new Date().getTime())`.

Guides / Administration / Operations / Operations managers / Principle

This operation manager is used to index the content of text documents. This processing is necessary to use full-text searches based on the `content` criterion.

This operation can be activated on the `CREATE` and `ADD_CONTENT` actions.

> **INFO**
>
> To manually define this operation handler, the `com.flower.docs.core.tsp.operation.fulltext.FullTextOperationHandler` identifier must be used as the value of the `OperationHandler` tag.

Using an external library

It is possible to call an external hook for indexing content using a library other than ARender Rendition, which is the one used internally by FlowerDocs. To do this, you can add the URL of the external hook and the specific mime types on which it will be called.

The external hook url can be built as follows : `http://{ip de la machine host}:{port}/{route}`, for example : http://25.42.62.95:3079/fullText/indexation.

> **INFO**
>
> The user token is automatically transmitted to this external hook : the prerequisite for using an external library is to have token authentication.

Guides / Administration / Operations / Operations managers / Logging

This operation manager is used to log the execution of an operation. A message is logged in the **FlowerDocs Core** log file, indicating the context in which the operation is carried out.

> **INFO**
>
> To manually define this operation handler, the `com.flower.docs.core.tsp.operation.LogOperationHandler` identifier can be used as the value of the `OperationHandler` tag.

Guides / Administration / Operations / Operations managers / Principle

An `OperationHook` is an operation manager exposed as a REST service. An `OperationHook` makes it possible to react to operations carried out on components from a remote WEB service.

Depending on the category of component involved in the operation, a POST request is sent to the following endpoints:

- `/{{scope}}/documents/`
- `/{{scope}}/folder/`
- `/{{scope}}/virtual_folders/`
- `/{{scope}}/tasks/`

It can be developed in any language that can be used to expose WEB services.

The body of requests sent to these endpoints contains an object describing the execution context of the operation, and therefore differs depending on the operation.

Configuring an OperationHook

An `OperationHook` can be configured in the same way as a conventional `OperationHandler`. Its name corresponds to the URL used to access the endpoints listed above. From the configured URL, it should be possible to send a POST to `{{hook URL}}/{{scope}}/documents/`.

Security

It is recommended to secure the `OperationHook` deployed. To do this, FlowerDocs allows you to configure a character string supplied during REST calls in an `Authorization` HTTP header.

Typically, BASIC authentication can be used to secure an `OperationHook`. For FlowerDocs to provide the authentication defined, the subscription must be updated with the authorisation string to be provided.

This character string can be generated online using blitter._

Error management

By default, when running an `OperationHook`, **FlowerDocs Core** logs errors returned by the REST service by parsing the body of the HTTP response.

To return context-sensitive exceptions, it is necessary to provide the `code` and `message` headers on the HTTP response.

With Spring, an `ExceptionHandler` can be easily defined to handle all FlowerDocs exceptions and return them correctly to **FlowerDocs Core**:

```
@ExceptionHandler(CodeBasedException.class)
public ResponseEntity<Object>
handleCustomException(CodeBasedException ex, WebRequest request)

    HttpHeaders headers = new HttpHeaders();
    headers.add("code", ex.getCode());
    headers.add("message", ex.getMessage());
    return new ResponseEntity<Object>(headers,
HttpStatus.INTERNAL_SERVER_ERROR);
```

In the case of synchronous `OperationHooks`, personalized error messages can be sent back to the end user. To do this, use the error code `F00039`:

```
throw ExceptionBuilder.createFunctionalException(F00039, "Custom
error message");
```

Guides / Administration / Appendices / Overview

Two utility objects are available in operation handler scripts. They serve different purposes:

Object	Access	Purpose
`util`	Instance variable	Service access, component persistence, fact creation — any operation that **interacts with FlowerDocs Core**
`RuleUtil`	Static class reference	In-memory component helpers — read/write tags, classes, and statuses **without calling FlowerDocs Core services**

`util` is an instance of `RuleContextUtil` (which extends `RuleUtil`). Because it is an instance, call its methods as `util.methodName(...)`.

`RuleUtil` is a static class reference injected by the script engine. Because all its methods are static, call them as `RuleUtil.methodName(...)`.

INFO

Since `RuleContextUtil` extends `RuleUtil`, calling static methods through the `util` instance (e.g., `util.setTagValue(...)`) also works in GraalJS. However, the recommended convention is to use `RuleUtil` for static calls and `util` for instance calls, to make the intent clear.

Access to services

This object exposes methods for accessing and interacting with the services exposed by **FlowerDocs Core**:

Services | Data model | Identity | Others

Name	Description
`getDocumentService()`	Get document management service
`getVersionService()`	Get document version management service
`getFolderService()`	Get folder management service
`getTaskService()`	Get task management service
`getVirtualFolderService()`	Get virtual folder management service
`getService(Component component)`	Get the appropriate management service for the component type

Component persistence

The following methods can be used to persist or transform a component:

Save Data Task

Methods	Description
`create(Component component)`	Create the component and return the one actually created
`update(Component component)`	Modify the component and return the updated version

RuleUtil - Static helpers

`RuleUtil` methods are **static**: they operate on objects already in memory without making service calls. Use them to read or modify tag values, class identifiers, and statuses on the `component` variable.

Data **Tags** **Task** **Others**

Methods	Description
`getClassId(Component component)`	Get component class value otherwise empty
`setClassId(Component component, String classId)`	Set component class
`getStatus(Component component)`	Get component status value
`setStatus(Component component, Status status)`	Set component status value

Usage examples

Reading tags and calling services

```
// RuleUtil: read tag values in memory (static call)
var clientName = RuleUtil.getTagValue(component, "NomClient");
var classId = RuleUtil.getClassId(component);

// util: call services (instance call)
util.changeClass(component, "NewClass");
util.update(component);

// util: access a service and perform a search
var request = SearchRequestBuilder.init()
    .filter(FilterClauseBuilder.and()

.criterion(CriterionBuilder.name("NomClient").operator(Operators.EQUA
            .type(Types.STRING).value(clientName).build())
        .build())
    .build();
var response = util.getVirtualFolderService().search(request);
```

Guides / Administration / Appendices / ExceptionBuilder

The builder can be used to create TechnicalException or FunctionalException with an error code.

You can pass it one or more parameters, including FlowerDocs identifiers, which will be formatted when the error message is returned.

The following example shows how to create a TechnicalException with error code T00000 and a message indicating the problem :

Java

```
Id tagId = new Id("idMyTag");
        ExceptionBuilder.createTechnicalException(T00000, "Tag {}
cannot have the value {}", tagId, "wrongValue", e);
```

- With the builder : "Tag idMyTag cannot have the value wrongValue".
- Without the builder : "Tag [Id=(value="idMyTag")] cannot have the value wrongValue".

Guides / Administration / Appendices / Principle

The Graal engine is used to execute scripts within the JVM. JavaScript syntax and Java classes loaded into the JVM can be used to add specific logic to a process.

No additional Javascript module is provided.

Instantiate an object

In order to instantiate an object or call a static method of a Java class, the full class name must be specified or a reference to the class must be obtained.

Complete name **Reference**

```
var document = new com.flower.docs.domain.document.Document();
```

To facilitate the development of scripts using the objects provided by FlowerDocs, certain classes can be directly named without using their full name or a reference.

Restrictions

For security reasons, a specific class loader is used to execute the scripts in the JVM. Access to certain classes is therefore restricted. This can be disabled using

the `secured.classloader.enabled=false` property. If required, certain classes or packages can be defined as secured using the `secured.classloader.whitelist.additional` property.

> **INFO**
>
> FlowerDocs uses the [Logback] library (https://logback.qos.ch/) to manage application logs.

Guides / Operation / Log files

The following parameters can be set in the `core.properties` and `gui.properties` files:

Parameter	Default value	Description
`logging.file.name`	*flowerdocs-xxx.log*	Log file path
`logging.file.max-size`	*10MB*	Maximum log file size at which rotation is enabled
`logging.file.max-history`	*undefined*	Maximum number of log files
`logging.level.root`	*INFO*	Default log level

Log levels

FlowerDocs and the libraries used push logs on several levels:

- `TRACE`: the finest level of information, providing debugging information (*can be activated following a support request*)
- `DEBUG`: level for obtaining debugging information (*can be activated following a support request*)
- `INFO`: messages given for information only
- `WARN`: warning messages
- `ERROR`: errors intercepted by the application

The log level can be defined by Java package by adding a `logging.level.<package>=<niveau>` parameter such as:

```
logging.level.com.flower.docs.core.tsp=DEBUG
```

Variables

On a shared or heavily-used platform, logs may need to be contextualized. FlowerDocs provides variables that can be displayed in log messages:

- `user`: current user identifier
- `scope`: identifier of the scope on which the current user is authenticated
- `request`: unique identifier of the request sent (in the case of a request sent to **FlowerDocs GUI** and propagated to **FlowerDocs Core**, the identifier is retained)

To use these variables, simply modify the `logging.pattern.level` parameter of the*appender* in question: `%X{variable}` such as:

```
logging.pattern.level=%X{request} %X{scope} %X{user} %5p
```

> **INFO**
>
> FlowerDocs relies on the Spring Boot Actuator project to provide REST web services to help monitor and manage a FlowerDocs stack.

Guides / Operation / Endpoints

Available endpoints are listed in the `/actuator` path

Endpoint	Description
`info`	Application information (name and version)
`health`	Application health
`env`	Application properties
`caches`	Management of used caches
`loggers`	Log configuration
`logfile`	Downloading the log file
`status`	Application health status (`UP` or `DOWN`)

These endpoints can be disabled by adding a parameter such as:

```
management.endpoints.web.exposure.exclude=caches,env
```

Security

By default, endpoints described, excepted `status`, require *Basic* authentication and the `SYSTEM_ADMIN` role for each application exposing these endpoints. It is also possible to provide a FlowerDocs authentication token for a user with the `SYSTEM_ADMIN` role.

The `status` endpoint is the only one that doesn't requires authentication

The security of these endpoints can be disabled by adding the `actuator.secured=false` parameter.

Guides / Operation / Metrics

The FlowerDocs platform showcases a range of metrics that provide quantitative feedback on the use of **FlowerDocs Core**. These metrics cover various aspects, from service consumption to JVM usage.

Most metrics are displayed with tags, providing contextualized, more precise metrics.

Application metrics

Components

Component metrics are listed in the tables below. They are displayed with tags enabling them to be filtered according to:

- the scope concerned (`scope`)
- the exception triggered (`exception`)

Documents | **Folders** | **Virtual folders** | **Tasks**

Name	Description
`createDocuments`	Create documents
`getDocuments`	Get documents

Name	Description
`searchDocuments`	Search documents
`updateDocuments`	Update documents
`deleteDocuments`	Delete documents
`addFilesDocuments`	Add content to documents
`getFileDocuments`	Get the contents of documents
`getFilesDocuments`	Get the contents of documents
`promoteVersion`	Create a document version
`getVersionsVersion`	Get versions of documents
`revertVersion`	Revert to document versions

OperationHandler

Executions of OperationHandlers can also be monitored using the following metrics:

- `invokeSyncOperationHandler` for those executed synchronously
- `processAsyncOperationHandler` for those executed asynchronously

These metrics are published with the following tags:

Name	Description
`ObjectType`	Object type concerned
`action`	Action to which the OperationHandler reacts
`phase`	Execution phase
`registration`	Identifier of the subscription
`scope`	Identifier of the scope concerned
`exception`	Provoked exception or `none`

A metric is exposed from the moment the operation was first executed.

Other

Dozens of other metrics are exposed, such as exports (ZIP or CSV), conversions and authentications. These may change from version to version and can be browsed in the export system used (see section below).

Technical metrics

The main technical metrics exposed by **FlowerDocs Core** are listed below.

System JVM JDBC

Name	Description
`process.cpu.usage`	CPU usage of the Java process
`process.files.max`	Maximum number of files that can be opened
`process.files.open`	Number of files opened
`process.start.time`	Java process start time
`process.uptime`	Java process uptime
`system.cpu.usage`	Host system CPU usage

Metrics consultation

REST API

The metrics are exposed on the `/actuator/metrics` endpoint and like the other Actuator endpoints. Access requires *basic* authentication by an account with the `SYSTEM_ADMIN` role.

List **Get** **Filter**

```
GET /core/actuator/metrics HTTP/1.1
Authorization: Basic <basic authorization>
```

These web services can be consumed from various monitoring applications. In this case, the use of a specific user account is recommended.

JMX

Metrics are exposed through JMX MBean in the `metrics` namespace. They can, for example, be consulted using tools provided by a JDK such as `jconsole` or `jvisualvm` (with the `VisualVM-MBeans` plugin). As MBeans do not support tag logic, the names of published metrics are concatenated with their tags.

By default, the technology can be accessed locally to monitor the application and consult MBeans.

This technology can also be used remotely, with additional configuration. Typically, by adding the properties below to the JVM, it is possible to access

MBeans **unsecured** from a remote machine.

```
-Dcom.sun.management.jmxremote=true -
Dcom.sun.management.jmxremote.port=6001 -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.rmi.port=6001
```

> **INFO**
>
> See official documentation for more information.

Guides / Operation / Authentication

> **INFO**
>
> *This type of metric is only available when using the OpenSearch connector.*

User connection history is maintained and can be queried on the endpoint `/rest/metrics/profiles`:

- by user profile (or team)
- by date range

These metrics provide information on users who have logged in for a particular profile. To obtain these metrics, you need the `ADMIN` role.

- `/rest/metrics/profiles/{profile}`: Lists the connections for a given profile
- `/rest/metrics/profiles/{profile}/count`: Counts connections for a given profile

This endpoint requires authentication, which must be provided by a user token:

- via a `token` parameter in the URL
- via an HTTP `token` header

Parameter	Description
start	Optional parameter for indicating the start of the date range to be measured. By default, this parameter is set to 24 hours before the end of the date range. `timestamp`
end	Optional parameter for specifying the end of the date range to be measured. By default, this parameter is valued at the present time. `timestamp`

Example:

```
GET /core/rest/metrics/profiles/ALL_USERS?token=<token>
```

Guides / Operation / Qualified browsers

Qualified browsers are tested and validated by Arondor for each product release. We do not rule out the possibility of anomalies on these qualified browsers, but the risk is minimal and restricted to particular or specific cases.

List of qualified navigators:

- Chrome 88
- Mozilla FireFox 85
- Microsoft Edge 88

Arondor undertakes to qualify the latest version of these browsers no later than 6 months after their release.

Supported browsers

Supported browsers correspond to those in which the application is deemed functional. As some browsers are released at a very fast rate, Arondor cannot fully qualify all browser/version combinations for each release. However, the vast majority of intermediate browser versions allow normal use of the application. Browser troubleshooting is covered by standard product support.

List of supported browsers:

- Chrome 58+
- Mozilla FireFox 53+
- Microsoft Edge 16+
- Safari for iOS 11+

Browsers not listed above are not supported as part of standard product support. Extended support for a particular browser version can be considered on a case-by-case basis.

INFO

- For user comfort, performance and visual rendering, we recommend the use of a recent HTML 5-compliant web browser.
- Browser anomalies linked to non-compliance with standards (HTML, JavaScript, etc.) are not covered by product support. A version upgrade of the browsers concerned is recommended.

The CacheGUI service lets you perform getAll, clear and clearAll operations on your scope's GUI caches.

- `getAll` allows you to retrieve all caches in the scope.
- `clearAll` purges all your scope's caches.
- `clear` allows you to purge a list of caches defined by their name.

Guides / Operation / Cache retrieval

The following example shows how to retrieve the list of all FlowerDocs scope caches.

GET ALL:

Cache purge

The examples below show how to purge FlowerDocs scope caches using the various operations of clear.

CLEAR ALL:

CLEAR:

Development / JS Plugins / Getting Started

FlowerDocs provides a number of IHM plugins for customisation and easy integration.

Development / JS Plugins / Principle

The `SelectComponentPlugin` makes it easy for users to select a component corresponding to a set of criteria. The criteria defined are used to execute a component search, the results of which are displayed in a selection popup.

Use

Key	Type	Description
`icon`	String	Selection popup icon
`title`	String	Selection popup title
`description`	String	Description displayed in selection popup header
`category`	String	Component category to search (default: `DOCUMENT`)
`criteria`	Table	List of search criteria
`fieldToDisplay`	String	Field used to display a component (default: `name`)
`callback`	Function	Function called after the selection validation

Basic **With criteria**

```
new SelectComponentPlugin({
    title: "My title",
    description: "Select a component",
    callback: (id, label) => {
        console.log("selected document: " + id);
    },
}).show();
```

Model selection

Based on the `SelectComponentPlugin` plugin, the `SelectTemplatePlugin` plugin offers users a library of document templates. By defining the type of model to be proposed, users can select the model to be used from the library.

Define an option

```
new SelectTemplatePlugin({
  'type': 'MSWord',
  'callback': function(id, label){
    new DownloadWordPlugin({'template': id, 'filename':
label}).download();

}).show();
```

Development / JS Plugins / Principle

Attachment plugins allow an action to be added to the configured attachment insert. This action allows to attach a component to the task, enabling it to be viewed in the viewer.

Several attachment plugins are thus available:

- `SearchAttachmentPlugin`
- `TemplateAttachmentPlugin`
- `CreateHTMLAttachmentPlugin`

A plugin for viewing attachment metadata is also available.

The constructors of these plugins accept a collection of options enabling them to be configured.

Define an option

```
new SearchAttachmentPlugin({
    '<option name>': <option value>
});
```

The following options are common to the various attachment plugins:

Key	Type	Description
`attachmentId`	String	Attachment definition identifier
`title`	String	Action title (title of the selection popup in the case of `SearchAttachmentPlugin`)
`icon`	String	Action Icon (Icon of the selection popup in the case of `SearchAttachmentPlugin`)
`postProcessor`	Function	Function called after attaching the attachment
`canAttach`	Function	Function defining when the action must be present

SearchAttachmentPlugin

The plugin `SearchAttachmentPlugin` adds a search action to an attachment insert. This action opens a popup allowing you to select, from the results of a search, a component to be added as a task attachment.

Key	Type	Description
`template`	String	Identifier of the search form to be displayed in the selection popup (default: `DefaultSearch`)
`category`	String	Component category to search (default: `DOCUMENT`)
`criteria`	Table	Search list of criteria

Basic **With criteria**

```
new SearchAttachmentPlugin({
    attachmentId: '<AttachmentId>',
    title: 'Search for an attachment',
    postProcessor: function(component){
        console.info('Component has been attached: ',
component.getId());

}).bind()
```

TemplateAttachmentPlugin

The plugin `TemplateAttachmentPlugin` adds an action for attaching a document generated from a Microsft Word template. By default, the plugin opens the template in FlowerDocs Companion, if installed on the user workstation. If this is not the case, or if download mode is enabled, then the generated document is downloaded.

Key	Type	Description
`downloadMode`	Boolean	Indicates whether the created document should be downloaded
`instanciator`	Function	Function for programmatically indexing the component

MetadataVisualizationAttachmentPlugin

This plugin allows you to view the data in the attachment without changing its location. The data to be visualized is opened in an OffMenu.

Key	Type	Description
`over`	Boolean	Indicates whether the OffMenu should be displayed on top of the current page

The plugin's OffMenu is closed when it is relocated.

Development / JSAPI / Getting Started

The FlowerDocs graphical user interface can be customised using scripts written in JavaScript. These scripts allow you to use the JS API to enrich and interact with the interface.

To be executed within the browser, these scripts must be loaded from a URL accessible from the user's workstation.

Script class Documents

Documents stored in the DMS with the `Script` class are loaded on the client side. Users who need to load these documents must have `READ` and `READ_CONTENT` permission to access them.

These JavaScript files are cached on the client side and are renewed with each update.

External scripts

External scripts can be loaded client-side by including WEB resources whose URLs are concatenated in the `js.api.scripts` property (separated by `,`). URLs must be accessible from client workstations.

If these scripts are to change, the URLs must be suffixed (e.g. with `?version`) to force the browser to renew them.
To facilitate access to certain actions, several shortcut containers can be manipulated through the JS API:

- `ContextualMenuAPI`: component contextual menu (present on search results tables)
- `MenuShortcutsAPI` : button `+` accessible from the menu bar

There are two ways of accessing these APIs:

- `get()`: access the loaded shortcut container at any time
- `registerForLoad(function(api){});` : subscribe to the loading of a shortcut container

Development / JSAPI / General

The following functions are available on shortcut containers.

Function	Description
addCircled(String id, String icon, String color, String name, String description, Callback callback)	Adds a shortcut with a circle type icon and hover description
addIconized(String id, String icon, String color, String name, Callback callback)	Adds a shortcut with an icon (FontAwesome)
add(String id, Element element, String name, Callback callback)	Adds a shortcut with a DOM element as icon
remove(String id)	Deletes a shortcut using its identifier
getIds()	Recovers shortcut identifiers

Examples

To subscribe to the loading of these shortcut menus, the following functions will be used:

Create a folder from the menu bar

```
MenuShortcutsAPI.get().registerForLoad(function (api) {
    api.addCircled(
        "createFolder",
        "fas fa-folder-open",
        "flat-red",
        "Folder",
        "Create a folder",
        function () {
            var newFolder = new Folder();
            newFolder.setClassId("Folder");
            var popup =
JSAPI.get().getPopupAPI().buildComponentCreation(newFolder);
            popup.show();
        }
    );
});
```

Contextual menu

Additional functions are provided for manipulating a contextual menu:

Function	Description
add(String groupId, String id, String icon, String label, Callback callback)	Adds a shortcut icon to the context menu and to a group in the table header
getSelected()	Retrieves selected components
getCategory()	Retrieves the category of selected components

By using the `add` function, you can group actions according to need, using the same group identifier.

Example: Creating a task from two documents

```
ContextualMenuAPI.get().registerForLoad(function(api){
        if(api.getSelected().length != 2 || api.getCategory() !=
"DOCUMENT"){
                return;

        api.add("customActions", "createTask", "fa fa-user" ,
"Task", function(){
                var newTask = new Task();
                newTask.setClassId("GEC_Step0_Creation");
                TagOracle.predict(api.getSelected(), newTask);
                newTask.addAttachments("Courrier",
Ids.from(api.getSelected()), "DOCUMENT");
                var popup =
JSAPI.get().getPopupAPI().buildComponentCreation(newTask);
                popup.show();
        });
});
```

Responses to a task

If the user selects tasks from the same class and these have responses, these responses will be displayed as actions in the contextual menu.

This feature can be disabled by adding the following JS script:

```
function registerToAddAnswersInTaskContextualMenu() {}
```

> **INFO**
>
> The components are the main objects used by FlowerDocs. Thanks to the JS API, they can be manipulated by scripts to meet specific needs in the development of vertical solutions.

Development / JSAPI / Components / Component instantiation

Each category of component can be manipulated using the JS API through a specific object. The following constructors can be used to instantiate a component:

```
var newDocument = new Document();
var newTask = new Task();
var newFolder = new Folder();
var newVFolder = new VirtualFolder();
```

Each of these objects has its own functions:

- common to all components
- specific to a component category

Access to internal information

Several pieces of information are maintained by the FlowerDocs platform for internal purposes. Depending on the information, the following functions are available to access or modify it (*when modification is authorised*):

Functions	Description
getId()	Component identifier retrieval
setId(String id)	Definition of component identifier
getName()	Component name retrieval
setName(String name)	Component name definition
setClassId(String name)	Component class definition
getClassId()	Component class identifier retrieval
String getCategory()	Category recovery
getACL()	Retrieving the ACL referenced by the component
setACL(String aclId)	Definition of the ACL to be applied
getStatus()	Component status retrieval
setStatus(String status)	Component status definition (see Javadoc)
getVersion()	Component version retrieval

Functions	Description
setVersion(long version)	Component version definition
getCreationDate()	Retrieving the component creation date
getLastUpdateDate()	Retrieving the component update date
getOwner()	Component owner retrieval

Tags management

The tags of a component can be manipulated via the JS API using several functions. The `getTags()` function can be used to determine the tags already present on a component by returning an array containing the names of each of the component's tags.

Adding or modifying a tag

To add a tag to a component, use the `addTag(name, value, readonly)` function. The second parameter `value` can be a character string or an array of strings to define multiple values.

Using this function for a tag already present on the component overwrites existing values. It can be used to modify tag values.

The third parameter `readonly` is a Boolean indicating whether the tag should be displayed as read-only or not.

Tag recovery

To access the values of a tag present on a component, the functions `getTagValue(String name)` and `getTagValues(String name)` are exposed on the component object. The first returns the first value of a tag (*character string*), while the second returns an array containing all the values of a tag.

RETRIEVING THE VALUE OF A TAG ON A TASK

```
var task = new Task();
task.addTag("Amount", "1234", false);
if(task.getTags().includes("Amount")){
    console.info('The amount is ' +
task.getTagValue("Amount"));
```

INFO

The component classes define component typologies. Thanks to the JS API, they can be manipulated by scripts to access a scope's data model.

Development / JSAPI / Components / Retrieving a component class

Each category of component class can be manipulated using the JS API through a specific object. An instance of this object can be retrieved using the following functions:

- Document class: `JSAPI.get().documentClass()`
- Folder class: `JSAPI.get().folderClass()`
- Virtual folder class: `JSAPI.get().virtualFolderClass()`
- Task class: `JSAPI.get().taskClass()`

This object can be used to retrieve component classes in the same category. To achieve this, a `get` function is exhibited. It is used to query the client-side cache first. If one or more classes have not already been cached, the services exhibited by **FlowerDocs Core** will be queried to retrieve the missing classes.

```
JSAPI.get().documentClass().get(["Document"],
        function(classes){
                console.info("Success!");
        },
        function(error){
                console.error("Document classes could not be got: "
+ error);

);
```

Component class information

Component classes contain information used by the Core or graphical user interface. They can be retrieved using the following functions:

Functions	Description
getId()	Component class identifier retrieval
getLocalizedDisplayName()	Retrieval of the internationalized class name according to the current user's language
getLocalizedDescription()	Retrieval of the internationalized class description according to the current user's language
getTagReferences()	Retrieval of tag references carried by the component class

Access to tag references

Each component class has a list of tag references. These references can be used to associate tag classes to a component typology.

From a component class, the `getTagReferences()` function exhibits tag references in the form of an array of objects. The following functions can be used for each of these objects:

Functions	Description
getTagName()	Retrieval of the tag reference name
getLocalizedDescription()	Retrieval of the internationalized reference description according to the current user's language
getPattern()	Retrieval of the tag reference pattern
getDefaultValue()	Retrieval of the default tag reference value
getOrder()	Retrieval of the position of the tag reference
isMandatory()	Determines whether the tag is mandatory
isMultivalued()	Determines whether the tag is multivalued
isReadonly()	Determines whether the tag is read-only
isTechnical()	Determines whether the tag is technical

INFO

This section describes how to consume services exposed by FlowerDocs directly from the JS API.

Development / JSAPI / Components / Access to a component service

Like the service layer, a service is provided for each component category:

- Document: `JSAPI.get().document()`
- Folder: `JSAPI.get().folder()`
- Virtual folder: `JSAPI.get().virtualFolder()`
- Task: `JSAPI.get().task()`

These services consume the SOAP web services exposed by **FlowerDocs Core**, so their use implies the application of the same consistency and security controls.

Each function exposed by these services accepts two function-type parameters:

- `successCallback`: a callback executed on success
- `errorCallback`: an optional callback executed in the event of an error

Creating components

After instantiating a [component] object (documentation/apis/jsapi/mcd/components.md) and initialized it, it can be created in FlowerDocs using the appropriate service. Component services expose the function `create(components, successCallback, errorCallback)`. The first parameter of this function is an array of the components to be created. The `successCallback` parameter is a function called asynchronously in the event of successful creation, with the array of components created as parameter. The third parameter `errorCallback` is an optional function executed if an error occurs.

Create a folder

```
var FOLDER = new Folder();
folder.setName(‘My Folder');
folder.setClassId('Folder');
JSAPI.get().folder().create([folder], function (created) {
  console.info('The folder ' + created[0].getName() + ' has been
created');
}, function(error){
  console.error(error);
});
```

Recovering components

A component service is used to retrieve stored components from the services exposed by **FlowerDocs Core**. The first parameter `ids` to be supplied is an array of component identifiers to be retrieved. The other two function-type parameters can be supplied to react respectively to the success and error of the operation.

Get a document

```
JSAPI.get()
    .document()
    .get(
        [id],
        function (documents) {
            console.info("The document " + documents[0].getName() +
" has been recovered");
        },
        function (error) {
            console.error(error);
        }
    );
```

A component can also be retrieved dynamically according to its category:

```
var reference = new ComponentReference(id, "DOCUMENT");
JSAPI.get()
    .getComponentGetAPI()
    .getComponent(
        reference.getCategory(),
        reference.getId(),
        function (doc) {
            console.info("The document " + doc.getName() + " has
been recovered");
        },
        function (error) {
            console.error(error);
        }
    );
```

Find components

Components can be searched using component services and a SearchRequest supplied as a parameter to the function `search(request, successCallback, errorCallback)`.

The `successCallback` parameter is a function called once the search has been successful, with the table of results found in accordance with the query supplied. Optionally, the `errorCallback` function can be passed as a parameter to react to any error.

Search tasks

```
var request = new SearchRequest();
JSAPI.get()
    .task()
    .search(request, function (results) {
        console.info(results.length + " results have been
retrieved");
        results.forEach((result) => console.log(result.getId()));
    });
```

Updating components

Just as component services can be used to create components, they can also be used to update components. To do this, the `update(components, successCallback, errorCallback)` function is exposed. It is used in the same way as the create function.

Update document

```
JSAPI.get()
    .document()
    .get([id], function (documents) {
        documents[0].setName("updated");
        JSAPI.get()
            .document()
            .update([documents[0]], function (updated) {
                console.info("The document " + updated[0].getName()
+ " has been updated");
            });
    });
```

Delete components

To physically (or permanently) remove components, the `doDelete(ids, successCallback, errorCallback)` function is available. The first parameter `ids` to be supplied is an array of the identifiers of the components to be removed. The other two function-type parameters can be supplied to react respectively to the success and error of the operation.

Please note *this operation is irreversible* and should therefore be used with caution.

Delete document

```
JSAPI.get()
    .document()
    .doDelete([id], function () {
        console.info("The document " + id + " has been permanently
deleted");
    });
```

Development / JSAPI / Components / Handling files

In addition to the tags associated with a document, files can be associated with it. The JS API enables them to be manipulated through various consultation or modification functions.

File access

From a `Document` object, the `getDocumentFiles()` function reveals an array of `DocumentFile` objects from which the following functions can be used:

Functions	Description
getId()	File identifier recovery
setId(String id)	File identifier definition
getName()	File name retrieval
setName(String name)	File name definition
getFormatCode()	File format retrieval
setFormatCode(String formatCode)	File format definition

Functions	Description
getCreationDate()	Retrieving the file creation date

List files in a document **Get document files**

```
var doc = formAPI.getComponent();
doc.getDocumentFiles().forEach(function (file) {
    console.info("The document has file: " + file.getId());
});
```

File identifiers can also be retrieved from a document using the `getFiles()` function.

Files modification

The list of files in a `Document` object can be initialised or modified using the JS API. To achieve this, the functions `addFile(tempFileId)` or `setFiles(tempFileIds)` can be used. Changes made to the object will only be taken into account when a document is created.

Create a document with a text file

```
function createDocument(tempFileId, callback){
    var doc = new Document();
    doc.setName('My document');
    doc.setClassId('Document');
    doc.addFile(tempFileId);
    JSAPI.get().document().create([doc],function(created){
        callback(created[0]);
    });

function uploadTextFile(text, callback){
    var formData = new FormData();
    formData.append('file', new File([new Blob([text])], 'my-
file.txt'));
    $.ajax({
        url: './upload',
        data: formData,
        processData: false,
        contentType: false,
        type: 'POST',
        success: function (data) {
            callback(data.split('|')[0]);

    });

uploadTextFile('Some text content', (tempFileId)=>{
    createDocument(tempFileId,(doc)=>{

JSAPI.get().getNavigationAPI().goToComponentPlace(doc.getCategory(),
doc.getId(), false);
    });
});
```

Versioning

If version management (or *versioning*) is enabled for a document, information about its different versions can be consulted using the following functions:

Functions	Description
getVersionSeriesId()	Retrieving the VersionSeries identifier
getVersionLabel()	Retrieving the label of the current version
isCurrentVersion()	Determines whether the document is the current version

A folder can contain components: its children. A folder's children can be components of any category, but only documents and sub-folders are displayed in **FlowerDocs GUI**.

Functions	Description
addChildren(String folderId, ComponentReference[] children, boolean replace)	Adding children to a folder
deleteChildren(String folderId, ComponentReference[] children)	Deleting children from a folder

Development / JSAPI / Components / Adding content

To add a component to a folder, it must be referenced as a child of the folder using a `ComponentReference` object. From this reference, the `addChildren(id, childReferences, replace, successCallback)` function revealed by the `JSAPI.get().folder()` service can be used. The Boolean `replace` is used to indicate whether the content of the folder should be replaced by the new referenced children.

Add document to a folder

```
var childReference = new ComponentReference("documentId",
"DOCUMENT");
JSAPI.get()
    .folder()
    .addChildren("folderId", [childReference], false, function () {
        console.info("The document has been added as child");
    });
```

Deleting content

A component can be deleted from a folder so that it is no longer referenced as one of its children, using the `deleteChildren(folderId, childReferences, successCallback) function`

Delete document from a folder

```
var childReference = new ComponentReference("documentId",
"DOCUMENT");
JSAPI.get().folder().deleteChildren(folderId, [childReference],
function () {
        console.info('The document has been removed from folder'");
});
```

Development / JSAPI / Components / Handling attachments

To contextualize task processing, components can be attached to a task. This is the notion of an attachment. Task attachments can be manipulated using the JS API.

The following two functions can be used to add components as task attachments:

- `addAttachment(attachmentId, componentId, category)` to attach a single component from its identifier `componentId`
- `addAttachments(attachmentId, componentIds, category)` to attach several components based on their identifier provided in an array `componentIds`

Use of these functions requires input of the attachment identifier and the expected component category (defined in the task class). Changes to task attachments need to be saved using the `create` or `update` functions exposed by the `JSAPI.get().task()` object.

Attach document to a task

```
JSAPI.get()
    .task()
    .get([id], function (tasks) {
        let task = tasks[0];
        task.addAttachments("Appendices", documentId, "DOCUMENT");
        JSAPI.get()
            .task()
            .update([task], function (updated) {
                console.info("The document has been attached to
task");
            });
    });
```

Based on the same principle, the following function retrieves existing attachments:

- `getAttachmentIds(attachmentId)` to retrieve the component identifiers linked to this attachment.

Responding to tasks

The notion of task responses enables users to make decisions about task processing. For each decision taken, a specific behavior can thus be added to take account of this decision and apply the appropriate treatment.

An answer can be applied to one or more tasks using the function `answer(ids, answer)` with the following parameters :

- a table of task identifiers
- a `Answer` (or `ReasonedAnswer`) object representing the answer to be applied

Answer to a task

```
var taskAPI = JSAPI.get().task();
taskAPI.answer([id], new Answer("Validate"), function () {
    JSAPI.get().getNotificationAPI().sendInformation("Answer was
applied");
});
```

How to assign tasks

Assigning tasks

A task must be assigned to a user in order to be processed. FlowerDocs provides several assignment mechanisms, including the JS API. The `assign(ids, userId)` function lets you assign one or more tasks to a user using :

- a table of task identifiers
- user ID to assign tasks to

Assign task to a user

```
JSAPI.get()
    .task()
    .assign([id], "sarah.hubert", function () {
        console.info("The task " + id + " has been assigned");
    });
```

The user to whom a task is assigned can be retrieved using the `getAssignee()` function exposed on the `Task` object.

Responding to an assignment

Assigning a task to a user may in some cases require actions to be taken in the graphical user interface. To achieve this, the JS API provides hooks that allow you to react to the launch of the assignment process with a native GUI action:

- from an indexing form with assignment and appropriation actions
- from the context menu opened from search results with these same actions

These hooks can be used to react or interrupt the process, either before the request is sent to **FlowerDocs Core** or afterwards. An assignment hook is a function taking as input the following three parameters (supplied by **FlowerDocs GUI**) :

- a table of tasks to be assigned
- user ID to assign tasks to
- a `executor` object to block execution of the assignment and/or resume it later with its context

React before assign **React after assign**

```
JSAPI.get()
    .task()
    .registerBeforeAssign(function (tasks, assignee, executor) {
        // Action to be taken before assignment
    });
```

The `executor` object supplied as hook input is used to block the assignment process in order to wait for asynchronous processing. In fact, when asynchronous processing needs to be finalized before the assignment process is resumed, it can be interrupted by calling the `executor.hold()` function and resumed once completed with `executor.resume()`.

In the examples below, asynchronous processing is simulated by using a 5-second timeout:

React before assign **React after assign**

```
JSAPI.get()
    .task()
    .registerBeforeAssign(function (tasks, assignee, executor) {
        executor.hold();
        setTimeout(function () {
            executor.resume();
        }, 5000);
    });
```

Process information

Functions	Description
getWorkflow()	Determine task processing identifier
setWorkflow(String workflow)	Define task processing identifier
getParticipants()	Identify the identities involved in processing
addParticipant(String participant)	Adding a participant to a task

INFO

Forms are the basis of the screens presented in the graphical user interface. This section lists how to acquire a `formAPI` object for interacting with a form, and the associated functions.

Development / JSAPI / Forms / Simple fields

Note: In this section, the variable `fieldName` corresponds, in the case of a tag, to the value of its identifier. If this is the component class, use `Class`.

Function	Description
hasField(String fieldName)	Determines whether the field exists in the form
setObjectValue(String fieldName, Object fieldValue)	Modifies the value of a field
getObjectValue(String fieldName)	Retrieves the value of a field
suggest(String fieldName, LookupResult suggestions)	Displays a list of suggestions for a STRING field
setDescription(String fieldName, String description)	Modifies the value of a field description

Some examples:

- Modification of a `EndDate` tag of type `TIMESTAMP`:

```
// Set value from a Date object
formAPI.setObjectValue("EndDate", new Date(2016, 11, 04));
// Definition based on a timestamp (number of milliseconds elapsed since 01/01/1970)
formAPI.setObjectValue("EndDate", 0);
```

- Modification of a `Received` tag of type `BOOLEAN`:

```
// Set the value from a Boolean
formAPI.setObjectValue("Received", true);
```

- Modification of class identifier:

```
formAPI.setObjectValue("Class", "Document");
```

Suggestion of values

To suggest values for a field to the user, you can use the function `suggest(fieldName, suggestions)`.

The `suggestions` to be passed as parameters are composed of a symbolic name (`setValue(symbolicName)`) and a label (`setName(displayName)`). So if the user selects one of the suggestions, the form will be submitted with the symbolic name.

```
function buildSuggestion(name, value){
        var suggestion = new LookupResult();
        suggestion.setName(name);
        suggestion.setValue(value);
        return suggestion;

var suggestions = new Array();

suggestions[0] = buildSuggestion("name1", "value1");
suggestions[1] = buildSuggestion("name2", "value2");

formAPI.suggest("MailObject", suggestions);
```

Choice lists

Function	Description
getAllowedValues(String fieldName)	Retrieves all permitted values for a CHOICELIST field
setAllowedValues(String fieldName, AllowedValueDefinition[] allowedValues)	Modifies the allowed values for a CHOICELIST field

```
function buildAllowedValue(symbolicName, label) {
        var language = new Language("EN");

        var allowedValue = new AllowedValueDefinition();
        allowedValue.setSymbolicName(symbolicName);

        var displayNames = new I18NLabel()
        displayNames.setLabel(language, label);
        allowedValue.setDisplayNames(displayNames);
        return allowedValue;

var restrictedAllowedValues = new Array();

restrictedAllowedValues[0] = buildAllowedValue("symbolicName1",
"firstLabel");
restrictedAllowedValues[1] = buildAllowedValue("symbolicName2",
"secondLabel");

formAPI.setAllowedValues("BillType", restrictedAllowedValues);
```

Field status

Several functions can be used to change the status of a field within a form. They are used with the parameters:

- `fieldName` corresponding to the field identifier
- `boolean` corresponding to state activation

Function	Description
setReadOnly(String fieldName, boolean isReadOnly)	Defines whether the field is read-only or not
getTagValidity(String fieldName)	Retrieves the validity of a field
setValid(String fieldName, boolean valid)	Modifies the validity of a field If the parameter `valid` is `false` then the form submission cannot be validated
setVisible(String fieldName, boolean isVisible)	Defines whether the field is visible or hidden

Example: Changing the visibility of a field in an indexing form

```
formAPI.setVisible("BillType", false);
```

Form validity change subscription

Various functions allow you to subscribe to changes in the validity of a form, depending on your needs:

- Validity of the complete form, i.e. fields and content:

```
formAPI.registerForValidity(function (isValid, invalidFields,
invalidTypes) {
    console.log(
        "Component is valid=" +
            isValid +
            ", invalidFields=" +
            invalidFields +
            ", invalidTypes=" +
            invalidTypes
    );
});
```

- Partial validity of the form:

This function allows you to subscribe only to the validity of a part of the form, either the fields (`FIELDS`), or the content (`CONTENT`).

In the following example, we only subscribe to changes in field validity.

```
formAPI.registerForPartialValidity("FIELDS", function (isValid,
type, invalidFields) {
    console.log(
        "Component is valid=" + isValid + ", invalidFields=" +
invalidFields + ", type=" + type
    );
});
```

The following functions are also available to find out the status of the current form:

Function	Description
getInvalidType(String type)	Retrieves validity from the name of a form part: `FIELDS` for invalid fields or `CONTENT` for invalid content / attachments
getInvalidTypes()	Retrieves names of invalid parts of the form

List form fields

The `formAPI.getFields()` function on a form retrieves the list of field identifiers present in the form.

Value change

To provide your users with interactive forms, it is possible to subscribe to changes in the value of a field in order to modify its status or modify another field.

```
formAPI.registerForFieldChange("Amount", function (fieldName,
fieldValue) {
    // Use the new value of a field
});
```

Development / JSAPI / Forms / Indexing

An indexing form lets you view and modify a component's tags. In order to interact with this type of form, it is necessary to subscribe to its opening:

```
JSAPI.get().registerForComponentChange(function (componentFormAPI, component, phase) {
    // Use the componentFormAPI object to interact with the form
});
```

Note: When displaying several component forms, it may be necessary to access a specific form: `JSAPI.get().getComponentFormAPI(<identifiant du composant>)`.

Actions in a form

The object `componentFormAPI.getActions()` is used to interact with the actions of a form presenting a document, folder...

A number of functions are available for this purpose.

Function	Description
getHeaderActions()	Retrieves action container from header
getTaskActions()	Retrieves task action container

Function	Description
getFooterActions()	Retrieves action container from footer (validation, cancellation...)

Section Actions documents the functions available for interacting with the various actions' containers and the actions retrieved from these 3 containers.

Adding cards

An indexing form is made up of a set of cards containing component information, tags, attachments and parent folders. You can also add cards via the JS API with the `addCardContainer(cardContainer)` function, which requires a container card as input.

The functions available on this object are available in the [Cards] section (documentation/apis/jsapi/cards.md).

Example: Adding a customised container card

```
var desc1 = buildDescription("Description 1");
var card1 = buildCard("Card", "Heading card", "card-style");

var div = document.createElement("div");
div.className="custom-card-container";
addChildren(div, [desc1,card1]);

var cardContainer = buildCardContainer("fa fa-
clone","Title","Caption","card-box custom-card-container";
cardContainer.setContent(div);
formAPI.addCardContainer(cardContainer);

function buildCardContainer(icon,title,caption, style){
        var cardContainer = new CardContainer();
        cardContainer.setIcon(icon);
        cardContainer.setTitle(title);
        cardContainer.setCaption(caption);
        cardContainer.setVisible(true);
        cardContainer.addStyle(style);
        return cardContainer;

function buildDescription(description){
        var desc = document.createElement("div");
        desc.innerHTML = description;
        return desc;

function buildCard(title, heading, style){
        var card = new Card();
        card.setTitle(title);
        card.setHeading(heading);
        card.addStyle(style)
        return card.asElement();

function addChildren(parent, children){
        for (i=0; i < children.length; ++i) {
        parent.appendChild(children[i]);
```

Form status

The `formAPI.isDirty()` function on a form can be used to find out whether the form has been modified.

Development / JSAPI / Forms / Subscription

Form

A search form allows users to enter criteria to access stored components. In order to interact with this type of form, it is necessary to subscribe to its opening:

```
JSAPI.get().registerForSearchOpen(function (searchFormAPI, id) {
    console.log("Hidden request: " +
searchFormAPI.getHiddenRequest());
});
```

Note : In the case of a form from the search screen, the `id` variable corresponds to the identifier of the form *template*.
In the case of a search form from a virtual folder, the `id` variable corresponds to the virtual folder identifier.

Criteria

If you wish to react to a change in the value of a specific search criterion, subscribe as follows:

```
criterionName = "EDS";
JSAPI.get()
    .getSearchFormAPI()
    .registerForCriterionChange(criterionName, function (criterion)
{
        console.log(
            "Criterion " + criterion.getName() + " values changed
to " + criterion.getValues()
        );
    });
```

Note : In this section, the `criterionName` variable corresponds, in the case of a tag, to the value of its identifier. If this is the component class, use `classid`.

The functions available for interacting with the criteria are available in the following section.

Actions

A search form has two action containers available through the following functions:

Function	Description
getHeaderActions()	Retrieves the action container above the search results
getFooterActions()	Retrieves the action container below the search results

The example below illustrates how to add an action to the container below the results table.

```
JSAPI.get().registerForSearchOpen(function (searchFormAPI, id) {
    console.error("Call of register for open");
    var actionAPI = JSAPI.get().getActionFactoryAPI();
    var action = actionAPI.buildTextual("myCustomAction", "Test",
function (actionPresenter) {
        console.info("on click");
        actionPresenter.setEnabled(false);
    });

    var footerActions = searchFormAPI.getFooterActions();
    footerActions.remove(action.getId());
    footerActions.add(action);
});
```

Development / JSAPI / Forms / Reasoned answer

The functions available on reasoned answers API are:

Function	Description
getPopup()	Retrieves response popup
registerForFieldChange(String fieldName, function callback)	Allows subscription to field modification in response popup
getReasonedAnswerId()	Retrieves action identifier
getTasks()	Retrieves the list of tasks associated with the response

Examples:

```
JSAPI.get().registerForReasonedAnswerOpen(function
(reasonedAnswerAPI, reasonedAnswerId) {
    console.log("Opened reasoned answer Id: " +
reasonedAnswerAPI.getReasonedAnswerId());
});
```

```
JSAPI.get().getReasonedAnswerAPI(<answer
identifier>).registerForFieldChange("Comments", function(fieldName,
fieldValue) {
    console.log("Value of " + fieldName + " changed to: " +
fieldValue);
});
```

Note: In this part, the variable `reasonedAnswerId` is used to identify the response just opened

Please note: When displaying several reasoned answers forms, it may be necessary to access a particular form: `JSAPI.get().getReasonedAnswerAPI(<answer identifier>)`.

Development / JSAPI / Forms / Document as attachment without indexing

It is possible to use the old mode of document creation as an attachment without an indexing form by defining a strategy to activate the *Legacy*mode.

In the following example, attachments with the identifier `Appendices` use the old mode, the others the new.

```
LegacyDocumentAttachmentCreationStrategy.registerStrategy(function(ta
{
        if("Appendices" == attachment.getId()){
                return true;

    return false;
});
```

Summary

Attachment summaries can be overloaded using the `TaskAttachmentSynopsisHelper` API.

Example: Define part summary based on parent task tags

```
helper =
JSAPI.get().getHelperFactory().getTaskAttachmentSynopsisHelper();
helper.register(function (task, definition, attachedComponent,
callback) {
    new TemplateResolver("TASK").resolveTemplate(task,
"${Priority}", function (resolved) {
        console.log("Resolved template: " + resolved);
        callback.onSuccess(resolved);
    });
});
```

Example: Define the summary of an unsigned attachment

```
helper =
JSAPI.get().getHelperFactory().getTaskAttachmentSynopsisHelper();
helper.register(function(task, definition, attachedComponent,
callback){
        if(!attachedComponent){
                callback.onSuccess("Attachment to define");

        else{
                //Using the default summary
                callback.onSuccess();

});
```

To search for components via the JS API, use the `search` function for the various service APIs **here**.

Development / JSAPI / Search / Queries, filters and criteria

Objects :

- `SearchRequest`

Function	Description
addSelect(String field)	Adds a field to be remounted
addFilterClause(FilterClause filterClause)	Adds an additional criterion
getFilters()	Retrieves an array containing all `FilterClause`
addOrderClause(OrderClause orderClause)	Adds a field on which to sort search results
setMax(int max)	Defines the maximum number of results to be returned
getMax()	Retrieves the maximum number of results to return
setStart(int start)	Defines the start of the search page
getStart()	Retrieves the start of the search page

- `AndClause`

Function	Description
addCriterion(Criterion criterion)	Add a search criterion
getCriteria()	Retrieves an array containing all the criteria of the clause

- `Criterion`

Function	Description
getName()	Retrieves the name of the criterion
setName(String name)	Defines the name of the criterion
getOperator()	Retrieves the criterion operator
setOperator(String operator	Defines the criterion operator. Possible values are : `EQUALS_TO` `CONTAINS`, `LESS_THAN`, `GREATER_THAN`, `STARTS_WITH`, `ENDS_WITH`, `DISPLAY`, `DIFFERENT`, `BETWEEN`
getType()	Retrieves the criterion type
setType(String type)	Defines the criterion type. Possible values are : `STRING`, `TIMESTAMP`, `BOOLEAN`, `INTEGER`, `CURRENCY`

Function	Description
getValues()	Retrieves criterion values
addValue(String value)	Adds a value to the criterion
addValues(String[] values)	Adds an array of values to the criterion

INFO

The `EQUALS_TO` and `DIFFERENT` operators are case-sensitive (differentiate between upper and lower case).

Sorting

- `OrderClause`

Function	Description
setName(String name)	Defines the name of the criterion
setType(String type)	Defines the type of field: `STRING`, `TIMESTAMP`, `BOOLEAN`, `INTEGER`, `CURRENCY`
setAscending(boolean isAscending)	Defines whether sorting is ascending or descending

Building a query :

```
var request = new SearchRequest();
request.setMax(20);
request.setStart(0);

var orderClause = new OrderClause();
orderClause.setName("creationDate");
orderClause.setType("TIMESTAMP");
orderClause.setAscending(true);
request.addOrderClause(orderClause);

var filters = new AndClause();
request.addFilterClause(filters);

var criterion = new Criterion();
criterion.setName("Matricule");
criterion.setType("STRING");
criterion.addValue("P0002095");
filters.addCriterion(criterion);

JSAPI.get().document().search(request, function(results)

                // Simple dump of all results
                console.log("results.length=" + results.length);
                for (var i = 0; i < results.length; i++)

                        var result = results[i];
                        console.log("id=" + result.getId() + ",
name=" + result.getFieldValue("name"));

                // Jump to first document found
                if ( results.length > 0 )

                        var result = results[0];
                        var id = result.getId();

JSAPI.get().getNavigationAPI().goToComponentPlace("DOCUMENT", id);
```

```
        },
        function(error){
                console.error("Error on document search: " +
error);
        });
```

It is possible to obtain the values of a search result in two different ways:

- `getFieldValue(fieldName)` : Retrieves all existing values for this field as a string. If the field contains several values, they are separated by the `§` character.
- `getFieldValues(fieldName)` : Retrieves all existing values for this field as an array of strings.

Development / JSAPI / Search / Date format search results

The date format can be adapted to suit your needs, depending on tags and results. To do this, simply define the JS function `getSearchResultDateFormat` to return the desired date format.

The settings are string as follows:

- `tagClassId`: tag class identifier
- `result`: the search result

An example of use:

```
function getSearchResultDateFormat(tagClassId, result){
        if(tagClassId == "MonTagAModifier"){
                return 'dd/MM/yyyyy';

    return 'dd MM yyyy HH:mm';
```

Decorate your search results

You can decorate your search results: modifying the displayed value of a given field. To do this, simply define a decorator for the tag to be decorated.

In the following example, we define a decorator for the field `fieldName` which will be called to resolve the value to be displayed for this field.

```
function decorate(fieldName){
        var decorators =
JSAPI.get().getHelperFactory().getSearchDecoratorsAPI();
        decorators.register(fieldName, function(result, callback){
                callback.onSuccess("decorated: " +
result.getField(fieldName).getValue());
        });

decorate("Personal number");
```

Formatting tag values

To display a tag, FlowerDocs formats its values:

- for dates, according to the defined format
- for value lists, according to defined labels

This mechanism can be used through the JS API, as shown in the following example:

```
var formatter =
JSAPI.get().getHelperFactory().getFieldFormatter("DOCUMENT");
formatter.format(result.getField(fieldName), function
(formattedValue) {
    console.log("Formatted value: " + formattedValue);
});
```

Development / JSAPI / Search / Subscription

Subscription to the execution of a search request is based on the search template. The information available when subscribing is the template name, request, response and response callback. The response can be manipulated before the callback is executed.

In the following example, a component creation popup is displayed if the search returns no results:

```
JSAPI.get().getComponentSearchAPI().registerForExecution("DefaultSear
response, callback){
    callback.onProcessed(response);
    if(response.getFound() == 0){
        JSAPI.get().getPopupAPI().buildComponentCreationFromSearchReq
request, null).show();

});
```

Development / JSAPI / Search / "Managing buckets"

The buckets (or aggregated results) of a search can be organised using the JS API.

Subscription

The first step, before being able to manipulate these buckets, is to subscribe to buckets retrieval from **FlowerDocs Core**:

- or by search, providing `searchId` the identifier of the search in question:

```
var bucketAPI = JSAPI.get().getHelperFactory().getBucketAPI();
bucketAPI.register(searchId, function (buckets, callback) {
    callback.onProcessed(buckets);
});
```

- or for all searches:

```
var bucketAPI = JSAPI.get().getHelperFactory().getBucketAPI();
bucketAPI.register(, function(buckets, callback){
        callback.onProcessed(buckets);
});
```

The registered subscription is called up with two variables:

- The `buckets` array contains all buckets of the same level
- The `callback` provided as input to the subscription must be called with a bucket array

EXAMPLE OF SORTING BUCKETS

```
var orderMap = {};

orderMap[STATUS_1] = 0;
orderMap[STATUS_2] = 1;

var bucketAPI = JSAPI.get().getHelperFactory().getBucketAPI();
bucketAPI.register("searchId", function(buckets, callback){
    var ordered = new Array();
        for(i in buckets){
                var bucket = buckets[i];
                var index = getBucketIndex(bucket);
        var bucket = buckets[i];

        callback.onProcessed(ordered);
});

function getBucketIndex(bucket){
        var request = bucket.getRequest();
    var filter = request.getFilters()[0];
    var statusName = filter.getCriteria()[0].getValues()[0];
        return orderMap[statusName];
```

INFO

Deleting or adding buckets is not supported. The total number of buckets determined must be equal to that determined by **FlowerDocs Core**.

Bucket modification

The JS API lets you modify buckets resolved by **FlowerDocs Core** by exhibiting the following methods:

Function	Description
getName()	Returns the name
setName(name)	Modifies the name
getCount()	Returns the number of components contained in the bucket (sum of child buckets)
setCount(long count)	Modifies the number of components contained in the bucket
getLevel()	Returns bucket level
hasChildren()	Determines whether the bucket contains child buckets
getChildren()	Retrieves child buckets
hasParent()	Determines whether the bucket has a bucket parent
getParents()	Retrieves parent buckets (all parent levels)

Function	Description
getRequest()	Retrieves the request executed to determine the bucket's contents
setRequest(SearchRequest request)	Modifies the request executed to determine the bucket's contents

> **INFO**
>
> Children and parents buckets are provided for information only. Modifications to these are not taken into account.

Hiding a bucket

The `setSkipDisplay(boolean skip)` function is also exhibited on the `Bucket` object, enabling a bucket to be hidden and only its children displayed.

EXAMPLE: HIDING BUCKETS WITH THE SAME NAME

```
var bucketAPI = JSAPI.get().getHelperFactory().getBucketAPI();
bucketAPI.register(function(buckets, callback){
        var ordered = new Array();
        for(i in buckets){
                var bucket = buckets[i];
                if(bucket.hasChildren() &&
bucket.getChildren().length ==1){
                        if(bucket.getName().toUpperCase() ==
bucket.getChildren()[0].getName().toUpperCase()){
                                bucket.setSkipDisplay(true);


                ordered.push(bucket);

        callback.onProcessed(ordered);
});
```

Recovery of buckets after resolution

The buckets of a search can be recovered after resolution as follows:

```
var bucketAPI = JSAPI.get().getHelperFactory().getBucketAPI();
bucketAPI.registerForResolved("searchId", function(buckets){
   for(i in buckets){
        var bucket = buckets[i];
                console.info(bucket.getName()+ " has been resolved,
count=" +bucket.getCount());

   callback.onProcessed(buckets);
});
```

This allows you to obtain resolved buckets, including counters in the case of partial buckets.

Development / JSAPI / Search / Task's serial processing

For more efficient processing of a task list, it is possible to define serial processing sessions. During these sessions, the user will move on to the next task without going back to the search that initiated the session.

The start of these processing sessions is configured using the JS API:

```
JSAPI.get()
    .batchSession()
    .registerBuilder(function (session, callback) {
        callback.enable(session);
    });
```

The context at the start of the processing session (search request) is retained during the processing session. However, the JS API allows you to modify the session by exposing the following methods:

Function	Description
getRequest()	Retrieves the query executed during the processing session
setRequest(SearchRequest request)	Modifies the query executed during the processing session

Function	Description
getPlace()	Retrieves the current place
getSourcePlace()	Retrieves the place at the origin of the processing session
getComponentSource()	Retrieves the component at the origin of the processing session
getLast()	Retrieves the identifier of the last component opened during the processing session
setEnabled(boolean enabled)	Disables the treatment session
isEnabled()	Determines whether the processing session is active or inactive

Several types of actions are available:

- Button: text presented as a button
- Icon: based on the icon font supported by FlowerDocs Font Awesome 6.5.2
- DOM: HTML element

Development / JSAPI / Actions' container

Actions are grouped into actions' containers from which the actions can be accessed to. To access an action, it is therefore necessary to identify which container is concerned.

To list the actions present in a container `container`, use the following function:

```
console.log("Container actions: " + container.getIds());
```

All FlowerDocs actions' containers are accessible via the API JS. This allows them to be manipulated. For example, you can add, delete, deactivate, hide or modify actions in this container.

In a form

The object `componentFormAPI.getActions()` is used to interact with the actions of a form presenting a document, folder...

Several functions are available for this purpose:

Function	Description
getHeaderActions()	Retrieves action container from header
getTaskActions()	Retrieves action container for task creation

Function	Description
getFooterActions()	Retrieves the footer action container (validation, cancellation...)

Go to an action

From an action container `container`, there are several ways to access an action.

- Synchronously (which requires the action to have already been added to the container at runtime):

```
var action = container.get(actionId);
```

- Asynchronously:

```
container.registerForAdd(function (action) {});
```

> **INFO**
>
> Any customization of existing actions must be carried out in the `registerForAdd` to ensure that the action is loaded onto the form.

Actions

Available functions

Once an action has been retrieved, you can interact with it using the functions provided:

Functions	Description
setEnabled(boolean enabled)	Activates or deactivates the action
isEnabled()	Determines whether the action is activated or not
setTemporaryDisabled(boolean disabled)	Temporarily disables an action (for buttons only)
getId()	Retrieves action identifier
execute()	Executes action programmatically

Example: Disabling the action `Cancel` of a component

```
formAPI.getActions().getFooterActions().registerForAdd(function(actio
{
    if(action.getName() == "delete") {
                action.setEnabled(false);

});
```

Interactions with component actions

The execution of an action can be reacted at within a component indexing form.

To do this, simply use the function `registerForActionConfirmation(container, name, function(value, executor){})`:

- `container`: string correspoonfing to the action's container, either `header`, `footer` or `task`
- `name`: string correspoonfing to the name of the action
- `function(value, executor)`: the function to be executed when the user interacts with the action

Example: To call a function each time a user clicks on the button `Cancel` on an indexing page

```
formAPI.getActions().getFooterActions().registerForAdd(function(actio
{
    if(action.getName() == "cancel") {

formAPI.getActions().registerForActionConfirmation("footer", "cancel"
function(value) { ...}

});
```

For even greater control over actions, the object `executor` supplied as a parameter to the closure can be used to:

- Block action execution:

```
executor.hold();
```

- Resume action execution:

```
executor.resume();
```

- Determine whether action execution is blocked or not:

```
executor.isHeld();
```

Note: if an action is not blocked, it will resume once the closure has been executed.

Examples: Blocking the execution of an action until an asynchronous call returns

```
var actions = formAPI.getActions();
actions.registerForActionConfirmation("footer", “Validate",
function(value, executor) {
    executor.hold();
    setTimeout(function(){
        executor.resume();
    }, 3000);
});
```

```
var actions = formAPI.getActions();
actions.registerForActionConfirmation("footer", “Refuse",
function(value, executor) {
    executor.hold();
    setTimeout(function(){
        JSAPI.get().getNotificationAPI().sendError("The task cannot
be refused.");
    }, 3000);
});
```

Build an action

Customized actions can be created thanks to API `ActionFactoryAPI`

Button action

Two types of button action can be created:

- textual or main ones:

```
JSAPI.get().getActionFactoryAPI().buildTextual(String id, String
displayName, function(a){});
```

- minors

```
JSAPI.get()
    .getActionFactoryAPI()
    .buildMinor(id, displayName, function (a) {});
```

Example: Add a button-type action to the table displaying the contents of a folder

```
JSAPI.get().registerForFolderChildrenLoaded(function(api, phase,
component) {
        var action =
JSAPI.get().getActionFactoryAPI().buildTextual("action", "My
action", function(a);

                console.log("execute action: " + a.getId());
        });
        api.getDocumentChildren().getActions().add(action);
});
```

Icon action

This type of action is based on the FontAwesome font, offering a wide choice of icons through CSS styles.

Example: Add icon action on task attachments

```
var cardAPI = JSAPI.get().getCardAPI();
cardAPI.registerForAttachment(function(card, task, definition,
component){
        var actionAPI = JSAPI.get().getActionFactoryAPI();
        var action = actionAPI.buildIcon("myAction", “My action",
"fa fa-user", function(actionPresenter){
                console.info("on click");
                actionPresenter.setEnabled(false);
    });
    card.getActions().add(action);
});
```

Menu action

This type of action is based on the FontAwesome font, offering a wide choice of icons through CSS styles and a label.

Example: Add a menu action to a form header.

```
var actionSet = formAPI.getActions().getHeaderActions();
var actionAPI = JSAPI.get().getActionFactoryAPI();
var action = actionAPI.buildMenu("myAction", “My menu", "fa fa-
user", function(actionPresenter){
                console.info("on click");
});
actionSet.add(action)
```

Responsive action

This type of action combines both an icon-type action and a menu-type action.

During construction, an integer is supplied to indicate the weight of this action. This weight determines the priority of the action to be displayed directly (as an icon) when enough space is available.

When there isn't enough space to display all actions, those with the greatest weight are displayed as icons first. The others are displayed in menu form.

```
var actionSet = formAPI.getActions().getHeaderActions();
var actionAPI = JSAPI.get().getActionFactoryAPI();
var action = actionAPI.buildResponsive(
    "myAction",
    "My menu",
    "fa fa-user",
    10,
    function (actionPresenter) {
        console.info("on click");
    }
);
actionSet.add(action);
```

You can also change the weight of a native responsive action using the function `setWeight(String actionId, int weight)` in the action container.

```
JSAPI.get().registerForComponentChange( function(formAPI, component,
phase) {

formAPI.getActions().getHeaderActions().registerForAdd(function(actio
{
       if(action.getId() == “attach") {

formAPI.getActions().getHeaderActions().setWeight(action.getId(), 0);
// In the menu

       else if(action.getId() == "replaceContent"){

formAPI.getActions().getHeaderActions().setWeight(action.getId(),
100); // In first position

    });
});
```

Note: *this type of action can only be used for* `headerActions` *of a component indexing form.*

DOM action

This fifth type of action also has the `name` and `label` properties. The last property is a DOM element `element`. It is possible to interact with this DOM element using classic DOM events such as *onclick*, *onmouseover*, ...

Example:

```
        var actionAPI = JSAPI.get().getActionFactoryAPI();

        var element = document.createElement('button');
        element.innerHTML = "Custom button title";
        var action = actionAPI.buildDOM("className", "hover label",
element);

        element.onclick= function(){
                console.log("Click on DOM button");
```

Development / JSAPI / OffMenu / Overview

Different types of OffMenu can be instantiated or modified:

- DOM
- IFrame
- Widget

These menus can be instantiated using the JS API:

```
JSAPI.get().addOffMenu();
```

Development / JSAPI / OffMenu / DOMOffMenu

DOM Content OffMenus are used to add a DOM element to the menu.

This allows you to add a JS element to the menu.

```
var domElement = document.createElement("div");
domElement.innerHTML = "Hello world";
var menu = new DOMOffMenu("MenuTitle", domElement);
JSAPI.get().addOffMenu(menu);
```

Development / JSAPI / OffMenu / IFrameOffMenu

IFrame OffMenus allow you to add an IFrame to the menu.

To instantiate an IFrameOffMenu, you need to provide it with the url of the IFrame to be built:

```
var menu = new IFrameOffMenu("MenuTitle", url);
JSAPI.get().addOffMenu(menu);
```

A callback can be provided to this method to react to the loading of the IFrame:

```
var menu = new IFrameOffMenu("MenuTitle", url, (window) => {
    console.info("The IFrame has been loaded");
});
JSAPI.get().addOffMenu(menu);
```

Development / JSAPI / OffMenu / WidgetOffMenu

Widget OffMenus allow you to view and interact with this widget to the menu. The [attachment metadata viewer plugin](/documentation/apis/plugins/attachment.md) is based on this type of menu.

To instantiate a WidgetOffMenu, you need to provide it with the Widget to be included in the menu:

```
var menu = new IFrameOffMenu("MenuTitle", Widget);
JSAPI.get().addOffMenu(menu);
```

Development / JSAPI / Popups / Overview

Different types of popup can be instantiated or modified:

- Components creation
- Displaying existing component metadata
- Activity
- Search
- DOM content
- Linking a document to a parent folder
- Confirmation of a user choice

These popups can be instantiated using the JS API:

```
JSAPI.get().getPopupAPI();
```

The following functions are available to modify the various common attributes of popups:

Function	Description
setIcon(String iconStyle)	Popup icon definition
setTitle(String title)	Popup Title definition

Function	Description
setDescription(String description)	Popup description definition
addStyle(String style)	Adding a style to the popup
setContent(Element content)	Overwrites existing popup body content
addContent(Element element)	Adding an element to the popup body
clearContent()	Overwrites popup body content
setClosable()	Enables the possibility of closing the popup with the popup cross
addCloseHandler(CloseCallback callback)	Add a callback on popup closure
show()	Popup opening
close()	Popup closing
setAutoCloseOnEnter(boolean autoCloseOnEnter)	Popup closing by pressing Enter

Development / JSAPI / Popups / Confirmation

> **INFO**
>
> This type of pop-up displays a message confirming the user's choice.

There are two types of confirmation popups: Validate/Cancel popups, and Yes/No popups.

Both types of popups work in the same way:

When the user clicks on the `Validate` button (for Validate/Cancel popups) or `Yes` (for Yes/No popups), the integrator has the power to define the treatment to be executed.

Validation/Cancel **Yes/No**

```
var popup =
JSAPI.get().getPopupAPI().buildCancelValidateConfirmationPopup('This
action will be performed', function(confirmed){
        if(confirmed){
                // .....

        popup.close();
});
popup.setDescription('My description');
popup.setTitle("My title");
popup.setIcon('fa fa-remove');
popup.show();
```

> **INFO**
>
> This type of popup allows you to launch the creation of a component by displaying an indexing form in a popup.

To instantiate a creation popup, it is necessary to provide it with an input component (here `newTask`):

```
var newTask = new Task();
var popup =
JSAPI.get().getPopupAPI().buildComponentCreation(newTask);
```

A callback can be provided to this method to react to the creation of the component:

```
var newTask = new Task();
newTask.setClassId("ClaimProcess_Start");
var popup = JSAPI.get()
    .getPopupAPI()
    .buildComponentCreation(newTask, function (saved) {
        console.info("The task has been created: " +
saved.getId());
    });
popup.show();
```

Development / JSAPI / Popups / File selector

In the case of document creation, it is also possible to display a popup without a file selector. This instantiates a blank document with no content.

To do this, use the `buildDocumentCreation()` document-specific function. This function has a parameter `allowFileAttachments`, which allows the file selector to be displayed or not within the popup.

```
var newDocument = new Document();
var popup =
JSAPI.get().getPopupAPI().buildDocumentCreation(newDocument,
false);
```

Development / JSAPI / Popups / Tags

> **INFO**
>
> This type of popup displays the tags of an existing component within a popup.

To instantiate this type of popup, you need to supply the API with an existing component:

```
var popup =
JSAPI.get().getPopupAPI().buildComponentFields(document);
```

Example : Displaying document tags in a popup window

```
JSAPI.get()
    .document()
    .get(
        ["documentId"],
        function (documents) {
            var document = documents[0];
            console.info("Got: " + document.getId());
            var popup =
JSAPI.get().getPopupAPI().buildComponentFields(document);
            popup.setTitle("My document");
            popup.setDescription("View document tags");
            popup.setIcon("fa fa-book");
            popup.show();
        },
        function (error) {
            console.error("Documents could not be get: " + error);
        }
    );
```

INFO

This type of popup opens a search form in a popup.

To open a search form in a popup, the following parameters are required as string:

- `category` : Category to be searched
- `template` : Name of template to be used for search

The `callback` parameter can also be added, but is optional. It allows you to select a component that can, for example, be added as an attachment to a task.

```
popup = JSAPI.get()
    .getPopupAPI()
    .buildSearch(category, template, function (component) {
        console.log("Selected component id=" + component.getId());
    });
```

Development / JSAPI / Popups / Search configuration

Once the popup has been built, you can access or modify the various parameters using the following functions:

Function	Description
getHiddenRequest()	Retrieving the hidden query from the search form
setHiddenRequest(SearchRequest request)	Update of the search form's hidden query
getHiddenColumns()	Retrieving the list of hidden columns from the search form
setHiddenColumns(String[] columns)	Update the list of hidden columns in the search form
setOpenOnCategoryClick(boolean open)	Activate/deactivate component opening when category icon is clicked. *Enabled by default*

Development / JSAPI / Popups / DOM

The DOM Content popup allows you to add an element *DOM* in a popup window. This allows you to add a JS element to a popup.

```
var domElement = document.createElement("div");
domElement.innerHTML = "Hello world";
var popup = JSAPI.get().getPopupAPI().buildDOM(domElement);
popup.show();
```

Development / JSAPI / Popups / Document attachment

This section describes the possible interactions with the widget called `ComponentAttacher` which allows you to attach a component to a folder.

This type of popup cannot be created manually, but it is possible to subscribe to its opening in the following way:

```
JSAPI.get()
    .getPopupAPI()
    .registerForComponentAttacherOpen(function (popup) {
        popup.setTitle("My new title");
        popup.setRapidSearchVisible(false);
        popup.getActions().get("advancedSearch").setEnabled(false);
    });
```

Properties that can be modified on other popup types can also be modified on this one. The following functions are specific to this type of popup:

Property	Description
setRapidSearchVisible(boolean visible)	Quick search visibility
setSearchTemplate(String searchTemplate)	Defining the search form used for advanced search

Property	Description
attach(Folder folder)	Attaches the current component to a folder
getActions()	Allows to retrieve the corresponding ActionSetAPI, to manipulate the various actions present.

The actions available via the `getActions()` method are:

- `browseFolder`: Browse folder tree
- `advancedSearch`: Advanced search
- `newFolder`: Creating a new folder

The popup displayed when the action is clicked `browseFolder` will display a folder tree. It's possible that for some requirements the complete tree structure won't meet your needs, and that a certain part of the tree structure is authorised for an `attachment`. In this case, you can restrict the display of the various folders in this tree structure as follows:

```
folderAttacher = JSAPI.get().getPopupAPI().getFolderAttacher();
folderAttacher.register(function(folder,toAttach, callback){
        if(folder.getName() == "2014"){
        console.error("Remove item " + folder.getName() + " from
browse tree");
                callback.onSuccess(null);

        else if(folder.getName() == "Arondor"){
                console.log("Force disable selection for "+
folder.getName());
                callback.onSuccess(new
SelectableTreeItem(folder.getName(),false));

        else{
                console.log("Use applicative selection for "+
folder.getName());
                callback.onSuccess(new
SelectableTreeItem(folder.getName());

});
```

In the example above, the component with the name *Arondor* will be displayed, but not selectable. The one with the name *2014* will not be displayed and the others will be.

In the latter case, the selection used is the one calculated by the application, i.e it's only displayed if the current component class is in the folder's list of authorised children.

INFO

The purpose of this section is to describe how to override product labels.

The following API is available for this purpose:

```
JSAPI.get().getLabelsAPI();
```

This functionality is also available by overloading part of the labels via property files. Documentation for this part is available here.

Development / JSAPI / Labels modification

This API can be used to modify native labels. To do this, you need to subscribe to the retrieval of labels from the server:

```
var api = JSAPI.get().getLabelsAPI();
api.registerForLoad(function (language, callback) {
    callback.onSuccess();
});
```

Within the subscription, it is then possible to modify the native labels:

```
var api = JSAPI.get().getLabelsAPI();
api.registerForLoad(function (language, callback) {
    api.setLabel("home", "Dashboard");
    callback.onSuccess();
});
```

To retrieve all available labels :

```
var api = JSAPI.get().getLabelsAPI();
api.registerForLoad(function (language, callback) {
    console.info("labels: " + api.getLabelsNames());
    callback.onSuccess();
});
```

Using labels

The following functions are available for modifying or retrieving existing labels:

Function	Description
getLabel(String labelName)	Retrieves the value of a label
getLabelWithParams(String labelName, Object... params)	Retrieves the value of a label with input parameters
getLabelWithPlural(String labelName, int pluralCount, Object... params)	Retrieves the value of a label with input parameters, taking the application of plural or not with the value of the parameter `pluralCount`

INFO

FlowerDocs uses a placeholder mechanism to define the activity (or screen) the user is currently viewing in the URL. The fact that this placeholder is carried by the URL means that users can :

- refresh their browser while remaining in the same context
- navigate through browser history (Previous and Next buttons)

Development / JSAPI / Navigation

To control these functions via the JS API, a navigation API is provided:

```
var navigationJSAPI = JSAPI.get().getNavigationAPI();
```

Function	Description
reload()	Allows you to reload the current space without confirmation
goBack(boolean confirmation)	Returns to previous position - `confirmation`: Boolean to deactivate confirmation at change of place if there are unsaved modifications
goToComponentPlace(String category, String identifiant, boolean confirmation)	Redirects the user to the component modification screen, providing : - `category` : `DOCUMENT`, `FOLDER`, `TASK`, `VIRTUAL_FOLDER` - `identifiant` : identifier of component to open - `confirmation`: Boolean to deactivate confirmation at change of place if there are unsaved modifications

Function	Description
goTo(Place place, boolean confirmation)	Redirects the user to a seat - `place` : identifier of place to open - `confirmation`: Boolean to deactivate confirmation at change of place if there are unsaved modifications

It is possible to redirect the user to a place built from different tokens.
Example : Redirect to a task creation screen `GEC_Step0_Creation`

```
var navigationJSAPI = JSAPI.get().getNavigationAPI();
var placeToken = "storeTask(GEC_Step0_Creation)";
navigationJSAPI.goTo(flower.docs.PlaceBuilder.build(placeToken),
false);
```

The Navigation API also allows you to override the default location (defined at profile level). To do this, you can subscribe to its opening via the `registerForPlaceChange(callback) function`

Example : Override the default placeholder with the search form `pliSearch`

```
var navigationJSAPI = JSAPI.get().getNavigationAPI();
navigationJSAPI.registerForPlaceChange(function (placeToken,
callback) {

callback.go(flower.docs.PlaceBuilder.build("search(pliSearch)"));
});
```

Current place

Thanks to the Navigation API, it is also possible to retrieve the current place and its characteristics using the `getWherePlace()` method.

```
var navigationJSAPI = JSAPI.get().getNavigationAPI();
var place = navigationJSAPI.getWherePlace();
```

This object exposes the following methods:

Function	Description
getWhereId()	To obtain the identifier of the current place
getPlaceType()	Used to obtain the type of the current place: `HomePlace`, `SearchPlace`, `BrowsePlace`, `ModifyVirtualFolderPlace`, `ModifyFolderPlace`, `AdminPlace`
getSelectedLeaf()	Displays the selected aggregation in the virtual folder
getOpenedItems()	Retrieves the tree structure open in the virtual folder
getTemplate()	Retrieves the selected search pattern
getRequest()	Retrieves the query executed in searches and virtual folders (visible in the URL)

Function	Description
getSearchDisplayMode()	Recovers the search display mode : `CARD` or `TABLE`
getVirtualFolderMode()	Recovers the display mode of the virtual folder : `VIEWER` or `AGGREGATION`
isAdminPlace()	Indicates whether the page is on the administration side

A JS API can be used to obtain information about users:

```
JSAPI.get().getUserAPI();
```

Development / JSAPI / Get a user object

Two methods are available to obtain a user object:

- Common user :

```
JSAPI.get().getUserAPI().getCurrentUser();
```

- Other user :

```
JSAPI.get()
    .getUserAPI()
    .getUser("kta", function (user) {
        console.log("user: " + user.getId());
    });
```

Functions	Description
getScope()	Retrieve the scope to which the user is connected
getCurrentUser()	Current user recovery
getUser(String id, UserCallback closure)	Recovering a user by identifier
addAttribute(String name,String[] values)	Add attribute for logged-in user

Functions	Description
removeAttribute(String name)	Delete attribute values for logged-in user

User information

The functions listed below can be called on a user object.

Functions	Description
getId()	User recovery
getDisplayNames()	User label recovery
getFirstName()	User first name recovery
getLastName()	User last name retrieval
getProfiles()	Retrieve the teams to which the user belongs
getGroups()	Retrieve groups to which the user belongs
getMail()	Retrieve user email address
getAttributeValue(String name)	Retrieve the value of a user attribute
getAttributeValues(String name)	Retrieve user attribute values

The token is no longer reassembled when users are retrieved by **FlowerDocs GUI**, to prevent it being used by a malicious individual.

It is still possible to retrieve the user's token when retrieving users via **FlowerDocs GUI** by configuration. To do this, the following property must be

added to the `gui.properties` file: `user.expose-token=true`

Assignee Provider

An `Assignee Provider` provides the GUI with a callback that is executed when a user searches for a user to assign a task to. By default, all users will be remoted.

In certain business situations, it may be necessary to filter these users. To do this, the User API provides the function :

```
var userAPI = JSAPI.get().getUserAPI();
userAPI.registerAssigneeProvider(function (tasks, key, callback)
{});
```

The function parameters are:

Parameter	Description
tasks	List of tasks on which assignment is made
key	User name
callback	callback function executed

2 methods are available to call the callback:

- *callback.provide(users)* - must be called with an object array `User`.
- *callback.na()* - do not filter users.

> **INFO**
>
> In the case of a search-based assignment, the `tasks` parameter is built from the search results. They do not include tags that are not present in the associated SearchResult object, and are therefore dependent on the search form.

A filter type is provided by default through the REST web service `./plugins/rest/profiles/<profiles>/users/`. This web service lets you search for users belonging to one or more teams.

Example: selection of users from the `LEGAL` and `ADMIN` teams:

```
var userAPI = JSAPI.get().getUserAPI();
userAPI.registerAssigneeProvider(function (tasks, key, callback) {
    var profile = "LEGAL,ADMIN";
    $.get("./plugins/rest/profiles/" + profile + "/users/" + key,
function (data) {
        var users = new Array();
        $.each(data, function (k, v) {
            users.push(User.fromJSON(JSON.stringify(v)));
        });
        callback.provide(users);
    });
});
```

Development / JSAPI / Available Objects

Card

The functions available on the cards are as follows:

Function	Description
setTitle(String title)	Modifies card title
setHeading(String heading)	Modifies card description
getActions()	Recovers the card's action container
getStyle()	Retrieves the card's CSS class
addStyle(String style)	Adds the CSS class to the card
setStyle(String style)	Replaces the card's CSS classes with the one provided
asElement()	Retrieves the card as a DOM element

Container cards

Container cards are large cards into which content can be added. The functions available are as follows:

Function	Description
setTitle(String title)	Modifies card title
setCaption(String caption)	Modifies card description
setIcon(String icon)	Modifies card icon
addCard(Card card)	Add a card to this one
setContent(Component content)	Allows to add a DOM element to the card
setVisible(boolean isVisible)	Allows to hide or show the card
setReduced(boolean isReduced)	Allows to reduce the card (only the title is visible)
getStyle()	Retrieves the card's CSS class
addStyle(String style)	Adds the CSS class to the card
setStyle(String style)	Replaces the card's CSS classes with the one provided

Recording

To access the cards, you need to register when you add them. There are two types of cards for which the JS API can register, cards of attachments and search results.

Search result

Subscription to the addition of a search result card is performed using the `cardAPI` object:

```
var cardAPI = JSAPI.get().getCardAPI();
cardAPI.registerForComponent(function(card component){
        ...
```

The function parameters are:

- `card`: the added card
- `component`: the component created from the search result

Example:

```
var cardAPI = JSAPI.get().getCardAPI();
cardAPI.registerForComponent(function(card component){
        var actionAPI = JSAPI.get().getActionFactoryAPI();
        var myElement = document.createElement('button');
        myElement.innerHTML = "Custom button title";
        var action = actionAPI.buildDOM("className", "My action
description", myElement);

        card.getActions().add(action);
        card.setTitle("Title");
        card.setHeading("Heading");

        myElement.onclick= function(){
                console.log(card);
                console.log(component.getClassId());

});
```

Attachment

Loading

For each attachment definition, you can subscribe to the loading of the corresponding card through a subscription mechanism using the `cardAPI` object:

```
var cardAPI = JSAPI.get().getCardAPI();
cardAPI.registerForAttachment(function(card, task, attDefinition,
component){
        ...
```

The function parameters are:

- `card`: the added card
- `task`: the task to which the attachment is attached
- `attDefinition`: the definition of the attachment
- `attComponent`: the component attached to the task

Examples:

```
var cardAPI = JSAPI.get().getCardAPI();

cardAPI.registerForAttachment(function(card, task, definition,
component){
        var actionAPI = JSAPI.get().getActionFactoryAPI();
        var myElement = document.createElement('button');
        myElement.innerHTML = "Custom button title";
        var action = actionAPI.buildDOM("className", "My action
description", myElement);

        card.getActions().add(action);
        card.setTitle("Title");
        card.setHeading("Heading");

        myElement.onclick= function(){
                console.log(card);
                console.log(task.getClassId());
                console.log(definition.getClassId());
                console.log(component);

});
```

To programmatically add an attachment to a task, you can trigger an event such as:

```
var doc = new Document();
doc.setName("Attached document");
doc.setClassId("Claim");

var event = new AddTaskAttachmentEvent("Attachment2", doc);
var formAPI = JSAPI.get().getComponentFormAPI(<identifiant du
composant>);
formAPI.fireEvent(event);
```

In order to display the previously programmatically added attachment in the viewer, it must be notified as follows:

```
formAPI.fireEvent(new
AttachmentsLoadedEvent(formApi.getComponent()));
```

Changes

To react to changes made by users to attachments, you can subscribe to changes on the `card` object:

```
card.registerForChange(function (attachment) {
    console.info("Detected attachment change: " + attachment);
});
```

Actions

Cards with attachments of type `DOCUMENT` have a set of native actions.

These actions can be managed programmatically using their identifier:

- `upload-version`: uploading a new version of a document
- `delete-version`: deleting the uploaded version
- `upload-attached`: uploading a document to be created
- `delete-attached`: deleting existing document added

A lookup involves retrieving data from a third-party or other repository, and building dynamic lists of values. This mechanism enhances the user experience and enriches the data stored or retrieved through FlowerDocs.

This functionality is based on the notion of plug-ins, which can be used to execute different types of calls/queries (database, web services, etc.).

Development / JSAPI / Lookup plugins

Example :

A lookup plugin can be defined in JavaScript using the API provided. To do this, you need to instantiate a `LookupPlugin` object, as shown in the example below. A plugin of this type has a `LookupHandler` called when the lookup is executed to provide FlowerDocs with a list of matching results.

```
var lookupPlugin = new LookupPlugin();
lookupPlugin.setLookupHandler(function (fieldName, fieldValue,
callback) {
    var results = new Array();
    callback.onSuccess(results);
});
```

Each plugin must then be made available to the application. The snipet below shows how to register the plugin where `lookupId` corresponds to the lookup name.

```
var lookupAPI = JSAPI.get().getLookupAPI();
lookupAPI.register("lookupId", lookupPlugin);
```

Using a lookup

Lookups stored in the application can be used in a variety of ways, depending on your needs.

Dynamic call

Here's the code for executing a lookup:

Example : Executing a lookup `SampleLookup`

```
JSAPI.get()
    .getLookupAPI()
    .lookup("SampleLookup", "BusinessReference", "ref123", function
(results) {
        // Using lookup results
    });
```

The `results` object is an array of `LookupResult` objects exposing the methods:

Functions	Description
getKey()	Retrieves the key of a result (i.e. symbolic name)
getValue()	Retrieves the value or label of a result

In this example, we execute the `LookupPlugin` whose identifier is `SampleLookup` with the tag name and value as parameters. By reacting to the modification of its value, it is then possible to modify the list of proposed values according to the user's input.

In these examples, the `SampleLookup` lookup is used to suggest values for the `Amount` tag, (either as a suggestion or as a closed list).

Example : Suggested value for a tag

```
formAPI.registerForFieldChange(“Amount", function(fieldName,
fieldValue) {
        JSAPI.get().getLookupAPI().lookup(“SampleLookup”,”Amount",
fieldValue, function(results) {
                formAPI.suggest("Amount", results);
        });
});
```

Example: Definition of allowed values in a list

```
formAPI.registerForFieldChange(“Amount", function(fieldName,
fieldValue) {
        JSAPI.get().getLookupAPI().lookup(“SampleLookup”,”Amount",
fieldValue, function(results) {
                var restrictedAllowedValues = new Array();
                for ( var i in results) {
                        result = results[i];
                        var allowedValue =
buildAllowedValue(result.getName(), result.getValue());
                        restrictedAllowedValues[i] = allowedValue;

                formAPI.setAllowedValues("EDS",
restrictedAllowedValues);
        });
});

function buildAllowedValue(symbolicName, label) {
        var language = new Language("EN");

        var allowedValue = new AllowedValueDefinition();
        allowedValue.setSymbolicName(symbolicName);

        var displayNames = new I18NLabel()
        displayNames.setLabel(language, label);
        allowedValue.setDisplayNames(displayNames);
        return allowedValue;
```

Link with a tag

This JS API also provides the ability to automatically link a tag from the return of a `LookupPlugin`. This can be done using the `function bindFieldOnLookup(String tagName, String lookupId)` where :

- `tagName` is the identifier of the tag concerned
- `lookupId` is the identifier of the `LookupPlugin` to be used.

Example : Binding a tag on the lookup `userLookup`

```
formAPI.bindFieldOnLookup("DestinataairePourInfo", "userLookup");
```

Development / JSAPI / Folder content

The management of creation actions within a folder can be configured by registering a processor. This processor determines whether or not creative actions are activated.

```
JSAPI.get().folder().registerForCreateAction(identifier,
processor);
```

The `identify` variable is used to identify the folder (or context) in which the native management of creation actions should be overloaded. This character string can take on the following values:

- a folder identifier
- a folder class identifier
- `'*'` for all folders

The `processor` variable is a closure used to define the desired behavior: a creation action is visible or not. The following variables are provided as input to this closure:

- folder: the folder concerned by the creation actions
- childCategory: the components category that can be created using the action
- action: the creative action
- callback: the callback indicates to the graphical user interface the configuration to be applied (enables asynchronous processing)

For example, to hide the document creation action in the `Projects` folder:

```
JSAPI.get().folder().registerForCreateAction("Folder",function(folder
action, callback){
    if(folder.getName() =='Projects'&& childCategory == 'DOCUMENT'){
                callback.onSuccess(false);
        }else{
        callback.onSuccess(true);

});
```

> **INFO**
>
> The history allows to display the modifications made to a component since its creation. This history can be manipulated to add, modify or delete facts.

A fact allows you to trace an action performed on a component and thus provide a component history.

Development / JSAPI / Factual information

The facts are made up of several pieces of information that allow us to identify the precise action performed and its context.

Object identification

The object on which a fact took place can be identified from the object type and its identifier.

Functions	Description
getObjectType()	Object type retrieval
setObjectType(String type)	Modification object type
getObjectId()	Object identifier retrieval
setObjectId(String id)	Object identifier modification

Context

In order to provide information on the context in which a fact took place, several types of information are available:

Functions	Description
getUser()	User recovery
setUser(String user)	User Modification
getAction()	Action performed recovery
setAction(String action)	Action performed modification
getDescription()	Description retrieval
setDescription(String description)	Description modification
getCreationDate()	Action completion date recovery
setCreationDate(Date date)	Action completion date modification

The actions (see `com.flower.docs.domain.fact.Action`) supported for facts are:

- `CREATE`: Component creation
- `UPDATE`: Component update
- `VERSION`: Creating a component version
- `ASSIGN`: Assignment of a task to a user
- `ADD_CONTENT`: Adding an attachment to tasks, adding or updating content for documents and adding components to a folder
- `DELETE_CONTENT`: Deleting an attachment to tasks, deleting content for documents and deleting components to a folder
- `REVERT`: Restores a version of a document

Additional fields

In addition to information natively stored in the facts, additional fields can be stored (or simply displayed) using `com.flower.docs.domain.common.ResultField`.

Functions	Description
getUpdatedFields()	Fact fields recovery
setUpdatedFields(ResultField[] fields)	Fact fields modification
addUpdatedField(ResultField fields)	Add a field to the fact
addUpdatedFields(ResultField[] fields)	Add several fields to the fact

```
var fact = new Fact();
var field = new ResultField();
field.setName("RefClient");
field.setValue("1234");
fact.addUpdatedField(field);
```

Add facts

In certain business contexts, it may be necessary to build your own facts. To do this, several functions are available for this purpose:

To add new facts, it is first necessary to obtain a fact instance with one of the following constructors:

Functions	Description
new Fact(Component component)	Creating a fact from a component
new Fact()	Creating an empty fact

To modify a component's history, you need to subscribe to the history opening feature:

```
var factAPI = JSAPI.get().getHelperFactory().getFactAPI();
factAPI.register(component.getId(), function (facts, callback) {
    //Manipulate facts of component
});
```

The `facts` array provided as input to the subscription contains the facts determined natively by the application. In order to modify the facts displayed to users, it is necessary to provide a modified array to the `callback` provided:

```
var factAPI = JSAPI.get().getHelperFactory().getFactAPI();
factAPI.register(component.getId(), function (facts, callback) {
    var processed = new Array();
    callback.onProcessed(processed);
});
```

EXAMPLE: ADDING TWO FACTS TO A COMPONENT'S HISTORY

```
JSAPI.get().registerForComponentChange(function
(componentFormAPI, component, phase) {
    var factAPI = JSAPI.get().getHelperFactory().getFactAPI();
    factAPI.register(component.getId(), function (facts,
callback) {
        var fact1 = new Fact(component);
        var date1 = new Date("January 15, 2017 11:13:00");
        fact1.setUser(JSAPI.get().getUserAPI().getUserId());
        fact1.setCreationDate(date1.getTime());
        fact1.setAction("UPDATE");
        var field1 = new ResultField();
        field1.setName("RefClient");
        field1.setValue("2018-01-01 12:01:07.006 +0100");
        fact1.addUpdatedField(field1);

        var fact2 = new Fact(component);
        var date2 = new Date("January 16, 2017 11:13:00");
        fact2.setCreationDate(date2.getTime());
        fact2.setDescription("Description for fact2");

        facts.push(fact1);
        facts.push(fact2);
        callback.onProcessed(facts);
    });
});
```

INFO

Configuring the fields to be historised with a class document `FactFieldsConfiguration`.

Development / JSAPI / Versions

A naming strategy defines the labels that users can enter when creating a document version. Different strategies are available:

- `MINOR`: MINOR version only.
- `MAJOR` : MAJOR version only.
- `NUMBERED`: MINOR and MAJOR versions.
- `CUSTOM` : Custom label only.
- `ALL` : Three strategies are offered to the user: MINOR, MAJOR and CUSTOM.
- `NONE` : No strategy, no promotional action displayed.

By default, the `ALL` strategy is applied. The Javascript API can be used to restrict the naming strategies offered to users:

```
var versioningAPI = JSAPI.get().getVersioningAPI();
versioningAPI.register(function (component, callback) {
    callback.onSuccess("CUSTOM");
});
```

Several resolvers can be defined, but the first valid value is used as the naming strategy.

INFO

Version numbers are automatically suggested from the previous version label.

- So if the previous version is 1.0, then for a minor version 1.1 will be proposed, and 2.0 for a major one.
- If for the previous version, the label is customized (noted: xxx) then the proposed labels will be xxx**_0.1** for a minor version and xxx**_1.0** for a major version.

INFO

Each component category has its own icon in **FlowerDocs GUI**. For tasks, the icon can be defined by class, to provide a different icon for each user step.

Development / JSAPI / Other / Icon Resolver

In certain situations, it may be necessary to define an icon based on a component's tags. To do this, you can define an *Icon Resolver* to define the icon to be applied to a component. *multiple Icon Resolvers can be defined, in which case the first one to respond with a value will be taken into account.*

Icon Resolvers are used to resolve a component's icon when it is displayed in **FlowerDocs GUI**.

Recording

An *Icon Resolver* can be registered using the function `registerComponentIconResolver(resolver,tags)` exposed by `JSAPI.get().getHelperFactory()`. The `resolver` parameter corresponds to a function accepting two parameters:

- `component`: the component whose icon is to be defined
- `callback`: object whose `onSuccess()` function must be called once the icon calculation has been completed (in the case where the icon is not defined by the *Icon Resolver*, the `onSuccess()` function must be called with no value)

The `tags` parameter corresponds to the list of tags according to which the icon will be resolved.

Example: To define an icon of type *loupe* for all components of class `IncomingMail`, it is possible to register an *Icon Resolver* such as :

```
JSAPI.get().getHelperFactory().registerComponentIconResolver(function
callback){
  if(component.getClassId() == 'IncomingMail'){
    callback.onSuccess('fa fa-search');
  }else{
    callback.onSuccess(null);

});
```

Example: To define a *loupe* type icon for all components with the tag value `TypeCourrier` equal to `Order` and the `Status` equal to `TODO`, it is possible to register a *Icon Resolver* such as :

```
JSAPI.get().getHelperFactory().registerComponentIconResolver(function
(component, callback) {
        if (component.getTagValue('MailType') == 'Order' &&
component.getTagValue('Status') == ‘TODO') {
                callback.onSuccess('fa fa-search');
        } else {
                callback.onSuccess(null);
        }}, 'MailType', 'Status');
```

Development / JSAPI / Other / Notifications

To send notifications, the Notification API can be accessed using the `JSAPI.get().getNotificationAPI()` function. It features the following functions:

- `sendInformation(String message)`
- `sendWarning(String message)`
- `sendError(String message)`

These three functions open modal windows.

To send a notification to FlowerDocs, you can use the `sendNotification(String message)` function.

Example:

```
JSAPI.get().getNotificationAPI().sendError("An error has
occurred");
```

Development / JSAPI / Other / Loading panel

To notify the user of ongoing processing during JavaScript execution, the loading panel can be displayed and then hidden at the end of processing with the following functions available from the `FlowerJSAPI.get()` API:

Function	Description
showLoadingPanel()	Displays the loading panel
hideLoadingPanel()	Hide loading panel

_example of loading panel display for 3 seconds__

```
JSAPI.get().showLoadingPanel();
setTimeout(function () {
    JSAPI.get().hideLoadingPanel();
}, 3000);
```

Development / JSAPI / Other / Permissions

It is possible to determine whether the current user has got a permission:

- from a component

```
ACLHelper.isGranted(component, "CREATE", function (granted) {
    console.info("granted: " + granted);
});
```

- from an ACL identifier

```
ACLHelper.isGranted(component.getACL(), "CREATE", function
(granted) {
    console.info("granted: " + granted);
});
```

Identifier extraction

To extract identifiers from an array of components, you can use the following function:

```
Ids.from(components);
```

Tag Propagation

To propagate common tag values from one or more components to a target component, the following functions are available:

Functions	Description
predict(sourceComponent, targetComponent)	Propagate common tags from a component to a target component
predict(sourceComponents, targetComponent)	Propagate common tags from a set of components to a target component

Example: Creating a task from two documents

```
ContextualMenuAPI.get().registerForLoad(function(api){
        if(api.getSelected().length != 2 || api.getCategory() !=
"DOCUMENT"){
                return;

        var icon = new Icon();
        icon.setContent("fa fa-user flat-mauve");
        api.add("createTask", icon.asElement() , "Task", function()
{
                var newTask = new Task();
                newTask.setClassId("GEC_Step0_Creation");
                TagOracle.predict(api.getSelected(), newTask);
                newTask.addAttachments("Courrier",
Ids.from(api.getSelected()), "DOCUMENT");
                var popup =
JSAPI.get().getPopupAPI().buildComponentCreation(newTask);
                popup.show();
        });
});
```

Version comparison

The `compareVersion('x.y.z')` function compares the input version with the deployed version:

Results	Description
`-1`	The deployed version is older than the supplied version *
`0`	The deployed version is identical to the supplied version *
`1`	The deployed version is newer than the supplied version *

**comparison does not take the patch number into account*

Development / Core APIs / Overview

FlowerDocs Core exhibits a set of services to facilitate integration with third-party applications. These services can be consumed in various ways, which are described in the following sections.

The javadoc is available online:

Core API

The CacheService service lets you perform getAll, clear and clearAll operations on your scope's caches.

- `getAll` allows you to retrieve all caches in the scope.
- `clearAll` purges all your scope's caches.
- `clear` allows you to purge a list of caches defined by their name.

Development / Core APIs / Examples of use / Cache retrieval

The following examples show how to retrieve the list of all FlowerDocs scope caches.

GET ALL:

Cache purge

The examples below show how to purge FlowerDocs scope caches using the various operations of clear.

CLEAR ALL:

REST JAVA

```
DELETE {{core}}/rest/caches HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json
```

CLEAR:

REST JAVA

```
DELETE {{core}}/rest/caches/{names} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
names: name of caches to be purged

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `authentication` service generates a user token for a given scope.

Development / Core APIs / Examples of use / Example

The example below shows how to generate a user token.

Authentication - REST **Authentication - Java**

```
POST {{core}}/rest/authentication HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
Content-Type: application/json

-- Body (json) --

    "password": "pwd",
    "scope”: "TEST",
    "user": "user"
```

Search queries enable you to search for components stored in FlowerDocs according to various criteria. These search queries are composed as follows:

Development / Core APIs / Examples of use / The different clauses

Select clause

Use `selectClause` to define the fields to be brought up. It consists of a list of `fields` values.

Clauses Filter

Use `filterClauses` to define filters to be applied to the search. A `filterClause` is composed as follows:

- `criteria`: filter criteria
- `filterClauses` : les sous-filters

A Filter clause can contain one or more other clauses, enabling you to perform complex queries with logical AND and OR operators. There are two types of Filter clause for this purpose:

- `AndClause` : Clause **AND**, a logical AND operator is applied between its criteria and sub-clauses
- `OrClause` : Clause **OR**, a logical OR operator is applied between its criteria and sub-clauses

Clauses Order

The `orderClauses` allow you to define the order in which results will be returned. They are composed as follows:

- `name`: the name of the criterion on which to sort
- `type`: type of criterion
- `ascending`: sorting in ascending or descending order

Pagination of results

- `start` : Defines the start of the search page
- `max` : Defines the maximum number of results to be returned

Searches return a maximum of 10,000 results; an error occurs when paging after this limit.

Examples

The examples below show how to search for a document with a name containing the `invoice` string.

REST JAVA

```
POST {{core}/rest/documents/search HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}
Content-Type: application/json

--Body (json) --

    "filterClauses": [

            "criteria": [

                    "name": "name",
                    "operator": "CONTAINS",
                    "type": "STRING",
                    "values": [
                        "invoice"
                    ]

            ]

    ],
    "max": 10
```

> **INFO**
>
> In search criteria, the `CONTAINS` operator is not case-sensitive (does not distinguish between upper and lower case), unlike the `EQUALS_TO` and `DIFFERENT` operators. So the above example will retrieve documents whose name contains: "invoice", but also “INVOICE", "Invoice" ...

These examples need to be adapted for each component category.
The `Reservation` service exhibits all the operations available around various components reservation.

Development / Core APIs / Examples of use / Component reservation

Component release

Reservation recovery

The `Encryption` service displays the encryption operation of `character strings`.

Development / Core APIs / Examples of use / Data Encryption

The examples below show how to secure a character string using the `post` operation.

REST JAVA

```
POST {{core}}/rest/encrypt/ HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
message: character string to be encrypted, enclosed in double
quotes

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `Document` service exhibits all the operations available around `DOCUMENT` type components.

Development / Core APIs / Examples of use / Document recovery

The examples below show how to retrieve documents using the various operations of `get`.

Document recovery

REST JAVA

```
GET {{core}}/rest/documents/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: identifier of the documents to be retrieved

-- Headers --
token: {{token}}
Content-Type: application/json
```

Version recovery

This service allows you to retrieve a specific version of a document:

REST JAVA

```
GET {{core}}/rest/documents/{documentId}/versions/{versionId}
HTTP/1.1

-- Paramètres d'URL
core: FlowerDocs Core host
documentId: identifier of the document to be retrieved
versionId: identifier of the version to be retrieved

-- Headers --
token: {{token}}
Content-Type: application/json
```

Retrieving associated files

This service retrieves the files associated with the document whose identifier is passed as input:

- the content: `includeContent` = true
- files: `includeContent` = false

REST JAVA

```
GET {{core}}/rest/documents/{id}/files?includeContent=
{includeContent} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
id: document identifier
includeContent: true or false for content retrieval

-- Headers --
token: {{token}}
includeContent: {{inclut le contenu}}
Content-Type: application/json
```

Document creation

The examples below show how to create documents using the following operation.

REST JAVA

```
POST {{core}}/rest/documents/ HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

        "data": {
            "classId": "Document"
        },
        "category": "DOCUMENT”,
        "name": "D1"

]
```

Document creation with a content

The examples below show how to create a document with its content using the following operation.

Document modification

The examples below show how to update documents.

Document modification with content replacement

This operation allows to modify the data of a document (class identifier, document name, ACL, etc.) as well as modifying its content in the same call.

Data modification

This operation updates a document's tags and data (class identifier, document name, ACL, etc.) but also its content.

> **INFO**
>
> This service operates on a cancel and replace basis, so all the contents and tag values must be supplied by the service at the time of update. It is therefore advisable to retrieve the document, make the changes and call the update service.

REST JAVA

```
POST {{core}}/rest/documents/{id} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
id: document identifier

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

        "files": [

                        "id": "98c1f765-7595-46c3-8f4a-
b75bd7c25ff7",
                        "size": 0

            ],
        "data": {
            "classId": "Document"
        },
        "category": "DOCUMENT”,
        "name": "D2"

]
```

Add file

This operation adds content to a document

- `replace`: replaces the existing file with the new content

REST JAVA

```
POST {{core}}/rest/documents/{id}/files?replace={replace} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
replace: true or false to replace content

-- Headers --
token: {{token}}
Content-Type: multipart/form-data

-- Body (form-data) --
file (type file): file to import
```

Rename file

This operation allows you to rename a file associated with a document:

REST

```
POST /core/rest/documents/{id}/files/{fileId}/name HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
id: document identifier
fileId : file identifier

-- Headers --
token: {{token}}

-- Body (form-data) --
New file name
```

Search document

The search operations all work on the same model as described here.

Document deletion

The examples below show how to delete documents.

Document deletion

This operation allows to delete the document and its associated files.

REST JAVA

```
DELETE {{core}}/rest/documents/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: identifiers of documents to be deleted

-- Header --
token: {{token}}
Content-Type: application/json
```

File deletion

This operation allows you to delete a file.

REST JAVA

```
DELETE /rest/documents/{documentId}/files/{fileId} HTTP/1.1

-- URL parameters --
Host: {{core}}
documentId: document identifier
fileId: content identifier to be deleted

-- Headers --
token: {{token}}
Content-Type: application/json
```

Content

Content recovery

This service retrieves the content associated with the file whose identifier is passed as input:

- with or without obfuscations, depending on the parameter `includeObfuscations`

REST JAVA

```
GET {{core}}/rest/documents/{documentId}/files/{fileId}/content?
includeObfuscations={includeObfuscations} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
documentId: document identifier
fileId: content identifier
includeObfuscations: true or false to include obfuscations

-- Headers --
token: {{token}}
Content-Type: application/json
```

Index document content

This service indexes the content passed in parameter and associated with the file identifier.

REST JAVA

```
POST
{{core}}/rest/documents/{documentId}/files/{fileId}/content/index
HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
documentId: document identifier
fileId: content identifier

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (text) --
document content
```

Remove content indexing from a document

This service removes the indexing of content associated with a document.

REST JAVA

```
DELETE
{{core}}/rest/documents/{documentId}/files/{fileId}/content/index
HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
documentId: document identifier
fileId: content identifier

-- Headers --
token: {{token}}
Content-Type: application/json
```

This service removes indexing from all files associated with the document.
The `Annotation` service exhibits all the operations available around a document annotations.

Development / Core APIs / Examples of use / Annotations recovery

The examples below show how to retrieve annotations using the various operations of `get`.

All annotations recovery

REST JAVA

```
GET {{core}}/rest/documents/{documentId}/annotations HTTP/1.1

-- URL parameters --
core: FlowerDocs core host
documentId: identifier of the document where are the annotations to
retrieve

-- Headers --
token: {{token}}
Content-Type: application/json
```

Annotation creation

From a json

REST JAVA

```
POST {{core}}/rest/documents/{documentId}/annotations HTTP/1.1

-- URL parameters --
core: FlowerDocs core host
documentId: identifier of the document on which to create
annotations

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

                "color": {
                        "b": 0,
                        "g": 0,
                        "r": 0
                },
                "creationdate": "2025-08-25T12:21:18.497Z",
                "date": "2025-08-25T12:21:18.497Z",
                "document-id": {
                        "id": "documentId"
                },
                "flags": {
                        "hidden": false,
                        "invisible": false,
                        "locked": false,
                        "norotate": false,
                        "noview": false,
                        "nozoom": false,
                        "obfuscate": false,
                        "print": false,
                        "readonly": false,
                        "togglenoview": false
                },
                "last-modifier": "string",
```

```
                "name": {
                        "id": "string"
                },
                "opacity": 0,
                "page": 0,
                "rect": {
                        "h": 0,
                        "w": 0,
                        "x": 0,
                        "y": 0
                },
                "type":
"com.arondor.viewer.annotation.api.CircleElemType",
                "title": "annotationTest"

]
```

From xml

Annotation modification

JAVA

```
@Autowired
private AnnotationService service;

public void update() throws FunctionalException, TechnicalException

        Id documentId = new Id("documentId");
        List<Annotation> updates = new ArrayList();

        List<Annotation> fetchedAnnotations =
annotationService.get(created.getId());
        Annotation annotToUpdate = fetchedAnnotations.get(0);
        annotToUpdate.setColor(new Color(21, 9, 98));
        updates.add(annotToUpdate):

        annotationService.update(documentId, updates);
```

Annotation deletion

The examples below show how to delete annotations.

Part of annotations deletion

This operation allows to delete some of the annotations of a document

REST JAVA

```
DELETE
{{core}}/rest/documents/{documentId}/annotations/{annotationIds}
HTTP/1.1

-- URL parameters --
core: FlowerDocs core host
documentId: identifier of the document where are the annotations to
delete
annotationIds: identifiers of annotations to delete

-- Header --
token: {{token}}
Content-Type: application/json
```

All annotations deletion

This operation allows to delete all the annotations of a document.

REST JAVA

```
DELETE
{{core}}/rest/documents/{documentId}/annotations/allAnnotations
HTTP/1.1

-- URL parameters --
core: FlowerDocs core host
documentId: identifier of the document where are the annotations to
delete

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `VersionService` service displays the following operations:

- `promote`: to create a version of a document
- `getVersions`: to retrieve document versions
- `revert`: to restore a version of a document
- `deleteVersion`: to delete a version of a document
- `deleteVersions`: to delete all versions of a document

Development / Core APIs / Examples of use / Creating a version

The example below shows how to create a version of a document.

REST JAVA

```
POST {{core}}/rest/documents/{documentId}/versions HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
documentId: document identifier

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body(raw) --
label: version name
```

Versions recovery

The example below shows how to recover versions of a document.

REST JAVA

```
GET {{core}}/rest/documents/{documentId}/versions HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
documentId: document identifier

-- Headers --
token: {{token}}
Content-Type: application/json
```

Restoring a version

The example below shows how to restore a version of a document.

REST JAVA

```
POST
{{core}}/rest/documents/{documentId}/versions/{versionId}/revert
HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
documentId: document identifier
versionId: document version identifier

-- Headers --
token: {{token}}
Content-Type: application/json
```

Version deletion

Deleting a version

The example below shows how to delete a version of a document.

REST JAVA

```
DELETE {{core}}/rest/documents/{documentId}/versions/{versionId}
HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
documentId: document identifier
versionId: document version identifier

-- Headers --
token: {{token}}
Content-Type: application/json
```

All versions deletion

The example below shows how to delete all versions of a document.

REST JAVA

```
DELETE {{core}}/rest/documents/{documentId}/versions HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
documentId: document identifier

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `TempFileService` service exposes the following operations:

- `create`: to create a temporary file
- `getContent`: to retrieve a temporary file
- `delete`: to delete a temporary file

Development / Core APIs / Examples of use / Creating a temporary file

The following examples show how to create a temporary file.

CREATE:

REST JAVA

```
POST {{core}}/rest/files/tmp HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
file: temporary file to create

-- Headers --
token: {{token}}
Content-Type: application/json
```

Recovering a temporary file

The following examples show how to recover a temporary file.

GET CONTENT:

REST JAVA

```
GET {{core}}/rest/files/tmp/{id} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
id: id of the temporary file

-- Headers --
token: {{token}}
Content-Type: application/json
```

Deleting a temporary file

The following examples show how to delete a temporary file.

DELETE :

REST JAVA

```
DELETE {{core}}/rest/files/tmp/{id} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
id: id of the temporary file

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `Task` service exposes all the operations available around `TASK` type components.

Development / Core APIs / Examples of use / Task recovery

The examples below show how to retrieve tasks from an ID list.

REST Java

```
GET {{core}}/rest/tasks/{ids} HTTP/1.1

-- URL parameters --
core : host
ids: identifier of the documents to be retrieved

-- Headers --
token: {{token}}
Content-Type: application/json
```

Task creation

The examples below show how to create a to-do list using the following operation.

REST Java

```
GET {{core}}/rest/tasks/ HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

        "workflow": "Inform",
    "assignee": "fadmin",
    "category": "TASK",
    "data": {
                "classId": "GEC_Step3_CourrierLu",
                "owner": "clm",
                "ACL": "ACL_TASK"
    },
        "name": "3-Courrier lu",
    "tags": [

                        "value": [
                                "Jean"
                        ],
                        "name": "PrenomClient",
                        "readOnly": false
                },

                        "value": [
                                "123456"
                        ],
                        "name": "RefClient",
                        "readOnly": false
                },
```

```
                        "value": [
                                "DUPONT"
                        ],
                        "name": "NomClient",
                        "readOnly": false

    ],
    "attachments": [

                        "componentIds": [
                                "c46de78c-92d4-45ef-b262-
8395aa76a4a8"
                        ],
                        "id": "Courrier",
                        "category": "DOCUMENT",
                        "order": 0

        ]

]
```

Task modification

This operation updates a task's tags and data (class identifier, task name, ACL, etc.).

> **INFO**
>
> This service operates on a cancel and replace basis, so all tag values must be supplied by the service at the time of update. It is therefore advisable to recover the job, make the changes and call the update service.

REST Java

```
GET {{core}}/rest/tasks/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: identifiers of tasks to be updated

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

        "workflow": "Inform",
    "assignee": "fadmin",
    "category": "TASK",
    "data": {
                "classId": "GEC_Step3_CourrierLu",
                "owner": "clm",
                "ACL": "ACL_TASK"
    },
    "name": "3-Courrier lu",
    "tags": [

                        "value": [
                                "Jean"
                        ],
                        "name": "PrenomClient",
                        "readOnly": false
                },

                        "value": [
                                "123456"
                        ],
                        "name": "RefClient",
                        "readOnly": false
                },
```

```
                        "value": [
                                "DUPONT"
                        ],
                        "name": "NomClient",
                        "readOnly": false

    ],
    "attachments": [

                        "componentIds": [
                                "c46de78c-92d4-45ef-b262-
8395aa76a4a8"
                        ],
                        "id": "Courrier",
                        "category": "DOCUMENT",
                        "order": 0

        ]

]
```

Job search

The search operations all work on the same model as described here.

Task deletion

The examples below show how to delete a to-do list from an ID list.

REST Java

```
DELETE {{core}}/rest/tasks/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json
```

Application of an answer

Simple answer

The examples below show how to apply a simple answer to a list of tasks from a list of identifiers.

REST Java

```
PUT {{core}}/rest/tasks/{ids}/answer HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: task identifiers on which to apply the response

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --

  "id": "Decline"
```

Answer with reason

The examples below show how to retrieve tasks from an ID list.

REST Java

```
PUT {{core}}/rest/tasks/{ids}/answer HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: task identifiers on which to apply the response

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --

  "id": "Decline",
   "type": "com.flower.docs.domain.taskclass.ReasonedAnswer",
   "tags": [

        "name": "Remark"
        "value": [
          "The receipt is not up to date."
        ]

    ]
```

Task assignment

The examples below show how to assign a task list to a user from an ID list.

REST **Java**

```
PUT {{core}}/rest/tasks/{ids}/assignee/{username} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: task identifiers on which to apply the response
username: assigned user identifier

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `TaskClassService` service exposes all operations available around `TaskClass` type components.

Development / Core APIs / Examples of use / Retrieving task classes

Retrieve all task classes

The examples below show how to retrieve all task classes.

REST Java

```
GET {{core}}/rest/taskclass HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
```

Retrieve a defined list of task classes

The examples below show how to retrieve a list of task classes from their identifiers.

REST Java

```
GET {{core}}/rest/taskclass/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: list of task class identifiers

-- Headers --
token: {{token}}
```

Creating task classes

The examples below show how to create a task class.

REST Java

```
POST {{core}}/rest/taskclass HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

    "answers": [

        "displayNames": [

            "language": "FR",
            "value": "Traiter"

        ],
        "id": "Treat"

    ],
    "autoAssign": true,
    "category": "TASK",
    "data": {
      "ACL": "acl-treatment",
      "creationDate": "2024-03-07 13:59:54.854 +0100",
      "lastUpdateDate": "2024-03-07 13:59:54.854 +0100",
      "owner": "fadmin"
    },
    "displayNames": [

        "language": "FR",
        "value": "Traitement du courrier"

    ],
```

```
    "icon": "fa fa-clone",
    "id": "GEC_traitement",
    "tagReferences": [

        "descriptions": [

            "language": "FR",
            "value": "Référence du client"

        ],
        "mandatory": true,
        "multivalued": false,
        "order": 0,
        "readonly": false,
        "tagName": "NumReference",
        "technical": false

    ],
    "technical": true,
    "workflow": "GEC"

]
```

Modifying task classes

The examples below show how to modify a task class.

REST Java

```
POST {{core}}/rest/taskclass/{ids} HTTP/1.1

-- URL Parameters --
core: FlowerDocs Core host
ids : comma-separated list of task class identifiers to be updated

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

    "answers": [

        "displayNames": [

            "language": "FR",
            "value": "Traiter"

        ],
        "id": "Treat"
      },

        "displayNames": [

            "language": "FR",
            "value": "Refuser"

        ],
        "id": "Refuse"

    ],
    "autoAssign": false,
    "category": "TASK",
    "data": {
      "ACL": "acl-treatment",
```

```
      "creationDate": "2024-03-07 13:59:54.854 +0100",
      "lastUpdateDate": "2024-03-07 13:59:54.854 +0100",
      "owner": "fadmin"
    },
    "displayNames": [

        "language": "FR",
        "value": "Traitement du courrier"

    ],
    "icon": "fa fa-envelope",
    "id": "GEC_traitement",
    "tagReferences": [

        "descriptions": [

            "language": "FR",
            "value": "Référence du client"

        ],
        "mandatory": true,
        "multivalued": false,
        "order": 0,
        "readonly": false,
        "tagName": "NumReference",
        "technical": false

    ],
    "technical": true,
    "workflow": "GEC"

]
```

If you use the REST service, any information that has not been filled in will be emptied: you need to send the entire task class, not just the information to be modified.

Deleting task classes

The examples below show how to delete a list of task classes.

REST Java

```
DELETE {{core}}/rest/taskclass/{ids} HTTP/1.1

-- URL Parameters --
core: FlowerDocs Core host
ids : comma-separated list of task class identifiers to be deleted

-- Headers --
token: {{token}}
```

Before deleting a task class, it's important to make sure you've deleted all instances of this class, as well as all references to it in workflow objects. The `TagClassService` service exposes all the operations available around `TagClass` type components.

Development / Core APIs / Examples of use / Retrieving tag classes

The examples below show how to retrieve all tag classes.

REST Java

```
GET {{core}}/rest/tagclass HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
```

Tag class creation

The examples below show how to create a tag class.

REST Java

```
POST {{core}}/rest/tagclass HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

        "data": {
            "owner": "user1",
            "creationDate": "2023-10-04 11:00:00.000 +0200"
        },
        "type": "STRING",
        "displayNames": [

                "value": "TagCreate",
                "language": "EN"
            },

                "value": "TagCreation",
                "language": "FR"

        ],
        "searchable": false,
        "id": "TagCreation"

]
```

Modifying tag classes

This operation updates the data of a tag class

REST Java

```
POST {{core}}/rest/tagclass/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: comma-separated list of virtual folder class identifiers to be
updated

-- Headers --

token: {{token}}
Content-Type: application/json

-- Body (json) --
[

        "data": {
            "owner": "user1",
            "creationDate": "2023-10-04 11:00:00.000 +0200"
        },
        "type": "STRING",
        "displayNames": [

                "value": "TagUpdate",
                "language": "EN"
            },

                "value": "TagMaJ",
                "language": "FR"

        ],
        "searchable": false,
        "id": "TagCreation"

]
```

Tag class search

The example below shows how to retrieve a tag class from a list of identifiers.

REST Java

```
GET {{core}}/rest/tagclass/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: comma-separated list of tag class identifiers to be retrieved

-- Headers --
token: {{token}}
```

Tag class deletion

This operation deletes a list of tag classes from a list of identifiers.

REST Java

```
DELETE {{core}}/rest/tagclass/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: comma-separated list of tag class identifiers to be deleted

-- Headers --
token: {{token}}
```

The `VirtualFolderClass` service exposes all the operations available around the `VirtualFolderClass` type components.

Development / Core APIs / Examples of use / Retrieving virtual folder classes

The examples below show how to retrieve all virtual folder classes.

REST Java

```
GET {{core}}/rest/virtualfolderclass HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json
```

Virtual folder class creation

The examples below show how to create ACLs using the operation of create.

REST Java

```
POST {{core}}/rest/virtualfolderclass HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[{
    "searches": [

        "category": "DOCUMENT",
        "request": {
          "selectClause": {
            "fields": [
              "CanalEntree",
              "ServiceDestinataire",
              "TypeCourrier"
            ]
          },
          "filterClauses": [

              "type": "com.flower.docs.domain.search.AndClause",
              "criteria": [

                  "name": "classid",
                  "operator": "EQUALS_TO",
                  "type": "STRING",
                  "values": [
                    "CourrierEntrant"
                  ]

              ]

          ],
```

```
          "orderClauses": [

              "name": "DateCourrier",
              "type": "TIMESTAMP",
              "ascending": false

          ],
          "start": 0,
          "max": 0,
          "aggregation": {
            "type":
"com.flower.docs.domain.search.FieldAggregation",
            "field": "DateCourrier"

        },
        "displayNames": [

            "language": "EN"
          },

            "language": "FR"

        ],
        "id": "searchCourrierEntrant"

    ],
    "id": "TestVF",
    "data": {
      "owner": "fadmin",
      "creationDate": "2023-08-28 10:41:52.340 +0200",
      "lastUpdateDate": "2023-08-28 10:43:49.169 +0200",
      "ACL": "acl-distribution-tab"
    },
    "displayNames": [

        "value": "TestVF",
        "language": "EN"
      },
```

```
        "value": "TestVF",
        "language": "FR"

    ],
    "descriptions": [

        "language": "EN"
      },

        "language": "FR"

    ],
    "RetentionDuration": {
      "value": 0,
      "unit": "MONTH"
    },
    "category": "VIRTUAL_FOLDER",
    "active": false
}]
```

Virtual folder class modification

This operation updates the data of a virtual folder class: tags and search

> **INFO**
>
> This service operates on a cancel and replace basis, so all tag values must be supplied by the service at the time of update. It is therefore advisable to retrieve the virtual folder class, make the modifications and call the update service.

REST Java

```
POST {{core}}/rest/virtualfolderclass/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: comma-separated list of virtual folder class identifiers to be
updated

-- Headers --

token: {{token}}
Content-Type: application/json

-- Body (json) --
[{
    "searches": [

        "category": "DOCUMENT",
        "request": {
          "selectClause": {
            "fields": [
              "CanalEntree",
              "ServiceDestinataire",
              "TypeCourrier",
                          "ClientId"
            ]
          },
          "filterClauses": [

              "type": "com.flower.docs.domain.search.AndClause",
              "criteria": [

                  "name": "classid",
                  "operator": "EQUALS_TO",
                  "type": "STRING",
                  "values": [
                    "CourrierEntrant"
                  ]
```

```
            ]

        ],
        "orderClauses": [

            "name": "DateCourrier",
            "type": "TIMESTAMP",
            "ascending": false

        ],
        "start": 0,
        "max": 0,
        "aggregation": {
          "type":
"com.flower.docs.domain.search.FieldAggregation",
          "field": "DateCourrier"

      },
      "displayNames": [

          "language": "EN"
        },

          "language": "FR"

      ],
      "id": "searchCourrierEntrant"

  ],
  "id": "TestVF",
  "data": {
    "owner": "fadmin",
    "creationDate": "2023-08-28 10:41:52.340 +0200",
    "lastUpdateDate": "2023-08-28 10:43:49.169 +0200",
    "ACL": "acl-distribution-tab"
  },
  "displayNames": [
```

```
        "value": "TestVF",
        "language": "EN"
      },

        "value": "TestVF",
        "language": "FR"

    ],
    "descriptions": [

        "language": "EN"
      },

        "language": "FR"

    ],
    "RetentionDuration": {
      "value": 0,
      "unit": "MONTH"
    },
    "category": "VIRTUAL_FOLDER",
    "active": false
}]
```

Virtual folder class search

The example below shows how to retrieve a virtual folder class from a list of identifiers.

REST Java

```
GET {{core}}/rest/virtualfolderclass/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: comma-separated list of virtual folder class identifiers to be retrieved

-- Headers --
token: {{token}}
Content-Type: application/json
```

Virtual folder class deletion

This operation deletes a list of virtual folder classes from a list of identifiers.

REST Java

```
DELETE {{core}}/rest/virtualfolderclass/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: comma-separated list of virtual folder class identifiers to be
deleted

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `VirtualFolder` service exposes all the operations available around the `VirtualFolder` type components.

Development / Core APIs / Examples of use / Virtual folder recovery

The examples below show how to retrieve virtual folders from a list of identifiers.

REST Java

```
GET {{core}}/rest/virtualFolder/{ids} HTTP/1.1

-- URL parameters --
ids: list of virtual folder identifiers to be retrieved

-- Headers --
core: FlowerDocs Core host
token: {{token}}
Content-Type: application/json
```

Virtual folder creation

The examples below show how to create a list of virtual folders.

REST **Java**

```
POST {{core}}/rest/virtualFolder/ HTTP/1.1

-- Headers --
core: FlowerDocs Core host
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

    "category": "VIRTUAL_FOLDER",
    "data": {
      "ACL": "acl-dossierclient",
      "classId": "DossierClient",
      "owner": "jna"
    },
    "name": "123654 - DOE Jules",
    "tags": [

        "name": "RefClient",
        "readOnly": false,
        "value": [
          "123654"
        ]
      },

        "name": "PrenomClient",
        "readOnly": false,
        "value": [
          "Jules"
        ]
      },

        "name": "NomClient",
        "readOnly": false,
        "value": [
          "DOE"
```

```
        ]

    ]

]
```

Virtual folder modification

This operation updates the data in a virtual folder: tags and data (class identifier, ACL, owner, etc.).

> **INFO**
>
> This service operates on a cancel and replace basis, so all tag values must be supplied by the service at the time of update. It is therefore advisable to retrieve the virtual folder, make the changes and call the update service.

REST Java

```
POST {{core}}/rest/virtualFolder/{ids} HTTP/1.1

-- URL parameters --
ids: list of virtual folder identifiers to be updated

-- Headers --
core: FlowerDocs Core host
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

    "category": "VIRTUAL_FOLDER",
    "data": {
      "ACL": "acl-dossierclient",
      "classId": "DossierClient",
      "owner": "jna"
    },
    "name": "123654 - DOE Marc",
    "tags": [

        "name": "RefClient",
        "readOnly": false,
        "value": [
          "123654"
        ]
      },

        "name": "PrenomClient",
        "readOnly": false,
        "value": [
          "Marc"
        ]
      },

        "name": "NomClient",
```

```
        "readOnly": false,
        "value": [
          "DOE"
        ]

    ]

]
```

Recherche de virtual folder

The search operations all work on the same model as described here.

Virtual folder deletion

The examples below show how to delete a list of virtual folders from a list of identifiers.

REST Java

```
DELETE {{core}}/rest/virtualFolder/{ids} HTTP/1.1

-- URL parameters --
ids: list of virtual folder identifiers to be deleted

-- Headers --
core: FlowerDocs Core host
token: {{token}}
Content-Type: application/json
```

The `ObfuscationService` service exposes a `create` operation. All parts of the document content corresponding to the search defined in the call will be automatically obfuscated.

Development / Core APIs / Examples of use / Search for areas to obfuscate

The research model used in the call for proposals is as follows:

```
"accentSensitive": true,
"caseSensitive": true,
"regex": true,
"text": "string"
```

The `text` is the value or pattern you are looking for. The default value is a pattern. To find the exact value and not a pattern, the value `regex` must be set to `false`.

The `accentSensitive` and `caseSensitive` parameters indicate that the search should be accent-sensitive and case-sensitive respectively.

Example

The examples below show how to obfuscate a value (in the example: "Demo") and a reason (here hides IBANs in the document).

Case sensitive value **Reason**

```
POST {{core}}/rest/documents/{{documentId}}/obfuscations HTTP/1.1

--URL parameters
core: FlowerDocs Core host
documentId: identifier of the document to be obfuscated

--Headers
token: {{token}}
Content-Type: application/json

--Body (json)

  "accentSensitive": true,
  "caseSensitive": false,
  "regex": false,
  "text": "Demo"
```

Development / Core APIs / Examples of use / HTML template

The `HTML` service shows the following operation:

- `getProcessedTemplate`: to download an enriched template with tag values

Model

The template used by the `getProcessedTemplate` call looks like this:

```
"string":"string"
```

This is a set of key/value pairs, the tag identifier and its value.

Reply

The reply is a text composed of the HTML template, with the tag identifiers replaced by their values.

Example

The examples below show how to obtain an HTML template by enriching it with tag values.

REST **Java**

```
POST {{core}}/rest/template/html/{id} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
id: template identifier

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --

    "MailObject": "Termination of a contract",
    "RefClient":"123456",
    "CustomerName": “Doe",
    "CustomerName": "John"
```

MS Word template

The `MSOffice` service shows the following operations:

- `downloadFromTags`: to download a template enriched with tag values
- `storeFromTags`: to create a temporary file in FLowerDocs from an enriched template

Model

The model used by the `downloadFromTags` and `storeFromTags` calls looks like this:

```
[

    "name": "string",
    "readOnly": true,
    "value": [
      "string"
    ]

]
```

Here is the description associated with the call data set:

- `name`: tag id
- `readOnly` :
- `value`: list of tag values

Reply

The `downloadFromTags` call returns the file in docx format with the replaced tag values. The `storeFromTags` call returns the identifier of the temporary file created in FlowerDocs. This identifier can then be used to create a document.

Examples - Download

The examples below show how to download an MSWord template and enrich it with tag values.

Download - REST **Download - Java**

```
POST {{core}}/rest/template/msoffice/{id} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core Host
id: MSOffice template identifier

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

    "name": "MailObject",
    "readOnly": true,
    "value": [
      "Termination of contract"
    ]
  },

    "name": "RefClient",
    "readOnly": true,
    "value": [
      "123456"
    ]
  },

    "name": "CustomerName",
    "readOnly": true,
    "value": [
      “Doe"
    ]

]
```

Examples - Temporary file

The examples below show how to create a temporary MSWord template file, enriching it with tag values.

Create - REST **Create - Java**

```
POST {{core}}/rest/template/msoffice/{id}/id HTTP/1.1

-- URL parameters --
core: FlowerDocs Core Host
id: MSOffice template identifier

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

    "name": "MailObject",
    "readOnly": true,
    "value": [
      "Termination of contract"
    ]
  },

    "name": "RefClient",
    "readOnly": true,
    "value": [
      "123456"
    ]
  },

    "name": "CustomerName",
    "readOnly": true,
    "value": [
      “Doe"
    ]

]
```

The PageService service lets you perform `showPublicPage` and `showPrivatePage` operations on your scope's pages.

- `showPublicPage` retrieves the HTML content of a public page.
- `showPrivatePage` retrieves the HTML content of a private page.

Development / Core APIs / Examples of use / Examples

The following examples show how to retrieve a public or private page from your scope.

public private

```
GET {{core}}/rest/public/{scope}/pages/{path} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
scope: the FlowerDocs scope
path: the page to retrieve

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `ProfileUserRestController` service displays the following operations:

- `search`: to search for user profiles

Development / Core APIs / Examples of use / Search user profile

The example below shows how to perform a user profile search.

SEARCH:

REST JAVA

```
GET {{core}}/rest/profiles/{profiles}/users/{username}?max={max}
HTTP/1.1
-- URL parameters --
core: FlowerDocs Core host
profiles: profile (team)
username: identifier to search
max: 1, maximum number of results returned (optional)
-- Headers --
token: {{token}}
Content-Type: application/json
```

The ACLService service displays various operations you can perform on ACLs:

- `get` retrieves all ACLs in the scope.
- `create` creates a list of `securityObjects`. The list of objects must be supplied as input, before they can be created in the application.
- `getForComponent` retrieves a component's ACL from the component's category and identifier.

- `getById` retrieves ACLs from the list of their identifiers.
- `updateById` updates ACLs using their identifiers.
- `deleteById` deletes ACLs based on their identifiers.

Development / Core APIs / Examples of use / ACL recovery

The examples below show how to retrieve ACLs using the various operations of `get`.

GET:

REST JAVA

```
GET {{core}}/rest/acl/ HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json
```

GET FOR COMPONENT:

REST JAVA

```
GET {{core}}/rest/acl/{category}/{ids} HTTP/1.1

-- URL parameters
core: FlowerDocs Core host
category: component category
ids: component identifier

-- Headers --
token: {{token}}
Content-Type: application/json
```

GET BY ID:

REST **JAVA**

```
GET {{core}}/rest/acl/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json
```

ACL creation

The examples below show how to create ACLs using the operation of `create`.

REST JAVA

```
POST {{core}}/rest/acl/ HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --
[

                "entries": [

                        "principal": "*",
                        "permission": "UPDATE_CONTENT",
                        "grant": "ALLOW"
                }],
        "id": "acl_test",
        "name": "ACL test"

]
```

ACL modification

The examples below show how to update ACLs using the operation of `update`.

REST JAVA

```
POST {{core}}/rest/acl/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: identifiers of ACLs to be modified

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json)
[

                "entries": [

                        "principal": "*",
                        "permission": "UPDATE_CONTENT",
                        "grant": "DENY"
                }],
        "id": "acl_test",
        "name": "ACL test"

]
```

Deleting ACL

The examples below show how to delete ACLs using the operation of `delete`.

REST JAVA

```
DELETE {{core}}/rest/acl/{ids} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: identifiers of ACLs to be deleted

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `UserService` service exposes the following operations:

- `create`: to create a user
- `get`: to retrieve a user
- `update`: to modify a user
- `password`: to change a user's password
- `search`: to search for users
- `delete`: to delete a user

Development / Core APIs / Examples of use / Creation et modification d'un user

Model

The model used by `create` and `update` calls looks like this:

```
"id": "string",
"firstname": "string",
"lastname": "string",
"displayName": "string",
"mail": "string",
"password": "string",
"credentialsExpired": true,
"attributes": [

    "name": "string",
    "values": [
      "string"
    ]

],
"groups": [
  "string"
],
"profiles": [
  "string"
]
```

Here is the description associated with the call data set:

- `id`: unique user identifier
- `firstname`, `lastname`, `displayName` and `mail`: user information
- `password`: password
- `profiles` and `groups`: respective lists of user profiles and groups to which this user belongs
- `attributes`: list of additional attributes
- `credentialsExpired`: if the user's credentials have expired.

Example

The examples below show how to create and modify a user.

Create - REST **Create - Java** **Update - REST** **Update - Java**

```
POST {{core}}/rest/users/ HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json

-- Body (json) --

  "id": "example",
  "firstname": "Firstname",
  "lastname": "Name",
  "displayName": "Firstname name",
  "mail": "example@gmail.com",
  "password": "password",
  "credentialsExpired": false,
  "attributes": [
  ],
  "groups": [
  ],
  "profiles": [
    "AllUsers", "eEnvelope"
  ]
```

Recovery of one or more users

Model

The parameters to be entered are :

Name	Description
`ids`	Unique identifiers of users to be tracked (separated by commas)
`resolveAuthorities`	Determines whether profiles and groups are to be remounted

Example

The example below shows how to retrieve users.

REST Java

```
GET {{core}}/rest/users/{ids}?resolveAuthorities=
{resolveAuthorities} HTTP/1.1

-- URL parameters --
core: hostFlowerDocs Core hosts: user IDs to be retrieved
resolveAuthorities : false

-- Headers --
token: {{token}}
```

Search for one or more user(s)

Model

The parameter to be entered is `search`, and corresponds to the searched value. The search can be based on the user's surname, first name, the name to be displayed (`displayName`) or the user's ID, either fully or partially filled in.

Example

The examples below show how to create and modify a user.

REST Java

```
GET {{core}}/rest/users/search?name={name} HTTP/1.1

-- URL parameters --
core: hostFlowerDocs Core hostme: user name

-- Headers --
token: {{token}}
```

Changing a user's password

Model

The parameters to be entered are :

Name	Description
`id`	The user's unique identifier
`newPassword`	The user's new password

Example

The example below shows how to change a user's password.

REST Java

```
PUT {{core}}/rest/users/{id}/password HTTP/1.1

-- URL parameters --
core: hostFlowerDocs Core host-- Headers --
token: {{token}}
id: user identifier

-- Body --

        "password": newpassword
```

Delete a user

Model

The parameter to be entered is `id`, the unique identifier of the user to be deleted.

Example

The example below shows how to delete a user.

REST Java

```
DELETE {{core}}/rest/users/{id} HTTP/1.1

-- URL parameters --
core: hostFlowerDocs Core host: user identifier

-- Headers --
token: {{token}}
```

The `token` service can be used to generate a token for specific documents, or to extend the life of a token.

Development / Core APIs / Examples of use / Token generation

The examples below show how to generate user tokens.

Generate a token with a specific lifetime

The example below shows how to generate a token with a configurable lifetime for the authenticated user.

REST Java

```
POST {{core}}/rest/token/user?validityTime={validityTime} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
validityTime: token validity in seconds

-- Headers --
token: {{token}}
Content-Type: application/json
```

Generate a new token

The example below generates a new token for the authenticated user.

REST Java

```
PUT {{core}}/rest/token/user HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host

-- Headers --
token: {{token}}
Content-Type: application/json
```

Token generation endpoints ending with `/token` are deprecated since version 2025.2.0 because they do not return an expiration date.

Generate a token to access documents

The example below generates a new token for the authenticated user to access a list of specific documents.

REST Java

```
POST {{core}}/rest/token/document/{ids}?readOnly={readOnly}
HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
ids: document identifiers for which to generate the token
readOnly: true or false for read-only or non-read-only access

-- Headers --
token: {{token}}
Content-Type: application/json
```

Token validation

The example below shows how to validate a token.

REST Java

```
POST {{core}}/rest/token/{tokenToValidate} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
tokenToValidate: the token to validate

-- Headers --
token: {{token}}
Content-Type: application/json
```

The `ScopeService` service displays the following operations:

- `get`: to retrieve information from a scope

Development / Core APIs / Examples of use / Scope fetch

The example below shows how to retrieve information from a scope.

Update - REST **Update - Java**

```
GET {{core}}/rest/scope/{idScope} HTTP/1.1

-- URL parameters --
core: FlowerDocs Core host
idScope: Identifier of scope to be recovered

-- Headers --
token: {{token}}
Content-Type: application/json
```

INFO

The Java client lets you interact with **FlowerDocs Core** through the web services on display.

This documentation is based on the use of Maven and [Spring Boot 2.7.18](https://spring.io/projects/spring-boot).

Development / Core APIs / Set-up

Maven

The libraries required for this Java client are published in the Artifactory Arondor. If you are extern to Arondor, please ask the FlowerDocs support to get the mentioned librairies.

To use the Java client in a Maven project, start by adding the following dependency :

```
<dependency>
    <groupId>com.flower.docs</groupId>
    <artifactId>flower-docs-starter-client</artifactId>
    <version>2025.3.0</version>
</dependency>
```

This dependency draws the necessary dependencies to start a FlowerDocs client project.

Spring Boot application

To define your Spring Boot application as a FlowerDocs client, add the `@FlowerDocsClient` annotation to the main class as :

```
@SpringBootApplication
@FlowerDocsClient
public class SampleClient

    public static void main(String[] args)

        SpringApplication.run(SampleClient.class, args);
```

This annotation is used to initialize the configuration required to initialize a FlowerDocs client. For the client to start up, the `ws.url` property must be defined in the *application.properties* file with the URL for access to the web services displayed by FlowerDocs.

Authentification

Calls to **FlowerDocs Core** require requests to be authenticated. A request can be authenticated by providing a FlowerDocs-specific user token.

Service account

The development of an application that interacts with FlowerDocs generally requires the use of a service account used to perform operations with **FlowerDocs Core**.

To this end, the Java client provides a utility class to simplify authentication management.

```
@Autowired
private Authenticator authenticator;
...
authenticator.authenticate("scopeId");
```

The configuration of the account used is done through the properties :

```
flower.user=<identifiant de l'utilisateur>
flower.password=<mot de passe>
```

These properties can either be passed in the Spring `application.properties` *configuration file or as JVM properties.*

Token generation

A user token can also be dynamically generated using the `AuthenticationService` service, which exposes the login method :

```
@Autowired
AuthenticationService service;
...
ContextHelper.setScope(new Id("GEC"));
Token token = service.login("user","password");
```

Development

Access to services

Access to FlowerDocs services is based on the Spring context. To retrieve a service instance, simply use the [@Autowired] annotation (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html).

For example, to access the document management service :

```
@Autowired
private DocumentService documentService;
```

Log management

FlowerDocs APIs are based on the log framework abstraction layer SLF4J. This framework abstracts from the logging framework used. The default implementation is Logback.

To instantiate a logger, simply :

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...
private static final Logger LOGGER =
LoggerFactory.getLogger(SampleClient.class);
...
LOGGER.info("Found {} documents", response.getFound());
```

INFO

The configuration of log levels is identical to that of FlowerDocs: Logs

Define protocol

By default FlowerDocs APIs are using SOAP protocol.

However it is possible to choose using REST APIs configuring your application. Add a property `rest.client.enabled` with value `true` in a *`application.properties`* file.

Development / Core APIs / REST

Documentation for this API is provided through *Swagger* exhibited by **FlowerDocs Core** under the path `{host.core}/swagger-ui/index.html`. You can also find the *Swagger* in our online demo environment here.

Development / Core APIs / DAO

The DAO API enables the development of FlowerDocs connector to access other data repositories.

The Javadoc is available online:

DAO API

With FlowerDocs Companion, a new Ribbon appears in the office tool.

Buttons are then provided to enable users to inject documents directly from the office suite, taking advantage of the tool's conversion.

Connectors / FlowerDocs Companion / Plugin for Microsoft Word

- Saving the current document in FlowerDocs in Word format
- Saving the current document in FlowerDocs in PDF format

Plugin for Microsoft Outlook

- Saving email with its attachments within FlowerDocs (MSG format)
- Saving an email attachment in FlowerDocs

Plugin for Microsoft Excel

- Saving the binder in FlowerDocs in Excel format
- Saving the binder in FlowerDocs in PDF format

Plugin for Microsoft PowerPoint

- Saving the slideshow in FlowerDocs in PowerPoint format
- Saving the slideshow in FlowerDocs in PDF format

Connectors / FlowerDocs Companion / Install FlowerDocs Companion

- Download the FlowerDocs Companion installer
- Run the downloaded msi file
- Follow the different installer steps
- Once installation is complete, check the "Launch FlowerDocs Companion" box to start the Companion on your workstation

Companion FlowerDocs is installed. A FlowerDocs ribbon is now available in all Microsoft software.

Uninstall

Uninstalling FlowerDocs Companion follows the same procedure as uninstalling a Windows program.

Connectors / FlowerDocs Companion / Configuration modification

Once FlowerDocs Companion has been installed, it must be configured by opening the configuration interface or directly from the configuration file. The configuration interface is accessible from the taskbar or from the plugin banner in Microsoft applications, only if the displayConfiguration parameter is set to 1. This graphical user interface lets you enter most of the parameters in the configuration file.

In addition, the window displayed will differ according to the user's rights. These rights correspond to the admin parameter. If it's set to 0, the popup will only allow you to connect via a login/password pair. If it’s set to 1, it will be used to configure most of the parameters present in the configuration file.

Access to FlowerDocs IHM

Plugins for Microsoft Office allow documents to be indexed within the FlowerDocs graphical user interface. To do this, you need to set up a URL to access it. This is associated with the parameter `GUI`.

Example: https://www.demo.flowerdocs.cloud/flower-docs-gui/

Access account & Scope

For access accounts, only the user ID is stored in the configuration file (user). The password is not saved for security reasons. However, once the login/password pair (associated with the rest of the configuration) has allowed the user to log in, a JWT identification token has been generated.

This token is guarded by the token parameter and will keep the session open even when the plugins are closed.

The scope is associated with the scope parameter.

Browser dimensions

When a document is exported, a browser is opened to index it before saving it. The dimensions of this browser are configurable and correspond to browserWidth for its width and browserHeight for its height. The default values are 700 and 800 respectively.

Configuration backup

To update the entire configuration, simply click on the `Save` button shown in the popup described above. This only applies to users with administrator rights.

Configuration test

To validate the configuration items, click on the Login button. If the message `User <nom> is connected.` is displayed, the configuration is operational. Otherwise, the displayed error explains the invalid parameter(s) to the user.

Connectors / FlowerDocs Companion / FAQ

#1. **Hiding configuration within applications**

The `displayConfiguration` parameter can be used to configure whether or not this action is displayed in the various Microsoft applications. By default, the action is hidden in the banner. To display it, enter the value `1` for this parameter.

#2. **A confirmation popup appears to let you know whether to update or create a new document**

All files exported or imported using FlowerDocs Companion are stored temporarily in the directory `../Companion/temp`. If this popup appears when importing a new file, a file with the same name already exists in this directory. This directory is emptied when FlowerDocs Companion is switched off.

Connectors / Plume / Getting Started

Plume is an ARender plugin that lets you write emails from your WEB browser.

This plugin is accessible via a new icon in the ARender menu bar.

> **INFO**
>
> Plume's main functions include:
>
> - Write an email,
> - Reply to an email,
> - Transfer viewed documents,
> - Select a model from the library,
> - Define a signature adapted to each user.

Two types of packaging are supplied:

- WAR (to be deployed on the same application server as the GUI)
- Executable JAR

Connectors / Plume / WAR

When Plume is deployed as a WEB application in a servlet container, a `plume.config.dir` property must be added to the JVM. This property must refer to the directory in which the configuration file `plume.properties` is stored.

The name of the configuration file can be changed by adding the property `spring.config.name` with the file name (without extension) as the value.

Executable JAR

Application configuration is based on Spring Boot mechanisms. Properties for configuring Plume and its connectors can be provided in a number of ways:

- By a JVM property :

```
java -D<name>=<value> -jar plume-<version>.jar
```

- By a property file. The name of this file (without extension) must be supplied as a JVM property `spring.config.name`. At runtime, Spring will search for this file in the following directories:
  - The `/config` subdirectory of the current (/ runtime) directory
  - The current directory

> **INFO**
>
> This documentation describes the various connectors available for Plume. At the very least, you need to define one in the `plume.properties` file in the `plume.config.dir` folder entered in the previous section.

The sender of the e-mail is indicated by the `email.from` property, with a default value of `plume@arondor.com`.

Connectors / Plume / Connectors

FlowerDocs

The FlowerDocs connector lets you store emails written with Plume in FlowerDocs. To activate it, add the `flower.enabled=true` property.

In addition, the `flower.url` property must be filled in with the URL of FlowerDocs web services.

```
flower.enabled=true
flower.url=http://&lt;host&gt;:&lt;port&gt;/<Core path>/services
```

SMTP

The SMTP connector allows you to send an e-mail written in Plume using the SMTP protocol. To activate this connector, add the property `smtp.enabled=true`

```
smtp.enabled=true
smtp.host=SSL0.OVH.NET
smtp.username=no-reply@flowerdocs.com
smtp.password=***
smtp.parameters.socketFactory.port=465
smtp.parameters.socketFactory.class=javax.net.ssl.SSLSocketFactory
smtp.parameters.starttls.enable=true
```

You can add parameters to the SMTP connector by adding properties `smtp.parameters.<name>=<value>`.

HMI activation

To activate Plume in ARender, you need to :

- Add the `plume.enabled` property to the profile supplied to ARender

```
gui.client.arender.profile=arender&plume.enabled=true
```

- Configure Plume as a **FlowerDocs GUI** plugin:

```
plugins.routes.plume.path=/plume/**
plugins.routes.plume.url=http://localhost:8080/plume
```

For more information on the use of **FlowerDocs GUI** plug-ins, see the documentation here.
These templates are determined from a document search using the `Template` class (this class can be modified using the `flower.template.class` property). The templates must therefore be accessible to Plume users.

Connectors / Plume / Filters

The models available to a user can be filtered according to different filters.

The first concerns the model source. In this documentation, only models from FlowerDocs are documented, which is why the various filter examples have the `source` property set to `flower`.

It is also possible to define filters on tags carried by models. For example, if the `Template` class has a `TemplateType` tag, it is possible to define a set of filters such as :

```
var filters = { source: ["flower"], TemplateType: ["Custom"] };
```

Model types

Depending on the type of template, different functions must be called to initialize the Plume plugin.

Email body

The first type of template concerns the body of the email to be written.

```
context.getTemplatesComponent().initTemplates(filters);
```

Signature

The second type of template concerns the signature of the user writing the email.

```
context.initSignature(filters);
```

Variables

The notion of variable can be used within templates to pre-fill them and thus facilitate the writing of an email.

By default, several variables are valued as :

- tags: all tags in the open component are added as variables,
- the URL of **FlowerDocs GUI**: `flowerURL`,
- the name of the logged-in user: `userDisplayName`,
- logged-in user's email address: `userMail`,
- the current scope: `scope`.

It is also possible to add custom variables:

```
$wnd.getARenderJS().preparePluginEvent("<name>", "<value>",
"plume");
```

Once defined, these variables can be used in models such as `{variableName}`.

Connectors / Plume / Indexing outgoing mail

Emails written in Plume can be stored in FlowerDocs. To do this, it may be necessary to index them to be able to find them again.

Outgoing mail is indexed using a `PreProcessor`. This object is used to manipulate the email on the client side, before it is sent to the server side for processing.

In the example below, we index the email written from the open component:

```
context.setPreProcessor(function (email) {
    var component = formAPI.getComponent();
    email.properties = {};
    email.properties.classid = "OutgoingMail";
    email.properties.name = email.subject;
    email.properties.CanalEntree =
component.getTagValue("CanalEntree");
    email.properties.RefClient =
component.getTagValue("RefClient");
});
```

Connectors / Plume / Clustering

> **INFO**
>
> This section addresses the situation where a Plume cluster needs to be deployed if session affinity is not preserved. In this case, several aspects need to be taken into account.

To centralise the various data stored at cluster instance level, you need to install the Redis key-value database and configure access to it from Plume. To do this, the following properties must be defined:

```
spring.redis.host=<redis host>
spring.redis.port=6379
```

> **INFO**
>
> For more information, see the [Spring Data Redis] documentation (https://docs.spring.io/spring-data/data-redis/docs/current/reference/html/)

Sessions

Plume relies on the user session to save emails as they are being written. It is therefore necessary to store sessions in Redis. To use Redis as a backend for storing HTTP sessions, add the property:

```
spring.session.store-type=redis
```

Caches

Various data are cached for performance reasons. These caches must therefore be replicated on each cluster instance. To do this, add the following property:

```
spring.cache.type=redis
```

Temporary files

The final aspect to consider is the management of temporary files used to store email attachments. These are stored temporarily in a directory (default: `${java.io.tmpdir}/plume/attachments`). This directory must be shared between the different instances of the cluster. To do this, you can change the location of the directory used with the `plume.tmp.dir` property.

Logs

Log management is based on Spring and its Logback integration. To configure the logs generated by the application, simply add the required properties to the property file.

The default log level is `INFO`.

Here are some configuration examples:

- To modify the application log level:

```
logging.level.root=trace
```

- To modify the application log level:

```
logging.level.com.arondor=ERROR
```

- To define the log file used:

```
logging.file=plume.log
```

> **INFO**
>
> For more information, see the [Spring Boot Logging] documentation (https://docs.spring.io/spring)-boot/docs/current/reference/html/howto-logging.html)

Session

It can be configured with the following parameters:

```
server.servlet.session.timeout=30
server.servlet.session.cookie.max-age=30
```

Connectors / Fast2 / Getting Started

To carry out mass and/or planned document injections within FlowerDocs, we suggest you use the Fast2 documentary ETL published by Arondor.

INFO

- This section only covers the use of the FlowerDocs injection task.
- This section does not describe how to install or configure the Fast2 product. For this, please refer to documentation or to fast2p8-support@arondor.com.

Connectors / Fast2 / Connector

To use FlowerDocs injection tasks within Fast2, it is necessary to add the FlowerDocs connnector to Fast2 libraries:

- add library `flower-docs-fast2p8-connector-2025.3.0-bundle.jar` in the folder *lib* of the installation directory
- restart Fast2

Connectors / Fast2 / Access to FlowerDocs

To inject Fast2 documents into FlowerDocs, you need to :

- add a task to the map `FlowerDocInjector`
- define the `flowerDocConnectionProvider` property with a `FlowerDocConnectionProvider` class object, then the properties:
  - `login`, the service account identifier used to authenticate to FlowerDocs
  - `password`, the service account password used to authenticate to FlowerDocs
  - `scope`, the identifier of the scope into which the documents are to be injected
  - `endPoint`, the URL used to access FlowerDocs web services (example: http://localhost:8080/flower-docs-ws/services)
- define a property `propertyHelper` with a class object `com.arondor.fast2p8.flower.docs.PropertyHelper`

Ignore certain properties

When integrating this task into a Fast2 process, documents may not be injected because they have a tag that is not referenced in the document class.

In fact, FlowerDocs refuses the creation of documents that are not valide. If these non-referenced tags are introduced by technical properties used in the Fast2 process, it is possible to exclude certain properties during the injection process. To do this, edit the `ignoredData` property.

Connectors / Fast2 / Import task

- In the map, open the task `ImportTask`
- Define the `fileScanner` property with a `DirectoryScanner` class object with the properties:
  - `filters` with the path of the description files (or `punnet`) (for example : `D:\fast2\punnet1.xml`)
  - `xmlSkipParse` disabled

Example of a punnet

To determine the test documents to be injected into FlowerDocs, we need to define a `punnet` in the XML format expected by Fast2. This XML file is used to specify the class of the document to be injected and the folders in which to file it, such as :

```
<?xml version='1.0' encoding='UTF-8'?>
<ns:punnet xmlns:ns="http://www.arondor.com/xml/document">
        <documentset>
                <ns:document name="Doc1" canCreate="true">
                        <ns:dataset>
                                <ns:data name="className"
type="string">

<ns:value>Document</ns:value>
                                </ns:data>
                                <ns:data name="name" type="string">
                                        <ns:value>My Document
2</ns:value>
                                </ns:data>
                        </ns:dataset>
                        <ns:folderset>
                                <ns:folder name="Test6">
                                        <ns:dataset>
                                                        <ns:data
name="className" type="string">

<ns:value>Folder</ns:value>
                                                        </ns:data>
                                        </ns:dataset>
                                        <ns:folder name="parents"
canCreate="true">
                                                <ns:dataset>
                                                        <ns:data
name="className" type="string">

<ns:value>Folder</ns:value>
                                                        </ns:data>
                                                </ns:dataset>
                                                <ns:folder
name="A1" canCreate="true">

<ns:dataset>
```

```
<ns:data name="className" type="string">

<ns:value>Folder</ns:value>

</ns:data>

</ns:dataset>
                                                                <ns:folder
name="A6" canCreate="true">

<ns:dataset>

<ns:data name="className" type="string">

<ns:value>Folder</ns:value>

</ns:data>

</ns:dataset>

</ns:folder>
                                                        </ns:folder>
                                                </ns:folder>
                                        </ns:folder>
                                </ns:folderset>
                        </ns:document>
                </documentset>
</ns:punnet>
```