FLOWX.AI

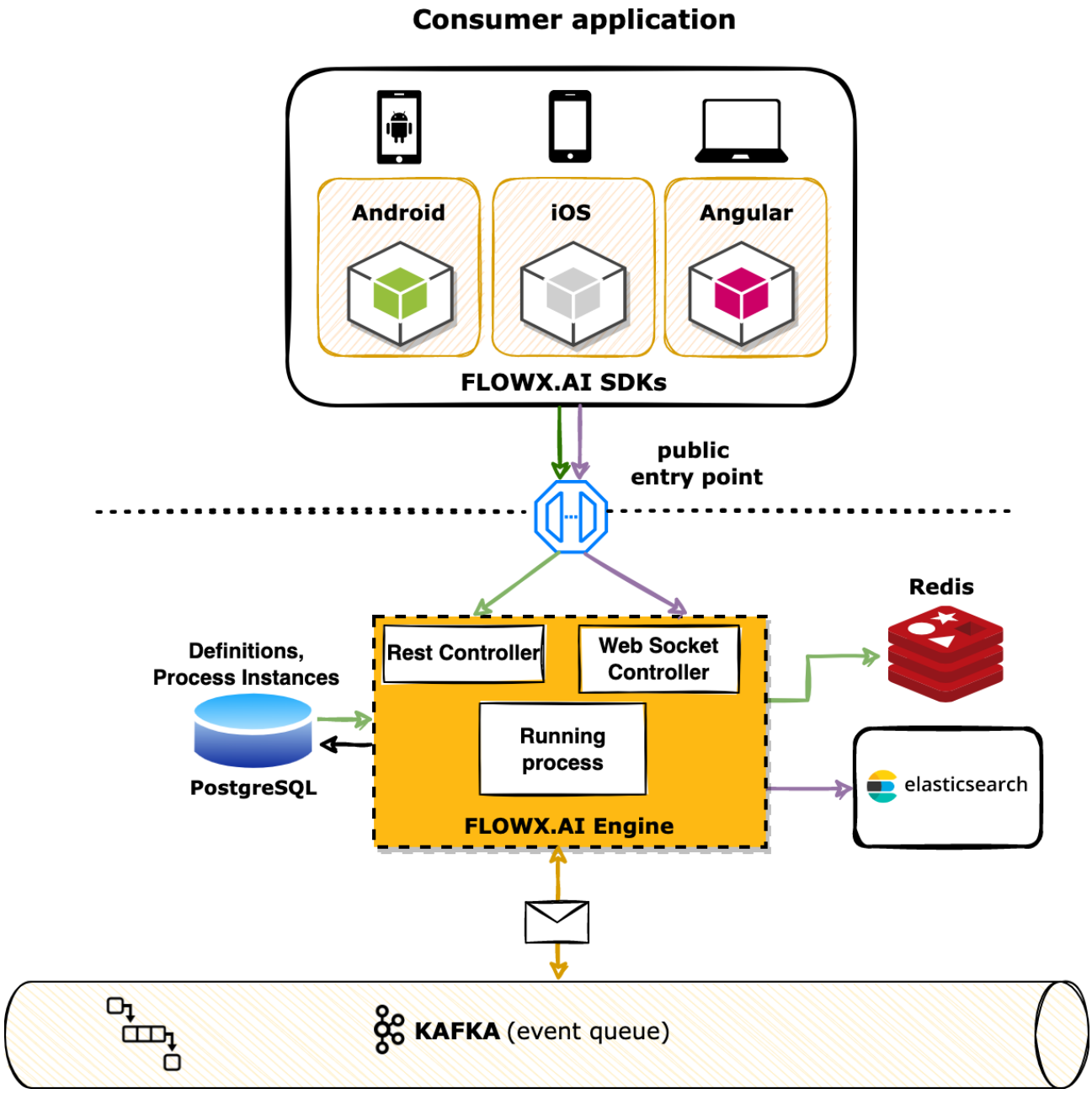
PLATFORM DEEP DIVE / Core components

Contents

PLATFORM DEEP DIVE / Core components / FLOWX.AI Engine

The engine is the core of the platform, it is the service that runs instances of the process definitions, generates UI, communicates with the frontend and also with custom integrations and plugins. It keeps track of all currently running process instances and makes sure the process flows run correctly.

A high-level overview



Orchestration

Creating and interacting with process instances is pretty straightforward, as most of the interaction happens automatically and is handled by the engine.

The only points that need used interaction are starting the process and executing user tasks on it (for example when a user fills in a form on the screen and saves the results).

REST API

The process can be started by making an API call. Certain parameters needed by the process can be sent on the request body.

Some special cases for stating process instances are:

- starting a process instance from another instance and inhering some data from the first one to the second
- a process can have multiple start nodes, in which case, a start condition must be set when making the start process call

> **POST**
> **ENGINE_URL/api/process/PROCESS_DEFINITION_NAME/start**

> **POST**
> **ENGINE_URL/api/process/PROCESS_DEFINITION_NAME/start/inheritFrom/RELATED_PROCESS_INSTANCE_UUID**

The `paramsToInherit` map should hold the needed values on one the following keys, depending on the desired outcome:

- `paramsToCopy` - this is used to pick only a subset of parameters to be inherited from the parent process; it holds the list of key names that will be

inherited from the parent parameters

- `withoutParams` - this is used in case we need to remove some parameter values from the parent process before inheriting them; it holds the list of key names that will be removed from the parent parameters

If none of these keys have values, all the parameter values from the parent process will be inherited by the new process.

GET ENGINE_URL/api/process/PROCESS_INSTANCE_ID/status

POST ENGINE_URL/api/process/PROCESS_INSTANCE_ID/token/TOKEN_INSTANCE_ID/action/ACTION_NAME/execute

POST ENGINE_URL/api/process/PROCESS_INSTANCE_ID/token/TOKEN_INSTANCE_ID/action/ACTION_NAME/upload

» FLOWX.AI Engine setup guide

Triggering or skipping nodes on a process based on Flow Names

There might be cases when you want to include or exclude process nodes based on some information that is available at start. For example, in case of a bank onboarding process, you might want a few extra nodes in the process in case the person trying to onboard is a minor.

For these cases, we have added the possibility of defining flow names. For each process node, we can choose to set if they are only available in certain cases. In the example above, we will need a flow name, let's call it *enroll_minor*. And we'll have to add this to the extra nodes.

When starting a process, in case a value is set on the *flowName* key in the values map, it will be used in order to decide which nodes to be included in a certain process instance. This means that if an adult starts the bank onboarding process, no extra key will be added when starting the process, so the extra nodes that have the *enroll_minor* flow name set will not be included. Those nodes will only be included in case a minor person starts an onboarding process.

> **INFO**
>
> If no *flowName* value is set on a node, this means the node will be included in all possible flows.

A node could also be a part of multiple flow names.

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Advancing controller

The Advancing Controller is a support service for the Process Engine that enhances the efficiency of advancing operations. It facilitates equal distribution and redistribution of the workload during scale-up and scale-down scenarios.

To achieve its functionality, the Advancing Controller microservice utilizes Postgres triggers in the database configuration.

> **INFO**
>
> A Postgres trigger is a function that is automatically executed whenever specific events, such as inserts, updates, or deletions, occur in the database.

Usage

The Advancing Controller Service is responsible for managing and optimizing the advancement process in the PostgreSQL/OracleDB databases. It ensures efficient workload distribution, performs cleanup tasks, and monitors the status of worker pods. If a worker pod fails, the service reassigns its work to other pods to prevent process instances from getting stuck. It is essential to have the Advancing Controller Service running alongside the Process Engine for uninterrupted instance advancement.

> **INFO**

> It is important to ensure that both the Process Engine and the Advancing Controller microservice are up and running concurrently for optimal performance.

Configuration

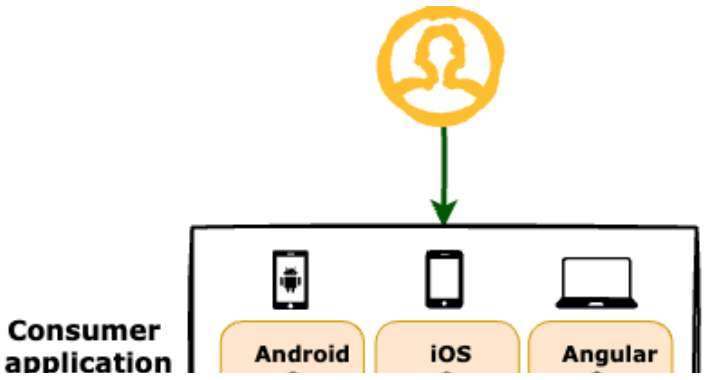
For detailed instructions on how to set up the Advancing Controller microservice, refer to the following guide:

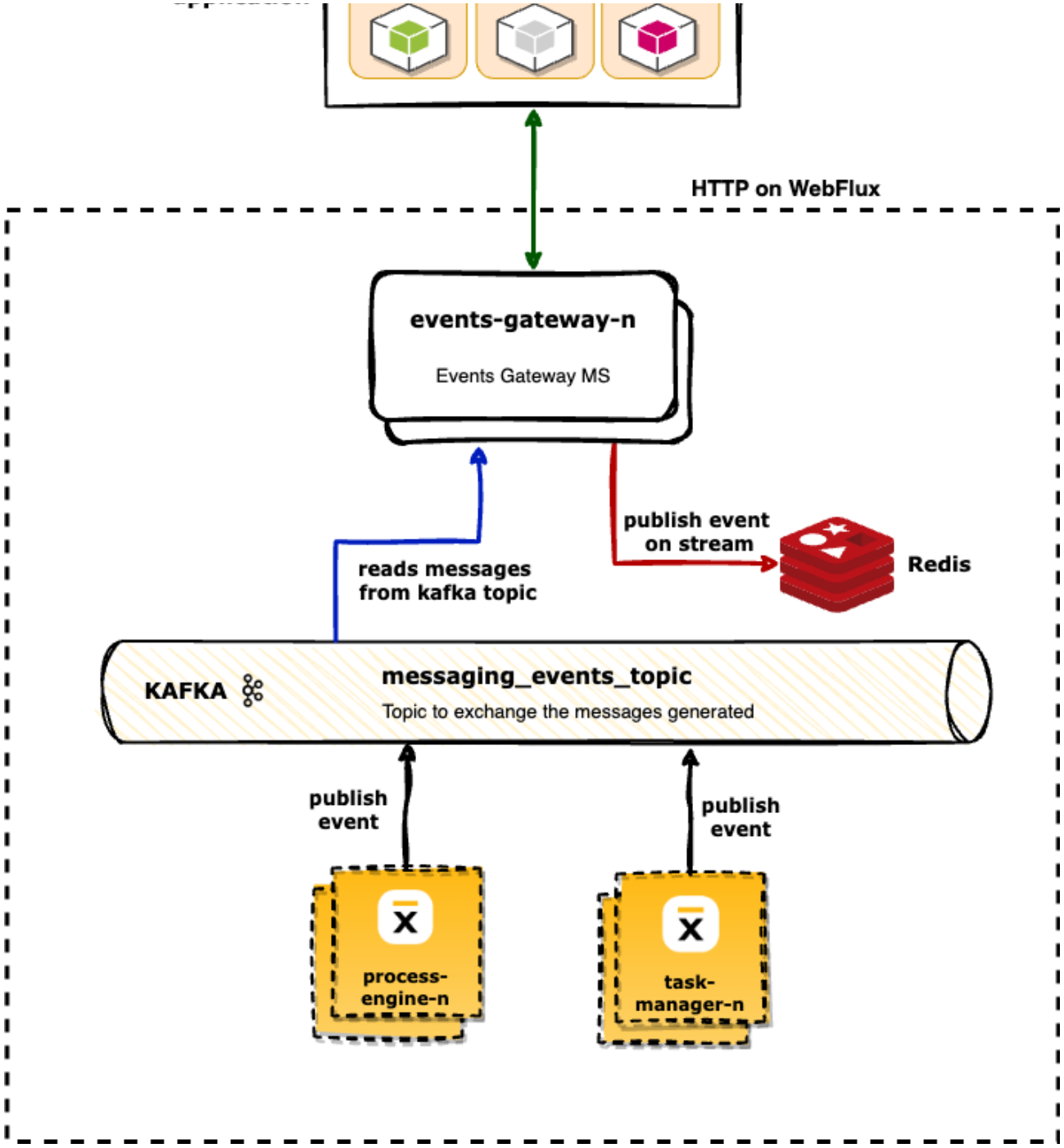
» Advancing controller setup guide

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Events gateway

Events Gateway a service that centralizes the communication with SSE (Server-sent Events) messages from Backend to Frontend.





It has an important role as a central component for handling and distributing events within the system:

- **Event Processing**: The events-gateway system is responsible for receiving and processing events from various sources, such as the Process Engine and Task Manager. It acts as an intermediary between these systems and the rest of the system components.
- **Message Distribution**: The events-gateway system reads messages from the Kafka topic (`messaging_events_topic`) and distributes them to relevant components within the system. It ensures that the messages are appropriately routed to the intended recipients for further processing.
- **Event Publication**: The events-gateway system plays a crucial role in publishing events to the frontend renderer (FE renderer). It communicates with the frontend renderer using `HTTP` via `WebFlux`. By publishing events, it enables real-time updates and notifications on the user interface, keeping the user informed about the progress and changes in the system.
- **Integration with Redis**: The events-gateway system also interacts with Redis to publish events on a stream. This allows other components in the system to consume the events from the Redis stream and take appropriate actions based on the received events.

In summary, the events-gateway system acts as a hub for event processing, message distribution, and event publication within the system. It ensures efficient communication and coordination between various system components, facilitating real-time updates and maintaining system consistency.

For more details about how to configure events-gateway microservice, check the following section:

» Events gateway configuration

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Content management / Using the service

After you deployed the CMS service in your infrastructure, you can start defining and using custom content types, such as different lists (which can have different values for the same code depending on the external system that is used), blog posts etc.

You can also set the default application name to be used in your configuration. This is needed when retrieving the contents.

```
application:
    defaultApplication: DEFAULT_APPLICATION_NAME
```

If this configuration is not set, the service will use `flowx` as the default value.

Define needed Kafka topics

Kafka topic names can be set by using environment variables:

Default parameter (env var)	Default FLOWX.AI val
KAFKA_TOPIC_REQUEST_CONTENT_IN	ai.flowx.dev.plugin.cms.trigger.r
KAFKA_TOPIC_REQUEST_CONTENT_OUT	ai.flowx.dev.engine.receive.plug

CAUTION

The Engine is listening for messages on topics with names of a certain pattern, make sure to use an outgoing topic name that matches the pattern configured in the Engine.

Example: Request a label by language or source system code

Used to translate custom codes into labels using the specified language or a certain source system.

Various external systems and integrations might use different labels for the same information. In the processes, it is easier to use the corresponding code and translate this into the needed label when necessary: for example when sending data to other integrations, when generating documents, etc.

You will need to add a Kafka send event CMS service.

The following values are expected in the request body:

- at least one of `language` and `sourceSystem` should be defined (if you only need the sourceSystem to be translated, you can leave language empty and

vice versa, but they cannot both be empty)

- a list of `entries` to be translated

Example:

```
{
  "language": "en-US",
  "sourceSystem": "CS"
  "entries": [
    {
      "codes": [
        "ROMANIA",
        "BAHAMAS"
      ],
      "contentDescription": {
        "name": "country",
        "application": "flowx",
        "version": 1,
        "draft": true
      }
    }
  ]
}
```

If the value for `application` is not sent, the `defaultApplication` value will be used when retrieving the contents from the database.

`version` and `draft` are not mandatory, if they are not specified, the latest published content will be used.

The service will respond with the following message structure:

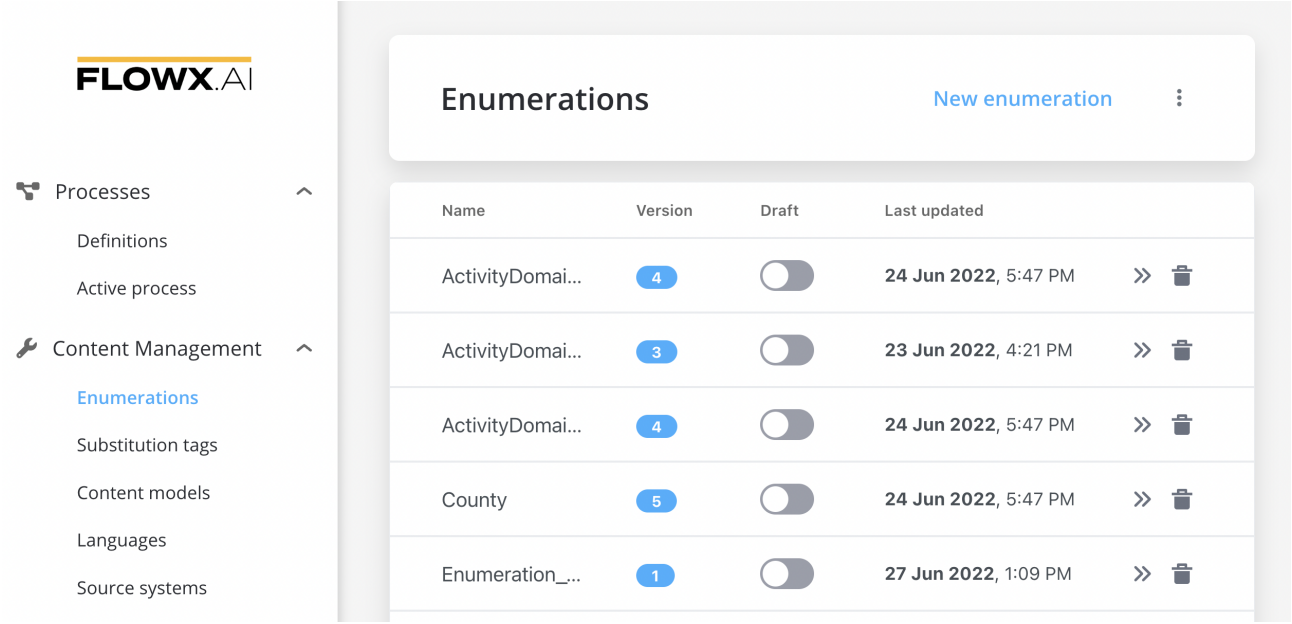
```
{
  "entries": [
    {
      "code": "ROMANIA",
      "label": "ROMANIA -en"
      "translatedCode": "ROMANIA-CS"
    },

    {
      "code": "BAHAMAS",
      "label": "BAHAMAS -en"
      "translatedCode": "BAHAMAS-CS"
    }
  ],
  "error": null
```

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Content management / Enumerations

A collection of values that can be utilized as content in UI components or templates is managed using

The fallback content to display on prerendering
. Values can be defined for certain source systems or languages.

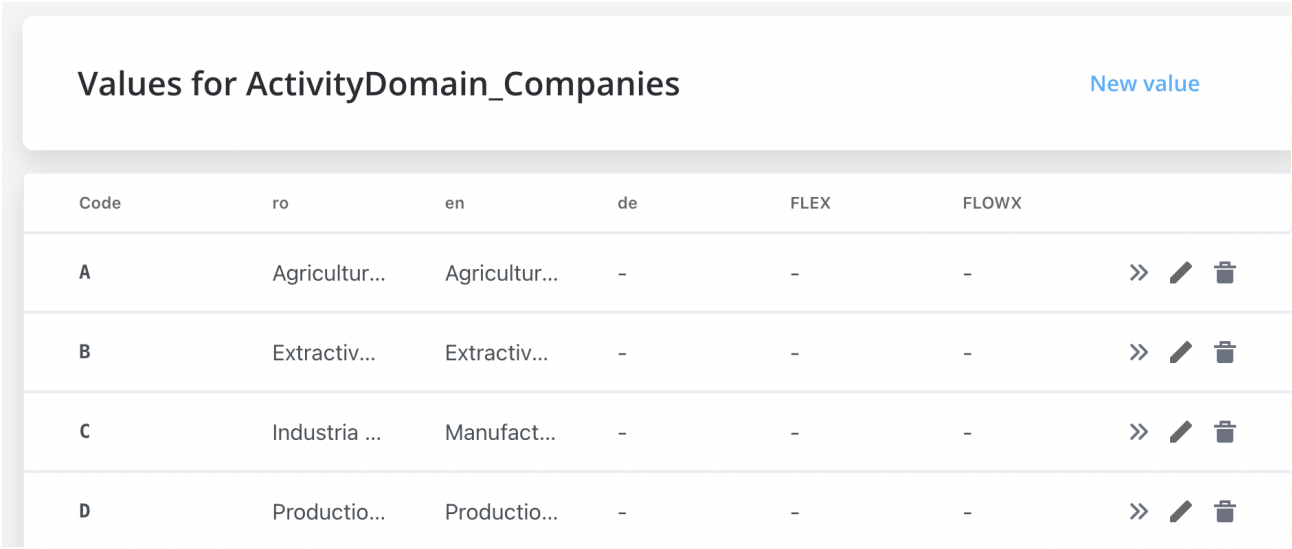


On the main screen inside **Enumerations**, you have the following elements:

- **Name** - the name of the enumeration
- **Version** - the version of the enumeration
- **Draft** - switch button used to control the status of an enumeration, could be **Draft** or **Published**
- **Last Updated** - the last time an enumeration has been updated
- **Open** - button used to access an enumeration to configure it/ add more values, etc.
- **Delete** - button used to delete an enumeration
- **New enumeration** - button used to create a new enumeration
- **Breadcrumbs** >** **Import/Export**

For each entry (when you hit the **Open** button) inside an enumeration we have to define the following properties:

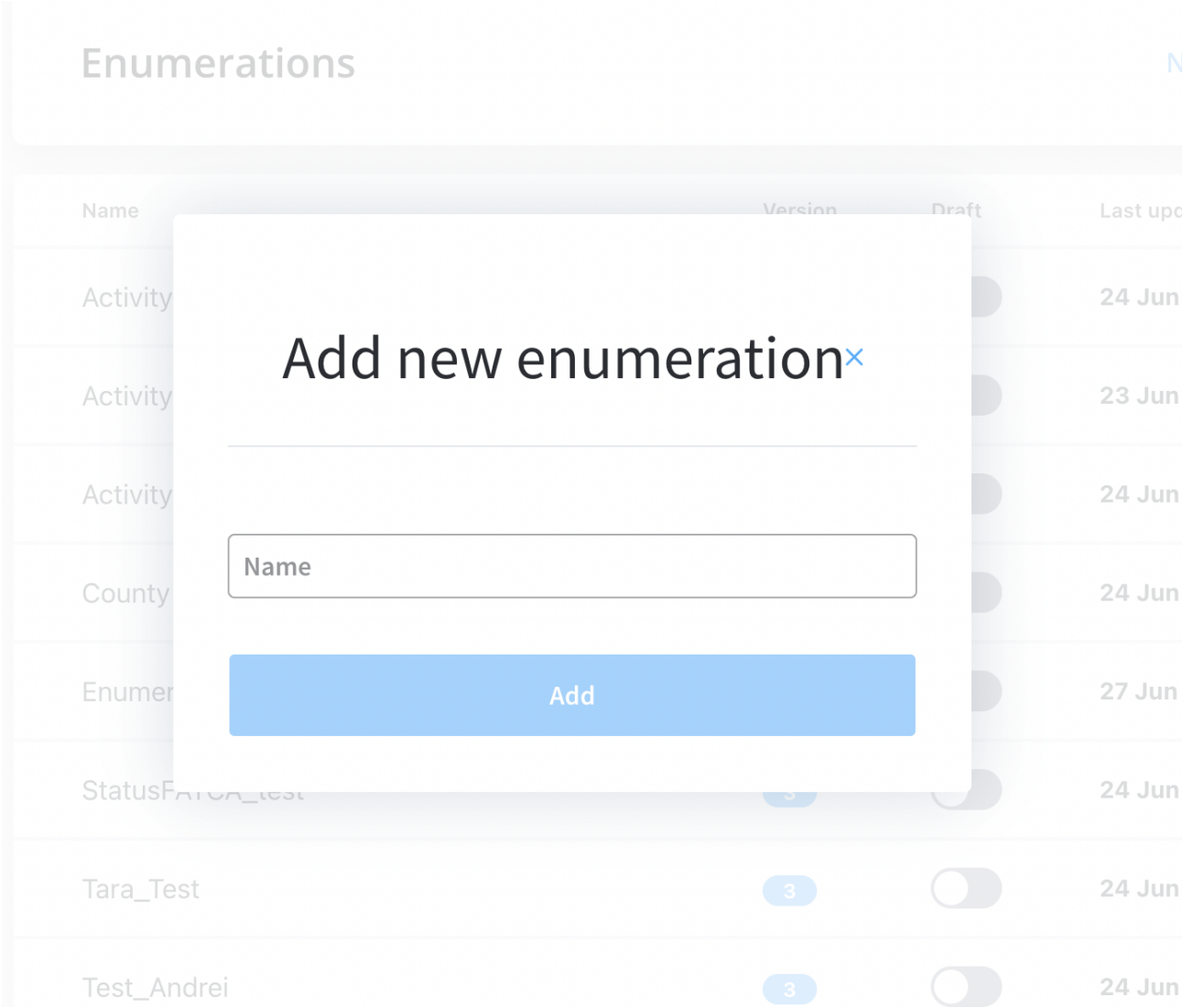
- **Code** - not displayed in the end-user interface, but used to assure value uniqueness
- **Labels** - strings that are displayed in the end-user interface, according to the language set for the generated solution
- **External source systems codes** - values that are set for each external system that might consume data from the process; these codes are further used by connectors, in other to send to an external system a value that it can validate



Adding a new enumeration

To add a new enumeration, follow the next steps:

1. Go to **FLOWX Designer** and select the **Content Management** tab.
2. Select **Enumerations** from the list.
3. Add a suggestive name for your enumeration and then click **Add**.



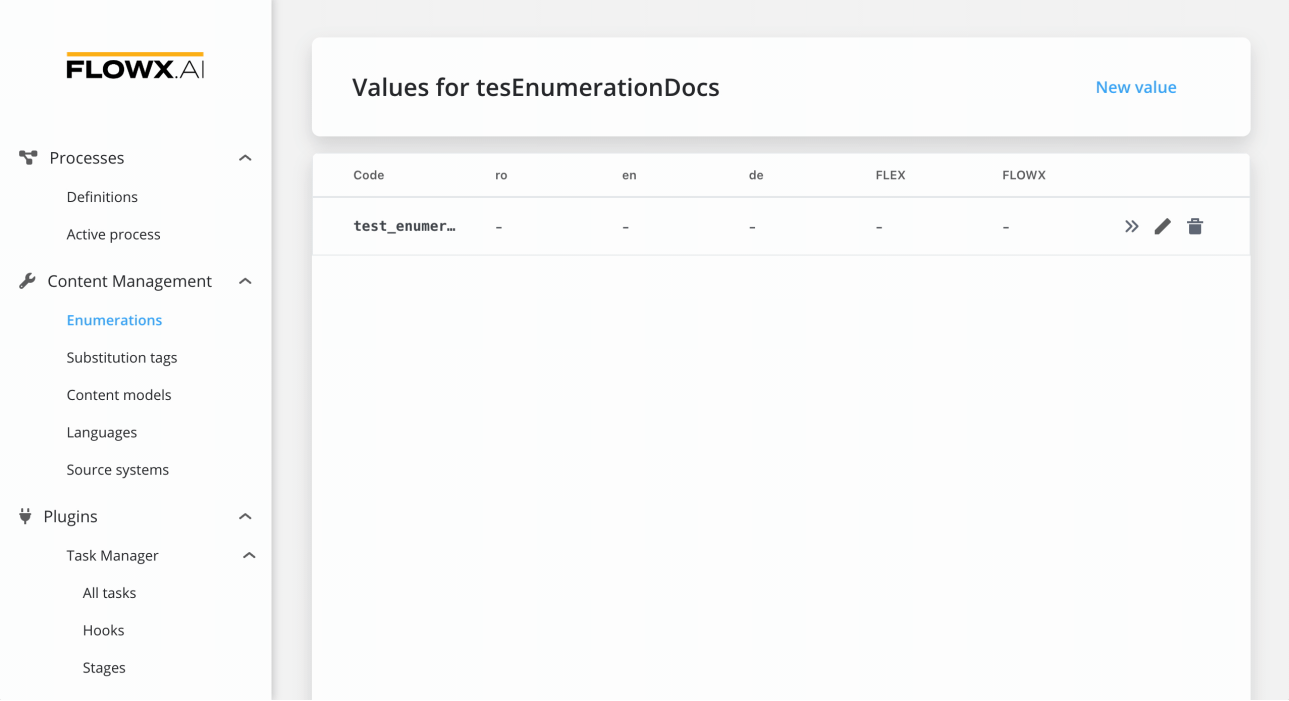
Configuring an enumeration

After creating an enumeration, you can add values to it.

To configure an enumeration value, follow the next steps:

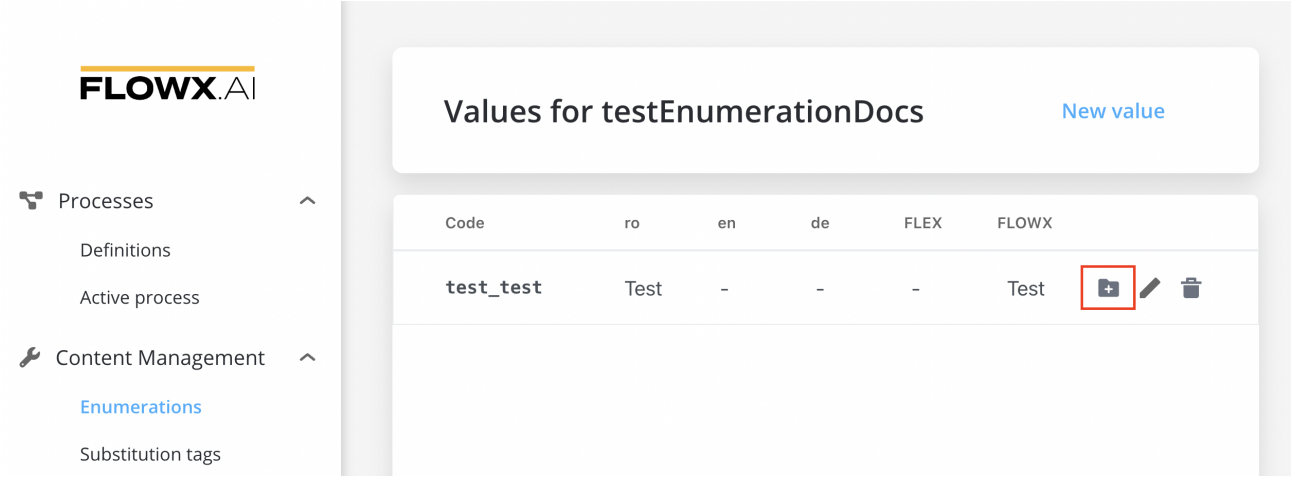
1. Go to FLOWX.AI Designer and select the **Content Management** tab.
2. Select **Enumerations** from the list and open an enumeration.
3. Click **New value** and fill in the necessary details:

- **Code** - as mentioned above, this is not displayed in the end-user interface but is used to assure value uniqueness
- **Labels** - set the value of the string for each language you would like to use
- **Source Systems** - values that are set for each external system that might consume data from the process



Creating a child collection

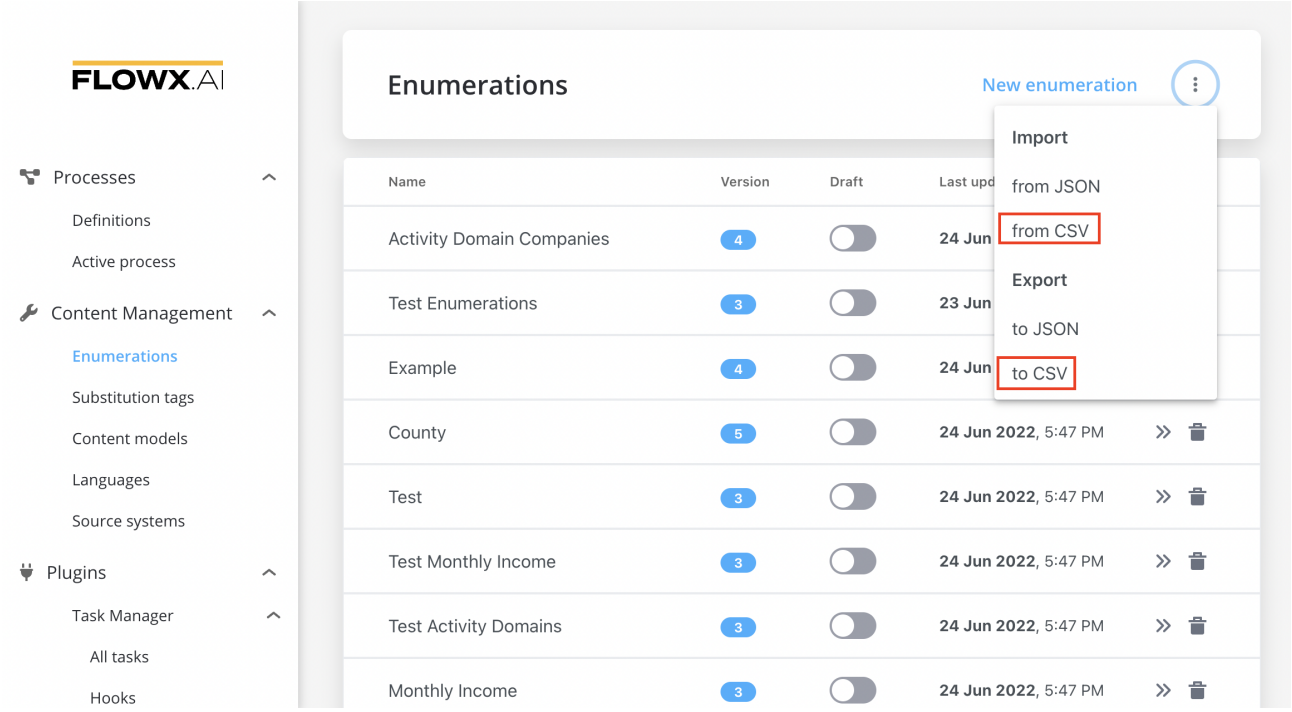
Enumerations can also be defined as a hierarchy - for each entry, we can define a list of children values (for example, name of the countries defined under the continents' enumeration values); hierarchies further enable cascading values in the end-user interface (for example, after selecting a continent in the first select UI component, the second select component will contain only the children of this continent).



Importing/exporting an enumeration

You can use the import/export feature to import or export enumerations using the following formats:

- JSON
- CSV



Enumerations example

Enumerations, for instance, can be used to build elaborate lists of values (with children). Assuming you wish to add enumerations for Activity Domain Companies, you can create children collections by grouping lists and other related domains and activities.

We have the following example for ***Activity Domain Companies***:

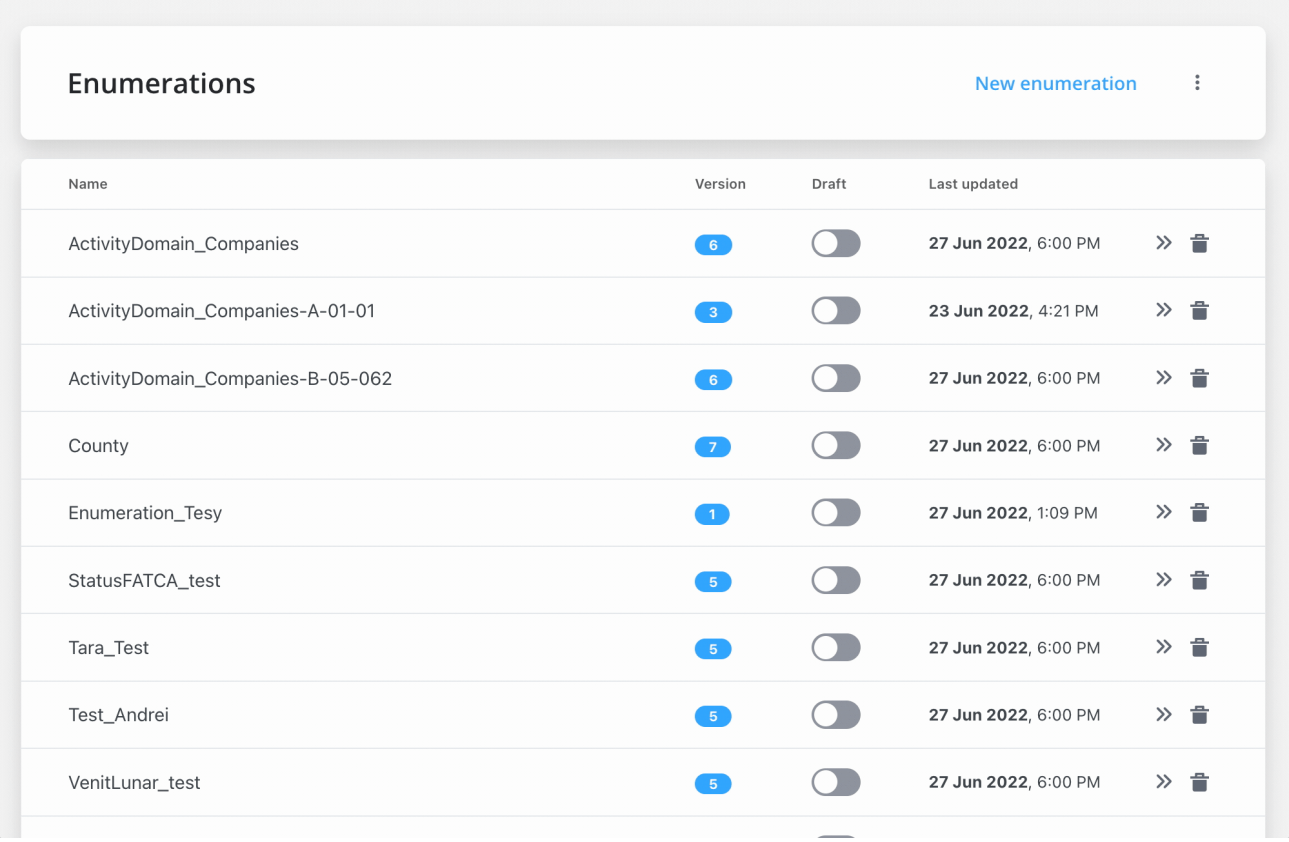
Activity Domain Companies → Agriculture forestry and fishing:

- **Agriculture, hunting, and related services →**

▸ Cultivation of non-perennial plants:

- Cultivation of plants from permanent crops:
- Animal husbandry:
- **Forestry and logging →**
- Forestry and other forestry activities:
- Logging:
- Collection of non-wood forest products from spontaneous flora:
- **Fisheries and aquaculture →**
- Fishing:
- Aquaculture:

This is the output after adding all the lists/collections from above:



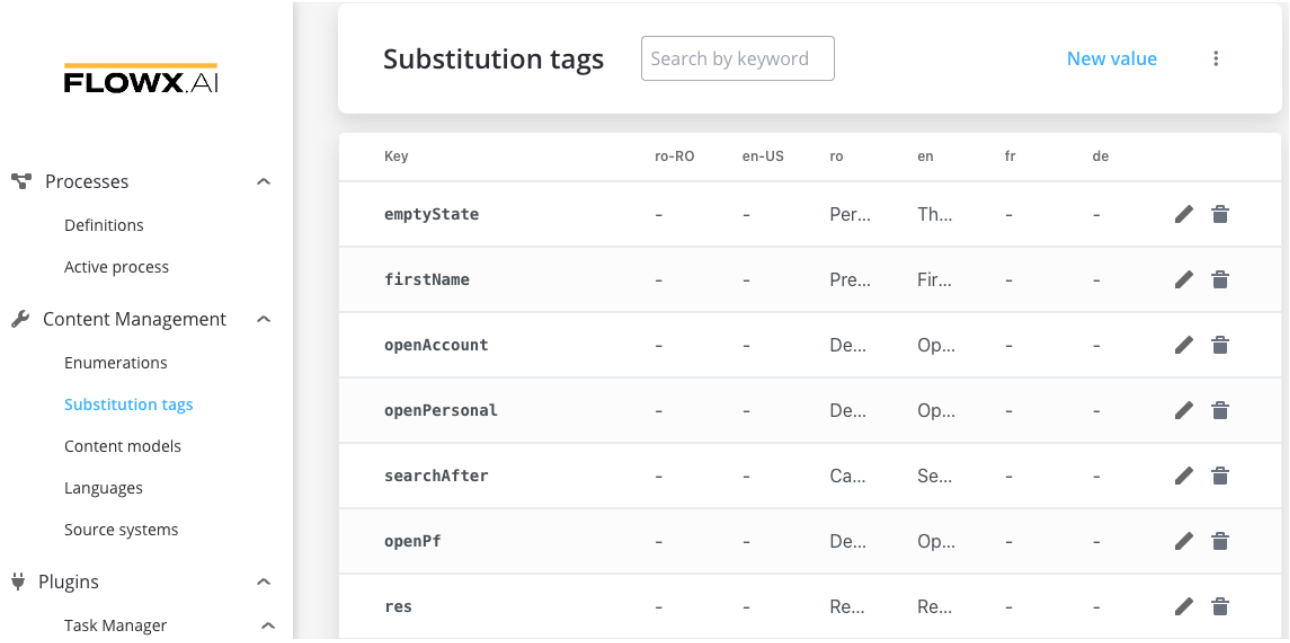
Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Content management / Substitution tags

Substitution tags are used to generate dynamic content across the platform. As

The fallback content to display on prerendering

, substitution tags can be defined for each language set for the solution.



On the main screen inside **Substitution tags**, you have the following elements:

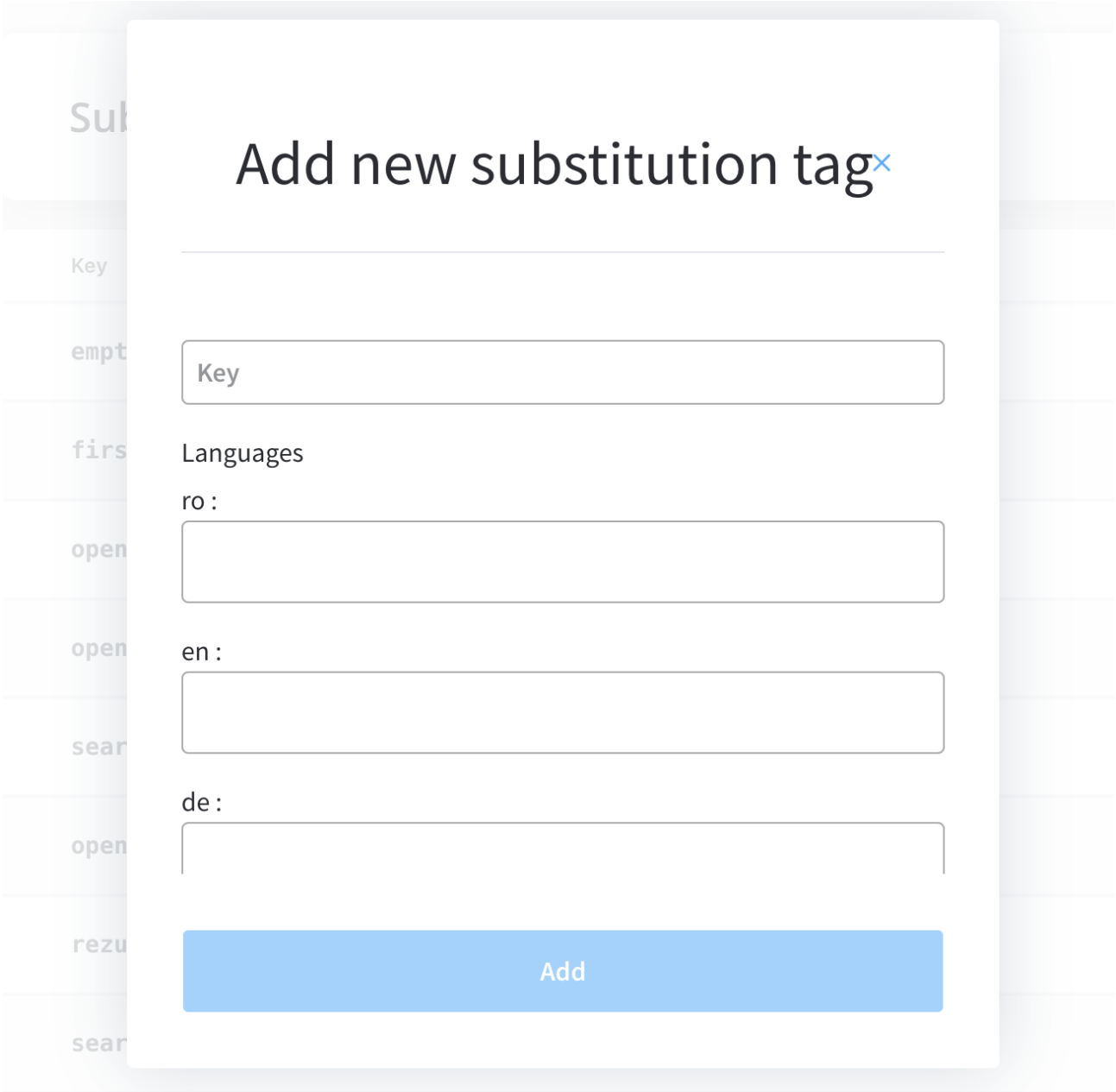
- **Key**
- **Values** - strings that are used in the end-user interface, according to the language set for the generated solution
- **Edit** - button used to edit substitution tags
- **Delete** - button used to delete substitution tags
- **New value** - button used to add a new substitution tag
- **Breadcrumbs menu**:
  - **Import**
    - from JSON
    - from CSV
  - **Export**
    - to JSON

    - to CSV
- **Search by** - search function used to easily look for a particular substitution tag

Adding new substitution tags

To add a new substitution tag, follow the next steps.

1. Go to
   The fallback content to display on prerendering
   and select the **Content Management** tab.
2. Select **Substitution tags** from the list.
3. Click **New value**.
4. Fill in the necessary details:
    - Key
    - Languages
5. Click **Add** after you finish.



CAUTION

When working with substitution tags or other elements that imply values from other languages defined in the CMS, when running a

The fallback content to display on prerendering

, the default values extracted will be the ones marked by the default language.

Getting a substitution tag by key

```
public func getTag(withKey key: String) -> String?
```

All substitution tags will be retrieved by the SDK before starting the first process and will be stored in memory.

Whenever the container app needs a substitution tag value for populating the UI of the custom components, it can request the substitution tag using the method above, providing the key.

For example, substitution tags can be used to localize the content inside an application.

Example

Localizing the app

> **INFO**
>
> You must first check and configure the FLOWX.AI Angular renderer to be able to replicate this example. Click here for more information.

The `flxLocalize` pipe is found in the `FlxLocalizationModule`.

```
import { FlxLocalizationModule } from 'flowx-process-
renderer';
```

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-dummy-component',
  template: ` <h3>{{ "stringToLocalize" | flxLocalize}}
</h3>`,

})
export class DummyComponent{
 stringToLocalize: string = `@@localizedString`
}
```

Strings that need to be localized must have the '**@@**' prefix which the **flxLocalize** pipe uses to extract and replace the string with a value found in the substitution tags enumeration.

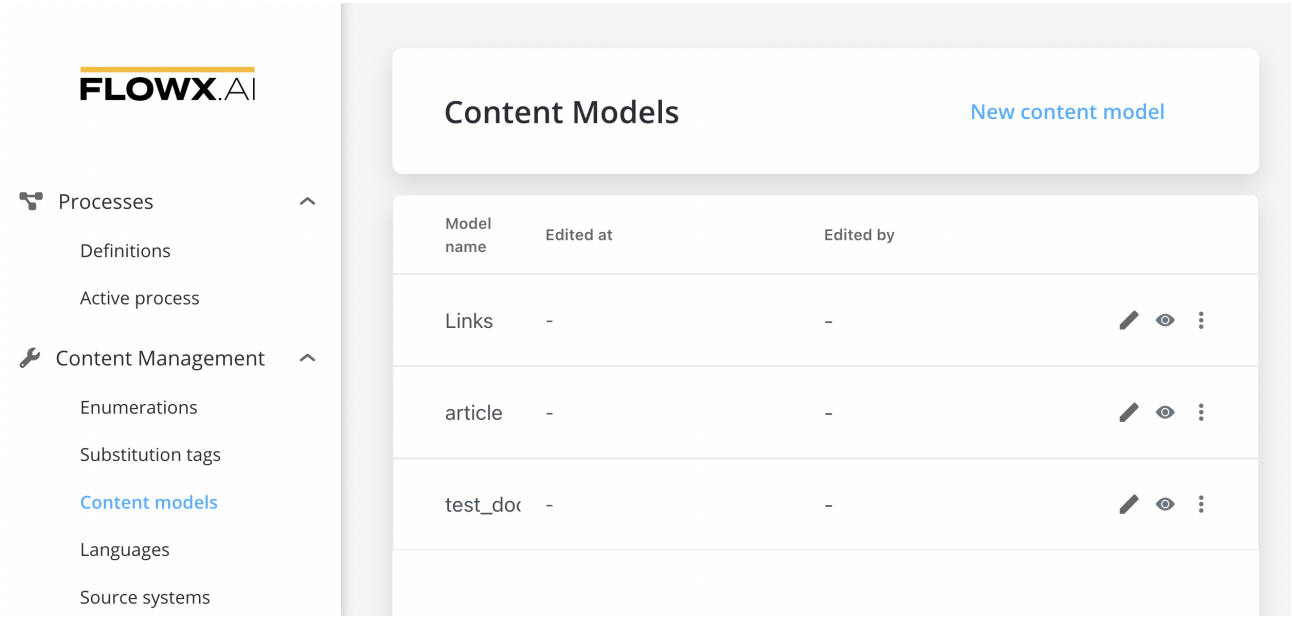
Substitution tags are retrieved when a start process call is first made, and it's cached on subsequent start process calls.

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions /

Content management / Content models

Content models are used to create complex content collections using customizable pieces of content. For example, you can define content models for an article that will be displayed on a page, creating customizable content.



On the main screen inside **Content models**, you have the following elements:

- **Model name** - the name of the content model
- **Edited at**
- **Edited by**
- **Edit** - button used to edit a content model
- **View process** - click **View process** to open a content model and see its attributes

- **Export/ Delete (Breadcrumbs menu)** - use the breadcrumbs to access the **Export** and **Delete** functions
- **New content model** - button used to add a new content model

For each entry (when you hit the **Open** button) inside an enumeration we have to define the following properties:

- **Language**
- **Name**
- **Attributes (depending on what you add as attributes)**

Configuring attributes

Attributes that can be added to a content model can have the following values:

- **String**
- **Number**
- **Boolean**

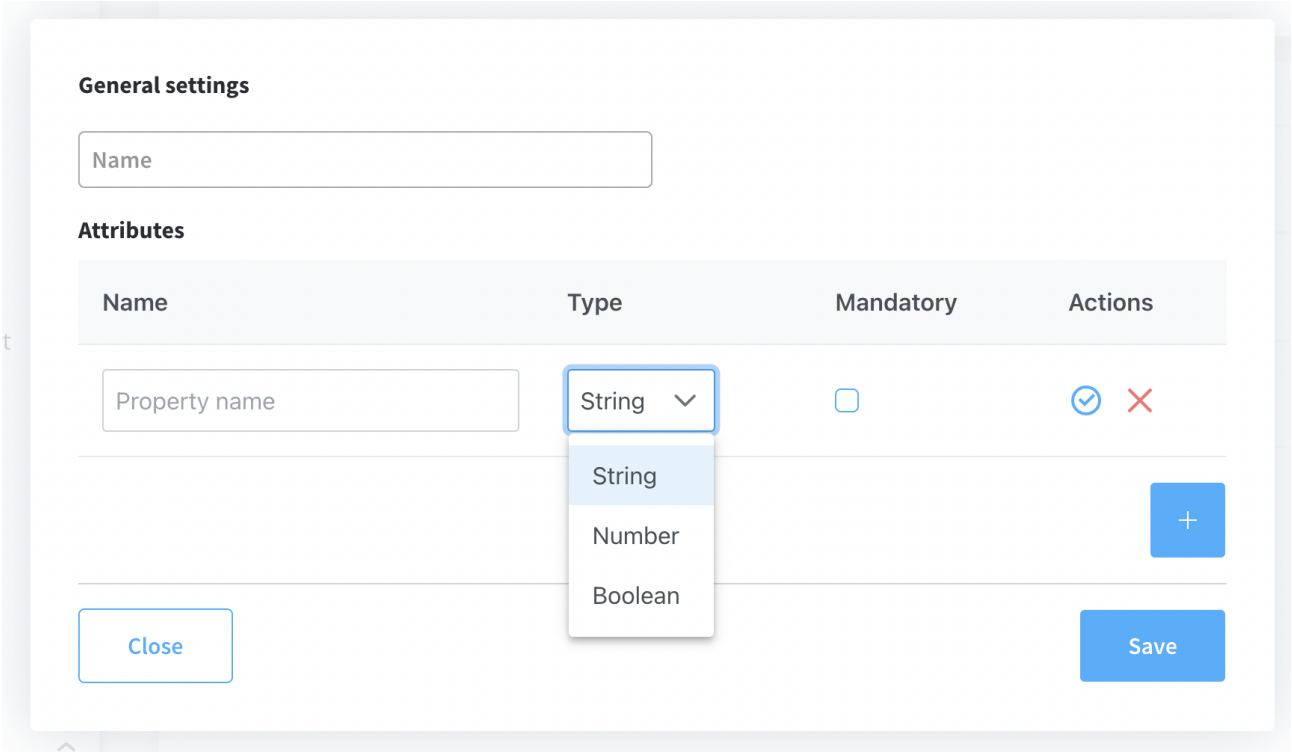
Also, attributes can be either **Mandatory** or **Optional**

Adding new content models

To add a new content model, follow the next steps:

1. Go to **FLOWX Designer** and select the **Content Management** tab.
2. Select **Content models** from the list.
3. Click **New content model**.
4. Add a suggestive name for your content model.

5. Fill in the following details:
    - **Name** (mandatory)
    - **Attributes**:
        - **Name** - add a name for the attribute
        - **Type** - String, Number or Boolean
        - **Mandatory** - mark an Attribute as mandatory
        - **Actions** - add or remove

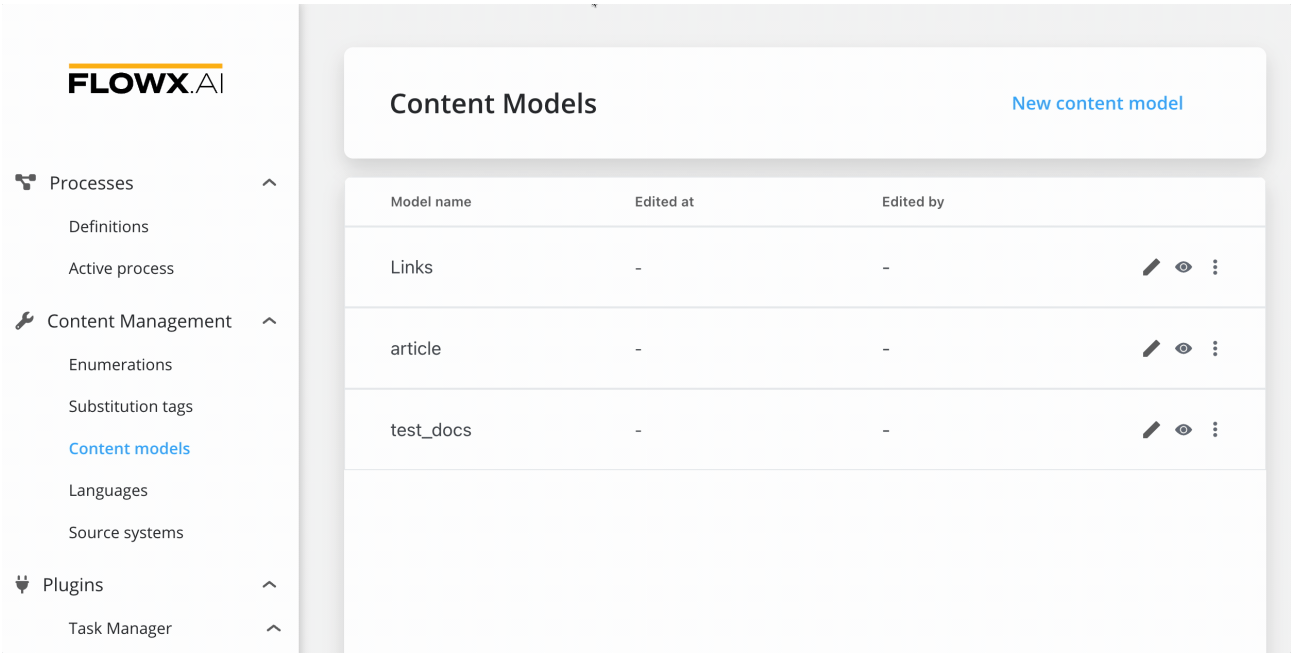


Editing content models

To configure an entry inside a content model, follow the next steps:

1. Go to **FLOWX Designer** and select the **Content Management** tab.
2. Select **Content models** from the list and select a content model.

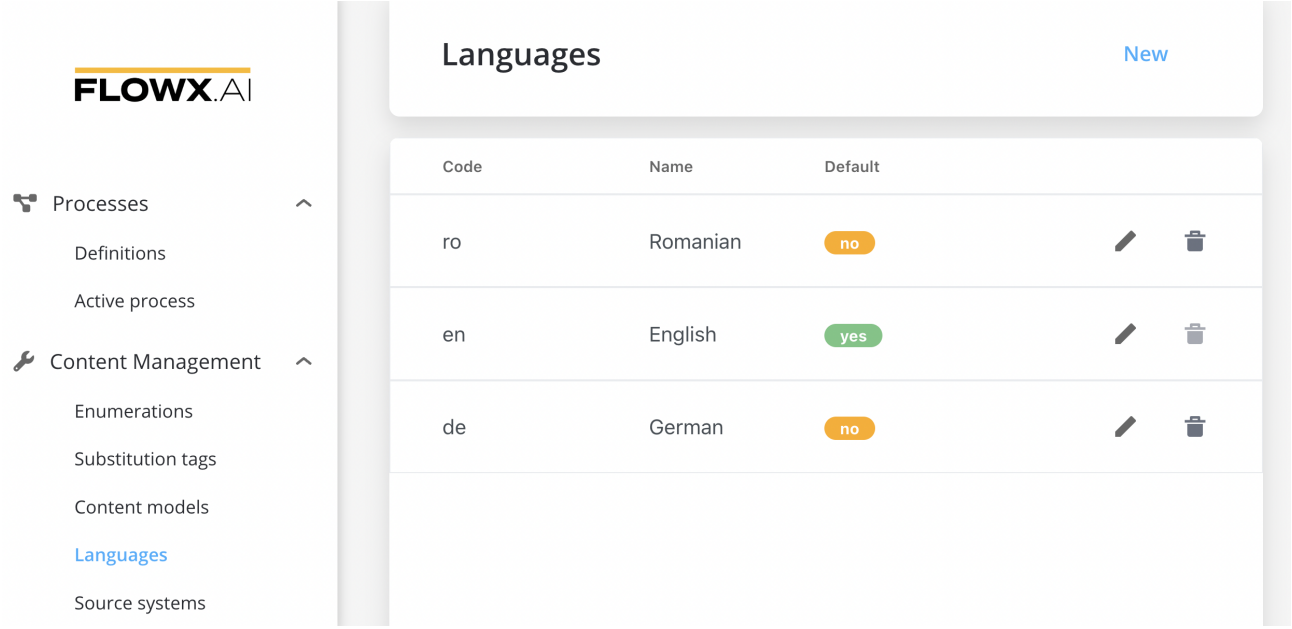
3. Click **View process** and then click **Open**.
4. Click **Edit** to modify the values.



Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Content management / Languages

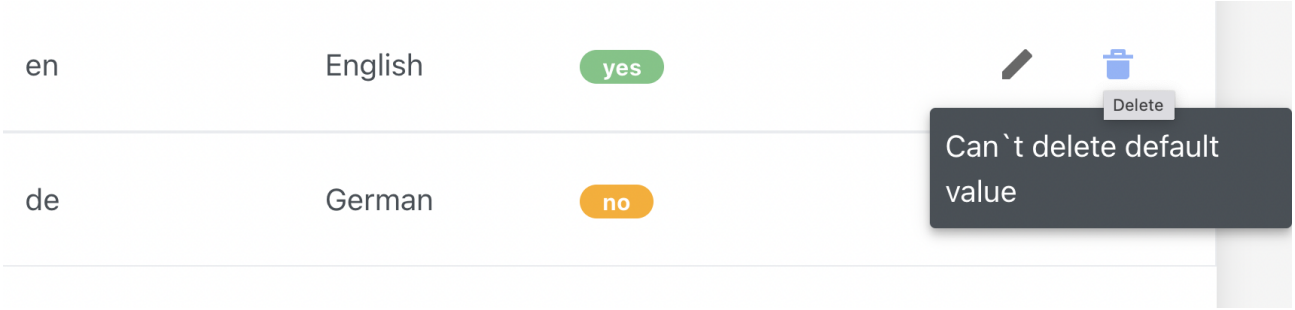
The FLOWX Headless CMS can store and manage languages. You can add a language and use it in almost any content management configuration.



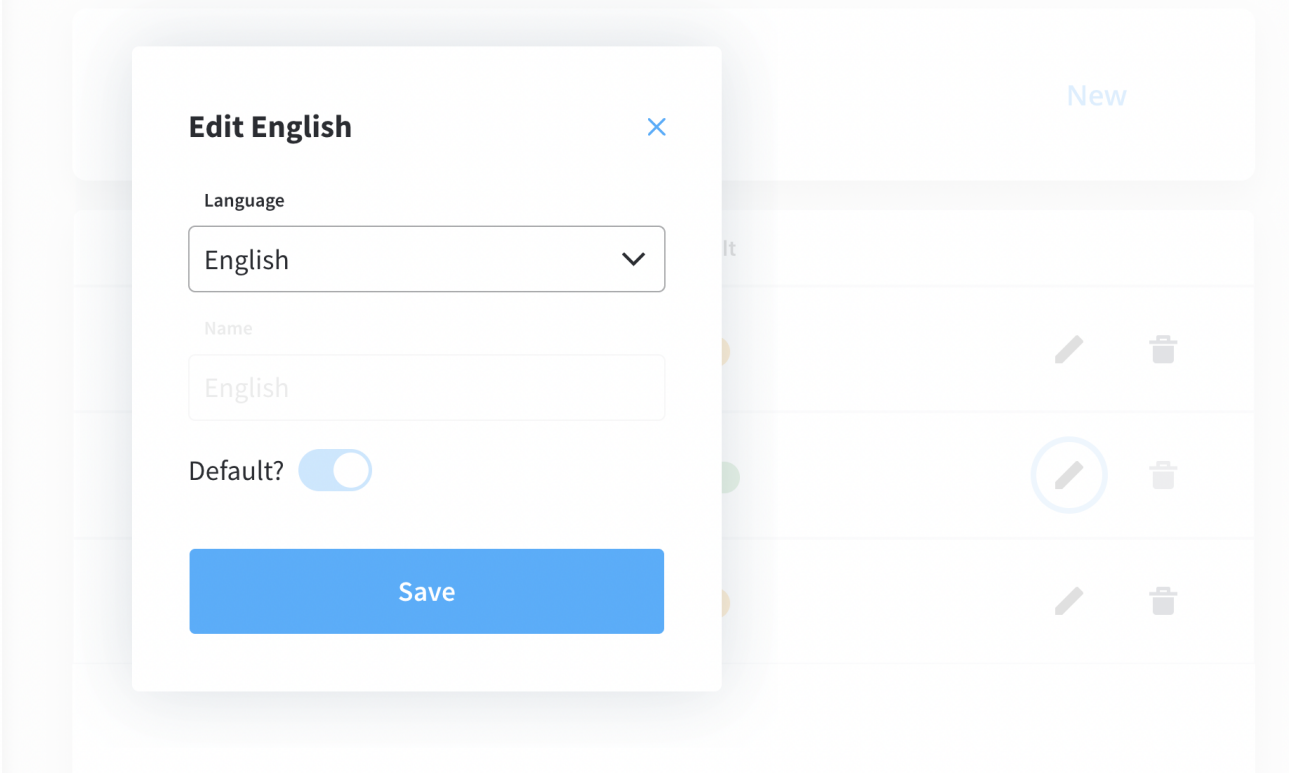
On the main screen inside **Languages**, you have the following elements:

- **Code** - not displayed in the end-user interface, but used to assure value uniqueness
- **Name** - the name of the language
- **Default** - you can set a language as **Default** (default values can't be deleted)

> **CAUTION**
>
> When working with substitution tags or other elements that imply values from other languages defined in the CMS, when running a process, the default values extracted will be the ones marked by the default language.



- **Edit** - button used to edit a language



- **Delete** - button used to delete a language

DANGER

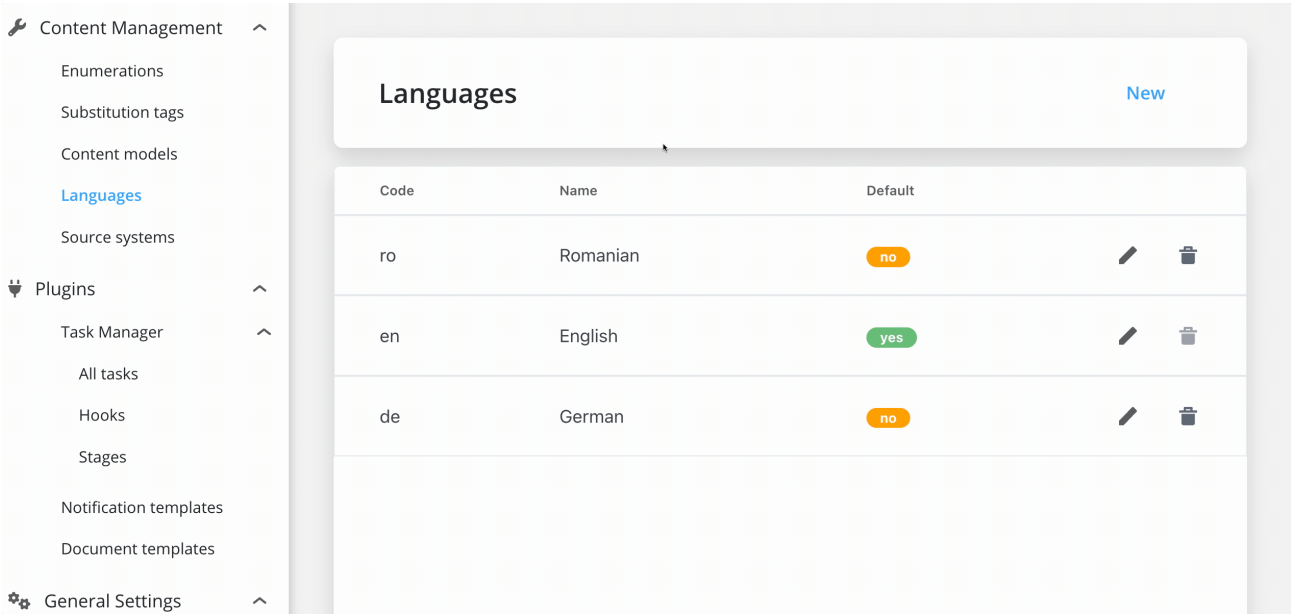
Before deleting a language make sure that this is not used in any content management configuration.

- **New** - button used to add a new language

Adding a new language

To add a new language, follow the next steps:

1. Go to
   The fallback content to display on prerendering
   and select the **Content Management** tab.
2. Select **Languages** from the list.
3. Choose a new **language** from the list.
4. Click **Add** after you finish.

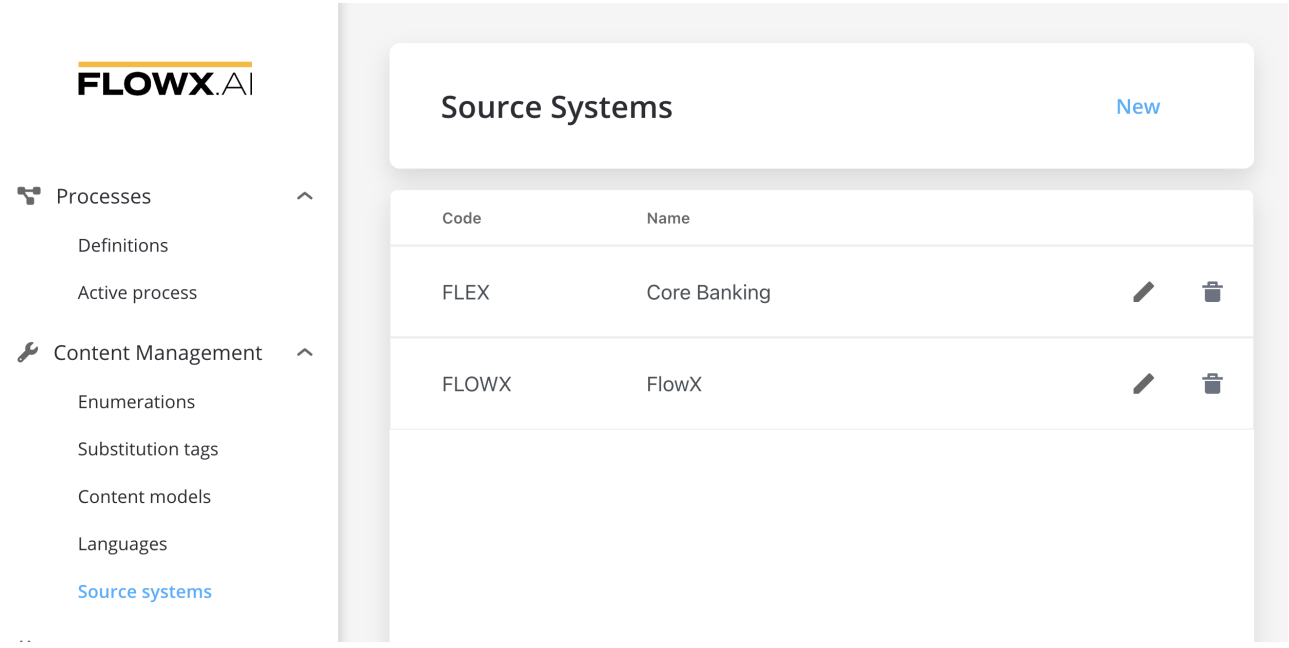


Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Content management / Source systems

If multiple

The fallback content to display on prerendering
values are needed to communicate with other systems, source systems can be used.



On the main screen inside **Source systems**, you have the following elements:

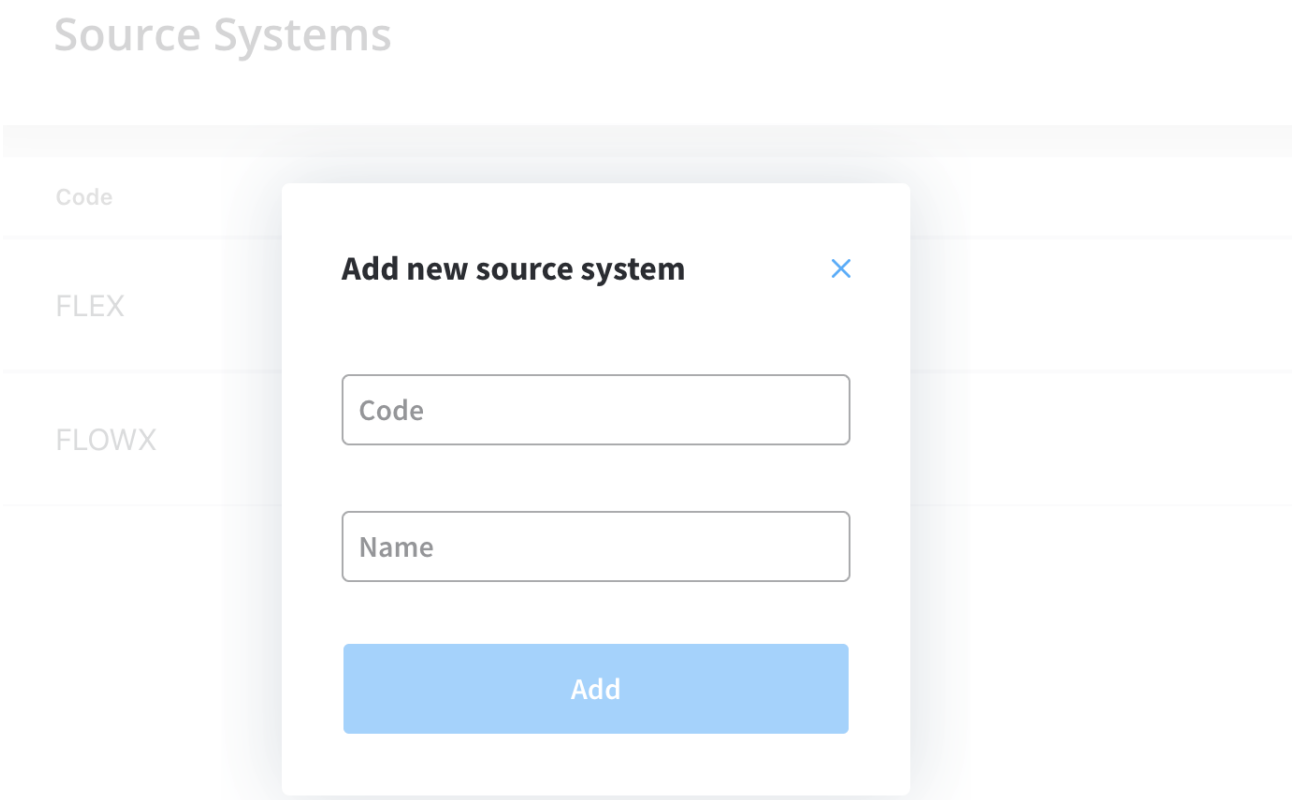
- **Code** - not displayed in the end-user interface, but used to assure value uniqueness

- **Name** - the name of the source system
- **Edit** - button used to edit a source system
- **Delete** - button used to delete a source system
- **New** - button used to add a new source system

Adding new source systems

To add a new source system, follow the next steps.

1. Go to
   The fallback content to display on prerendering
   and select the **Content Management** tab.
2. Select **Source systems** from the list.
3. Fill in the necessary details:
   - Code
   - Name
4. Click **Add** after you finish.

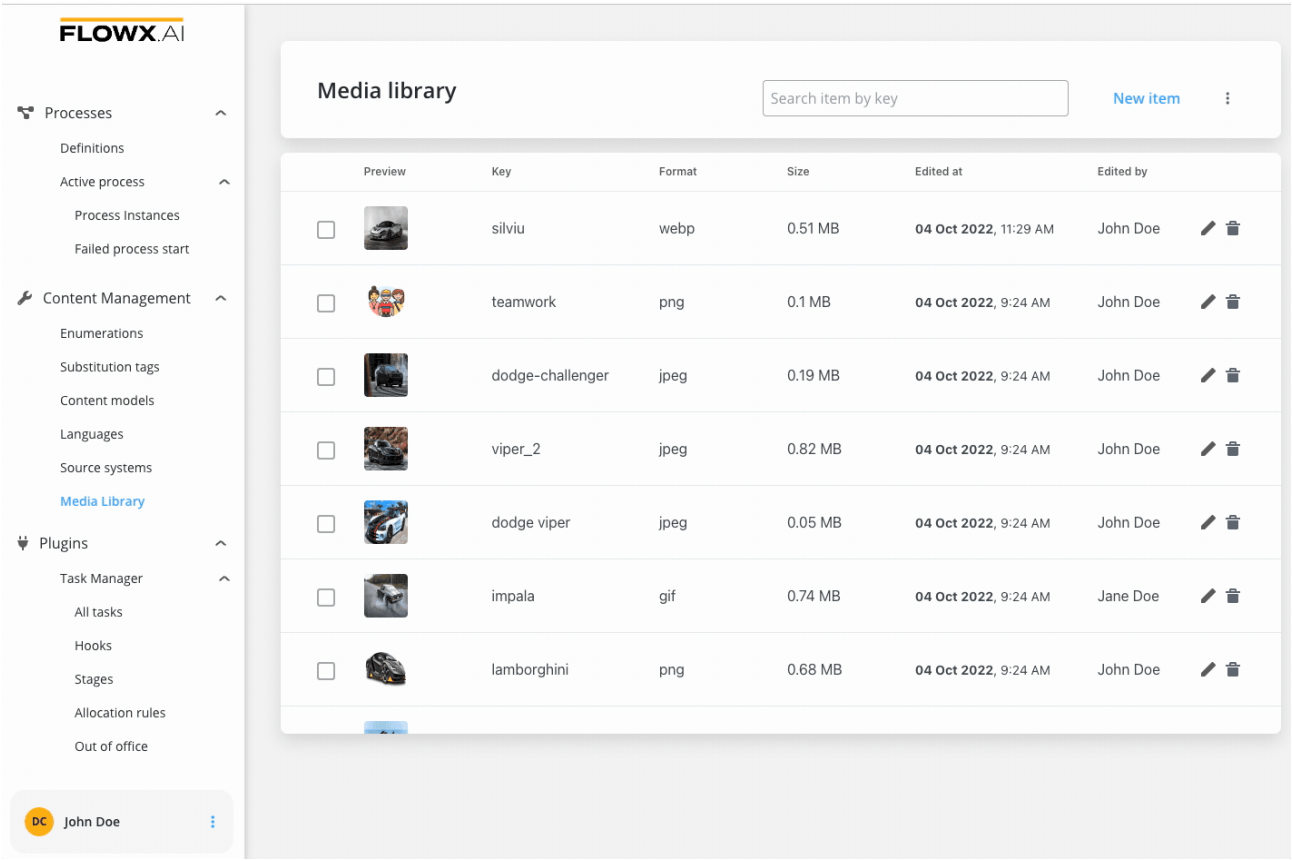


Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Content management / Media library

The media library serves as a centralized hub for managing and organizing various types of media files, including images, GIFs, and more. It encompasses all the files that have been uploaded to the

The fallback content to display on prerendering
, providing a convenient location to view, organize, and upload new media files.



INFO

You can also upload an image directly to the Media Library on the spot when configuring a process using the **UI Designer**. More information **here**.

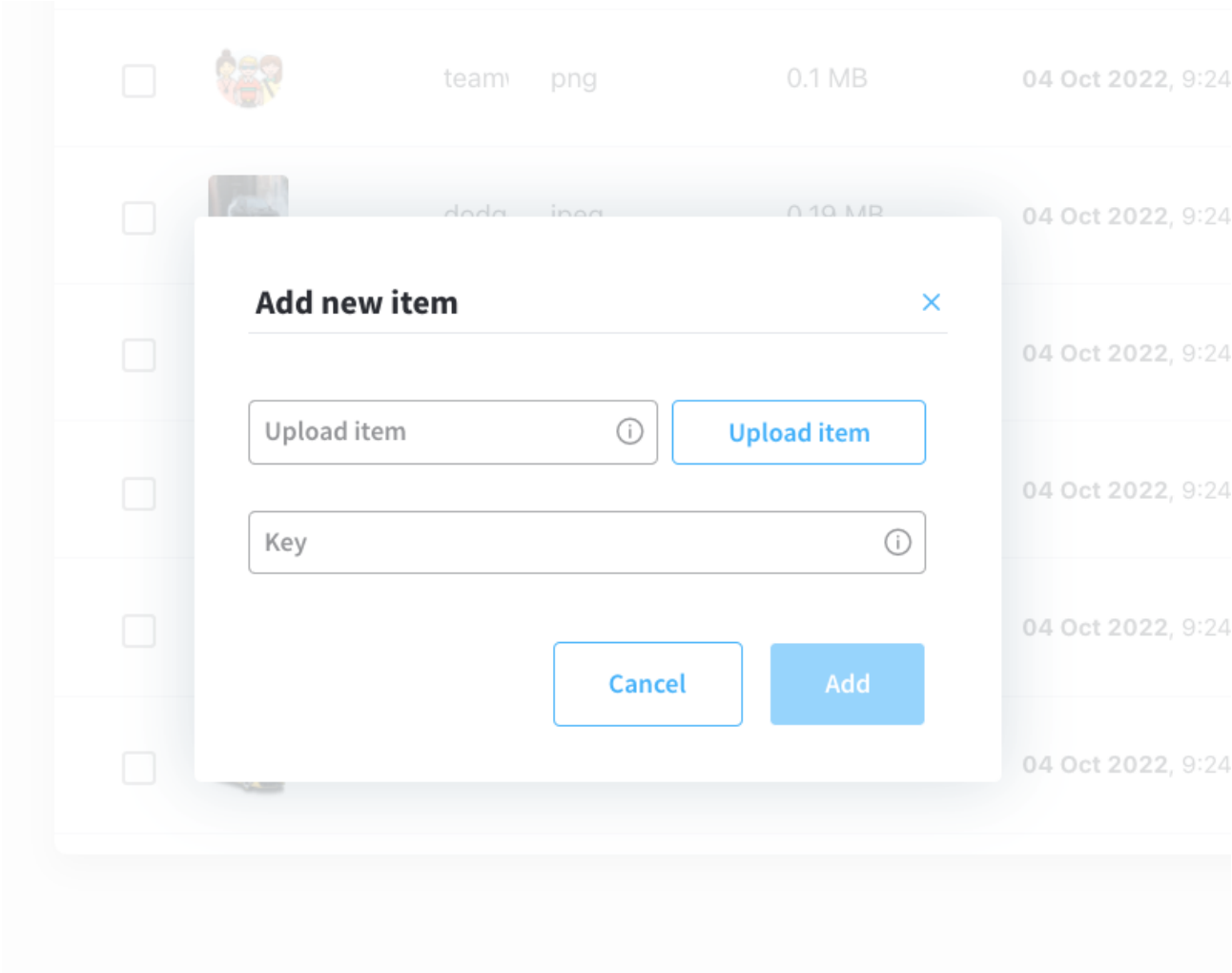
Uploading a new asset

To upload an asset to the Media Library, follow the next steps:

1. Open

   The fallback content to display on prerendering

   .
2. Go to **Content Management** tab and select **Media Library**.
3. Click **Add new item**, the following details will be displayed:
   - **Upload item** - opens a local file browser
   - **Key** - the key must be unique, you cannot change it afterwards



4. Click **Upload item** button and select a file from your local browser.
5. Click **Upload item** button again to upload the asset.

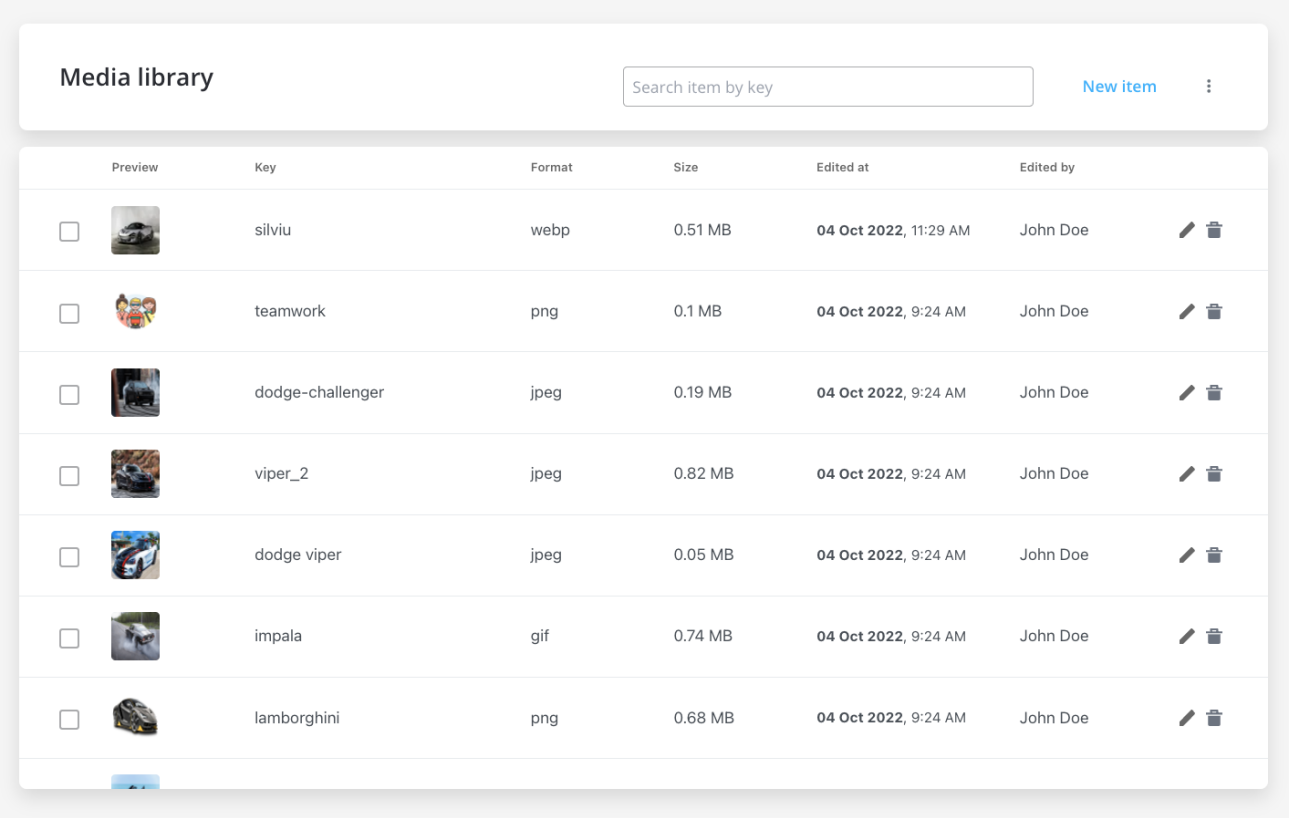
> **CAUTION**
>
> Supported formats: PNG, JPEG, JPG, GIF, SVG or WebP format, 1 MB maximum size.

Displaying assets

Users can preview all the uploaded assets just be accessing the **Media Library**.

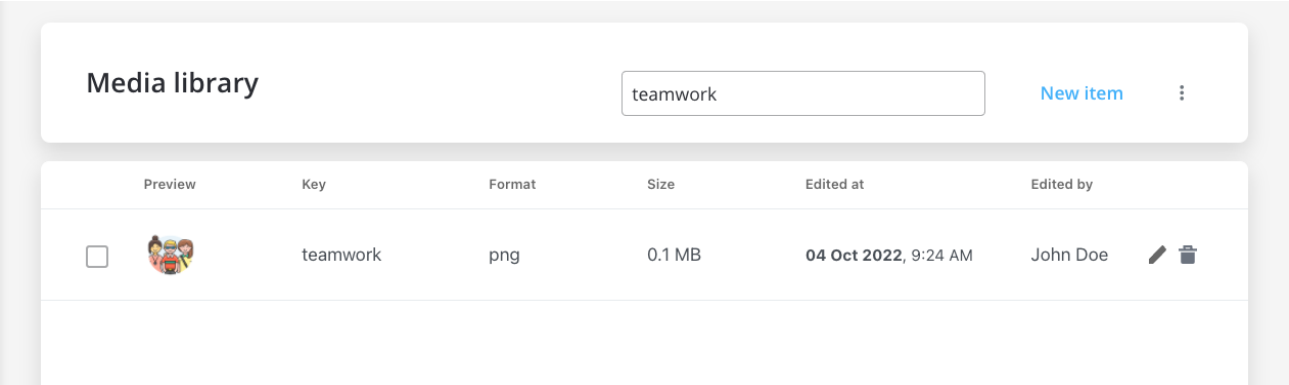
You have the following information about assets:

- Preview (thumbnail 48x48)
- Key
- Format ("-" for unknown format)
- Size
- Edited at
- Edited by



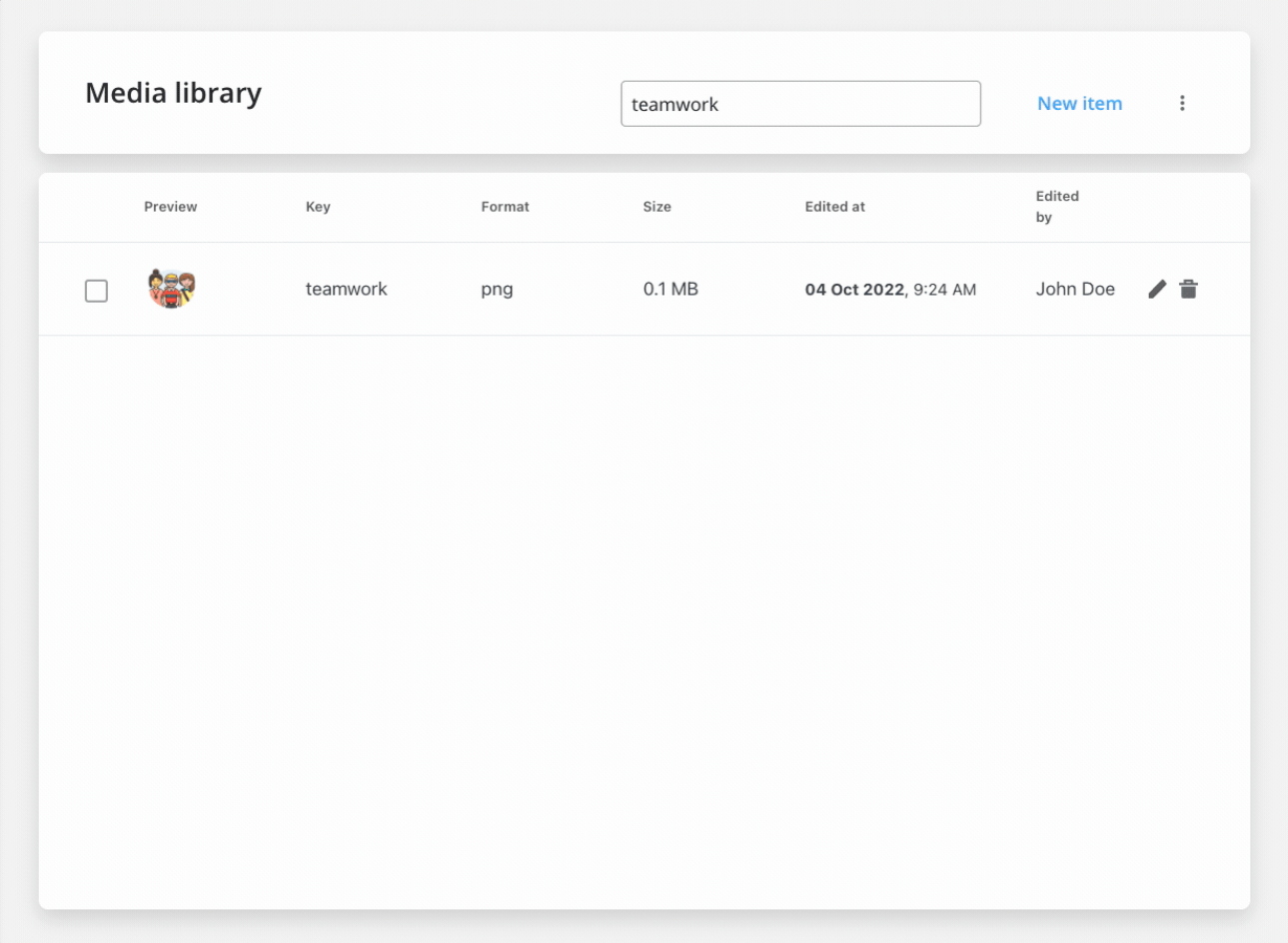
Searching assets

You can search an asset by using its key (full or substring).



Replacing assets

You can replace an item on a specific key (this will not break references to process definitions).

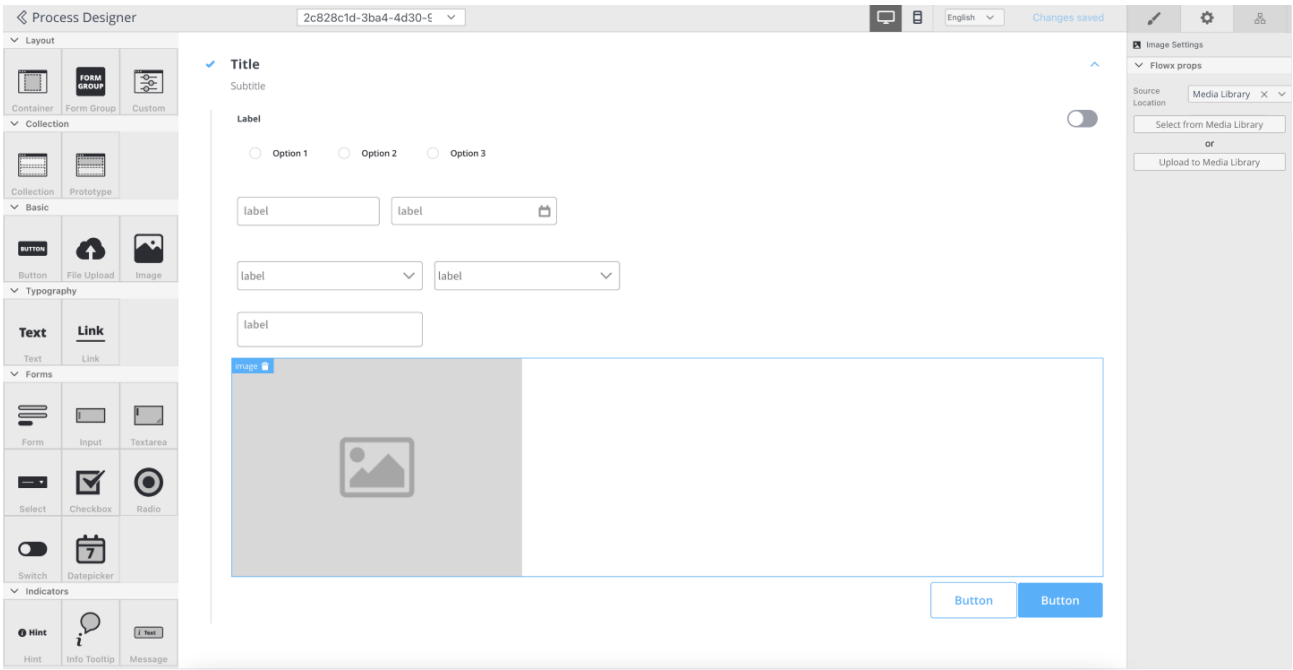


Referencing assets in UI Designer

You have the following options when configuring image components using UI Designer:

- Source Location - here you must select **Media Library** as source location
- Image Key

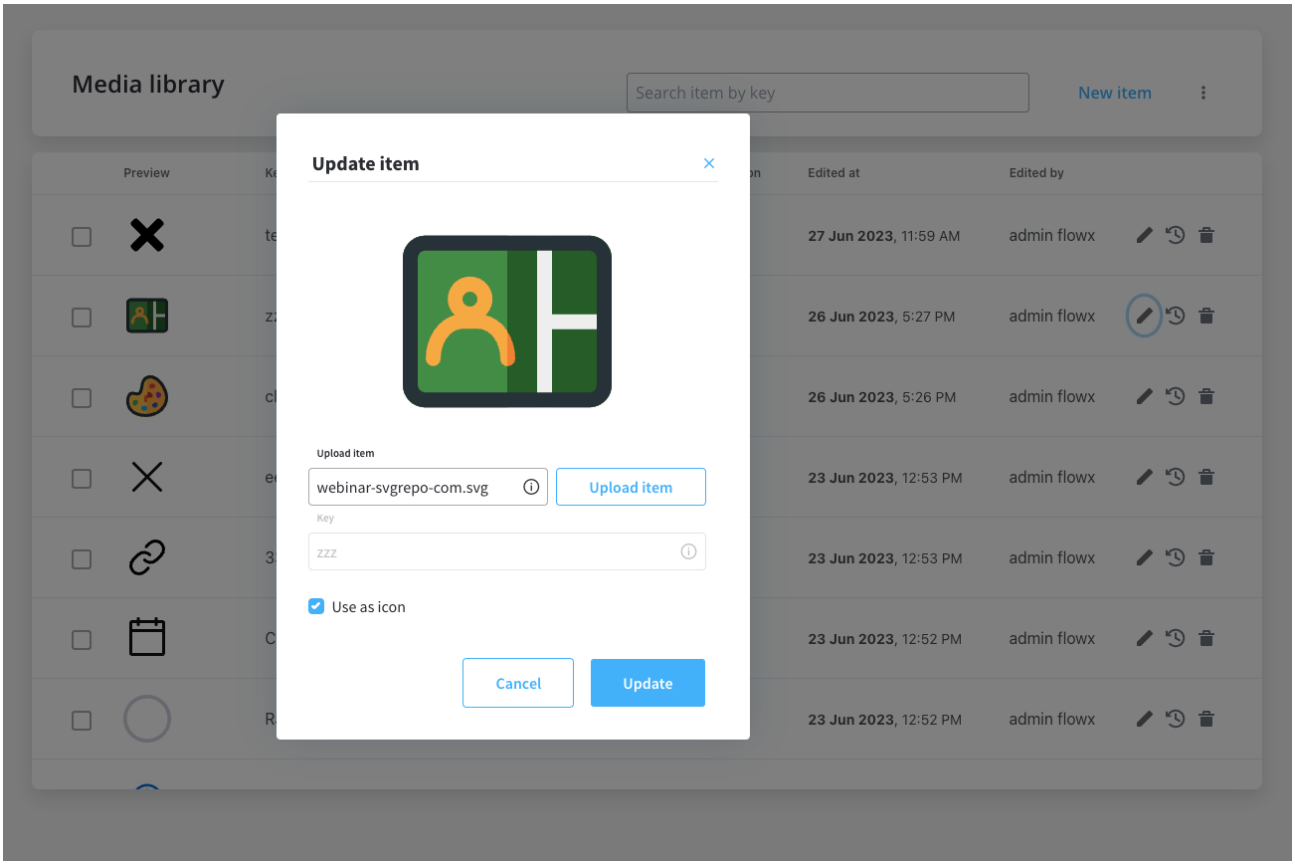
- **Option 1**: trigger a dropdown with images keys - you can type and filter options or can select from the initial list in dropdown
- **Option 2**: open a popup with images thumbnails and keys then you can type and filter options or can select from the initial list



> **INFO**
>
> More details on how to configure an image component using UI Designer - **here**.

Icons

The Icons feature allows you to personalize the icons used in UI elements. By uploading SVG files through the Media Library and marking them, you can choose icons from the available list in the UI Designer.



INFO

When selecting icons in the UI Designer, only SVG files marked as icons in the Media Library will be displayed.

INFO

To ensure optimal visual rendering and alignment within your UI elements, it is recommended to use icons with small sizes such as: 16px, 24px, 32px.

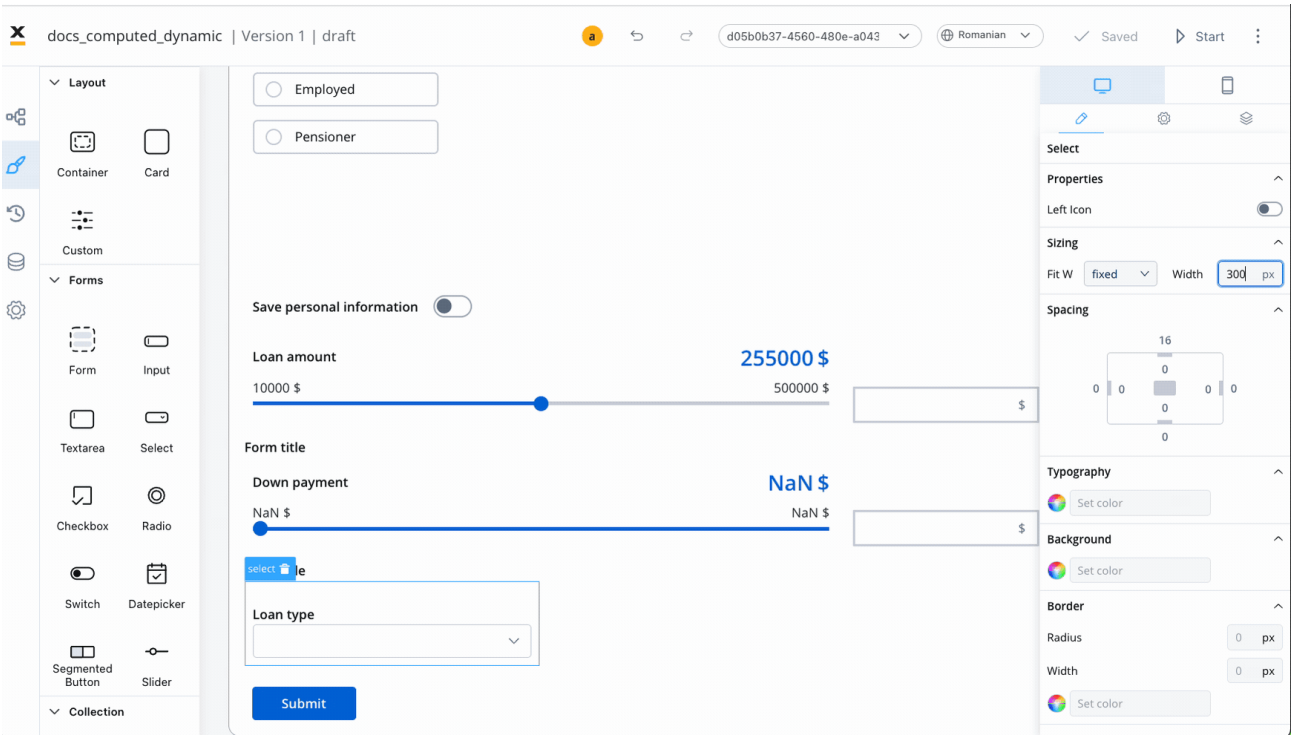
Using icons specifically designed for these sizes helps maintain consistency and ensures a visually pleasing user interface. It is advisable to select icons

from icon sets that provide these size options or to resize icons proportionally to fit within these dimensions.

> **CAUTION**
>
> Icons are displayed or rendered at their original, inherent size.

Customization

Content-specific icons pertain to the content of UI elements, such as icons for input fields or send message buttons. These icons are readily accessible in the UI Designer.



More details on how to add icons on each element, check the sections below:

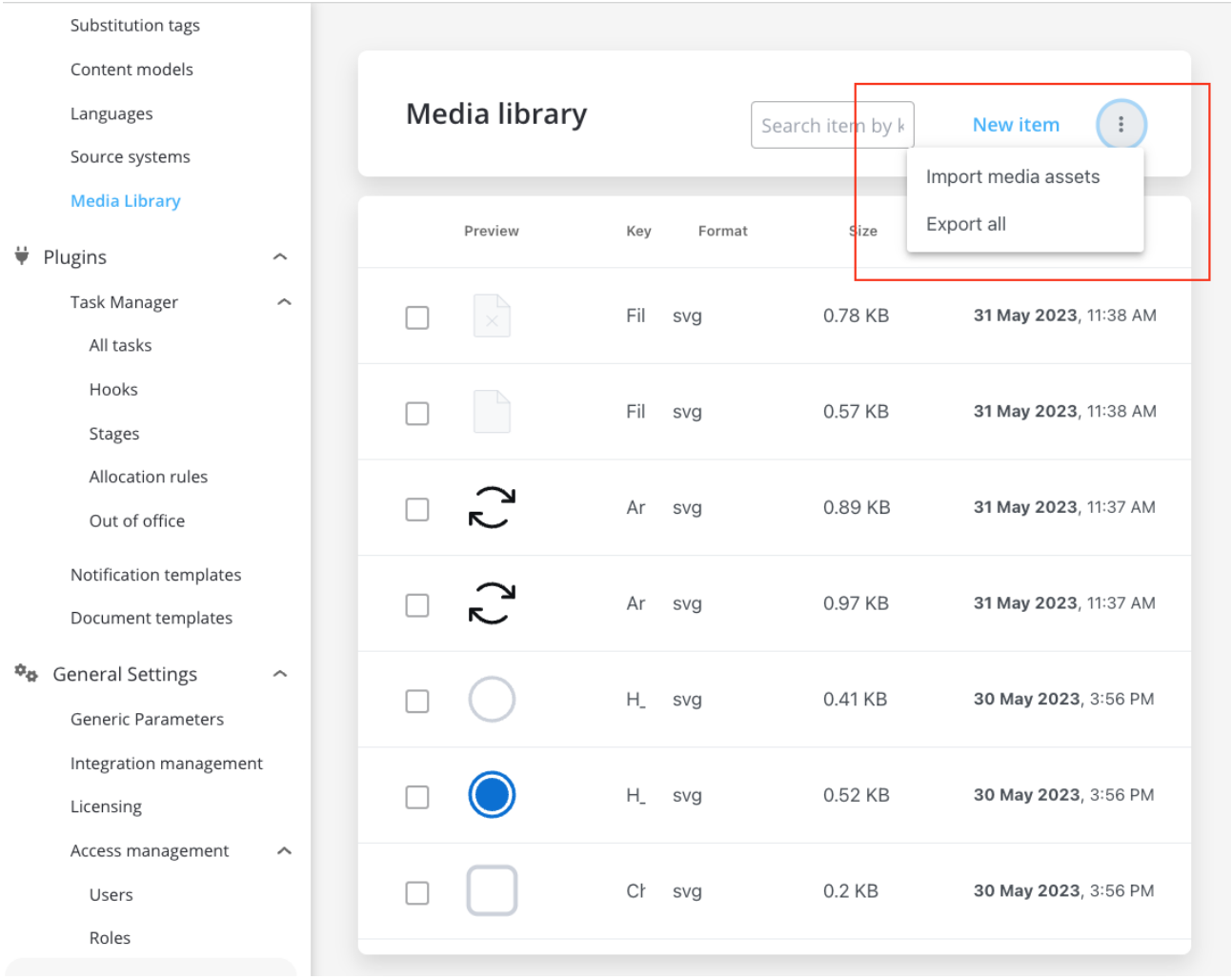
» Input element

» Select element

» Buttons

Export/import media assets

The import/export feature allows you to import or export media assets, enabling easy transfer and management of supported types of media files.



Import media assets

Use this function to import media assets of various supported types. It provides a convenient way to bring in images, videos, or other media resources.

Export all

Use this function to export all media assets stored in your application or system. The exported data will be in JSON format, allowing for easy sharing, backup, or migration of the media assets.

The exported JSON structure will resemble the following example:


```
{
  "images": [
    {
      "key": "cart",
      "application": "flowx",
      "filename": "maxresdefault.jpg",
      "format": "jpeg",
      "contentType": "image/jpeg",
      "size": 39593,
      "storagePath":
"https://d22tnnndi9lo60.cloudfront.net/devmain/flowx/cart/1681
      "thumbnailStoragePath":
"https://d22tnnndi9lo60.cloudfront.net/devmain/flowx/cart/1681
    },
    {
      "key": "pizza",
      "application": "flowx",
      "filename": "pizza.jpeg",
      "format": "jpeg",
      "contentType": "image/jpeg",
      "size": 22845,
      "storagePath":
"https://d22tnnndi9lo60.cloudfront.net/devmain/flowx/pizza/168
      "thumbnailStoragePath":
"https://d22tnnndi9lo60.cloudfront.net/devmain/flowx/pizza/168
    }
  ],
  "exportVersion": 1
}
```

- `images`- is an array that contains multiple objects, each representing an image
- `exportVersion` - represents the version number of the exported data, it holds the image-related information
- `key`- represents a unique identifier or name for the image, it helps identify and differentiate images within the context of the application
- `application` - specifies the name or identifier of the application associated with the image, it indicates which application or system the image is related to
- `filename` - the name of the file for the image, it represents the original filename of the image file
- `format` - a string property that specifies the format or file extension of the image
- `contentType` - the MIME type or content type of the image, it specifies the type of data contained within the image file
- `size` - represents the size of the image file in bytes, it indicates the file's storage size on a disk or in a data storage system
- `storagePath` - the URL or path to the location where the original image file is stored, it points to the location from where the image can be accessed or retrieved
- `thumbnailStoragePath` - the URL or path to the location where a thumbnail version of the image is stored, it points to the location from where the thumbnail image can be accessed or retrieved

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Generic parameters

Generic parameters are variables or settings that are used to control the behavior of a software application or system. These parameters are designed to be flexible and adaptable, allowing users to customize the software to their specific needs.

Through the

The fallback content to display on prerendering
, you can create, edit, import, or export these generic parameters. You can also assign the relevant environment(s) to these parameters, ensuring they are applied exactly where they are needed.

Why do you need generic parameters?

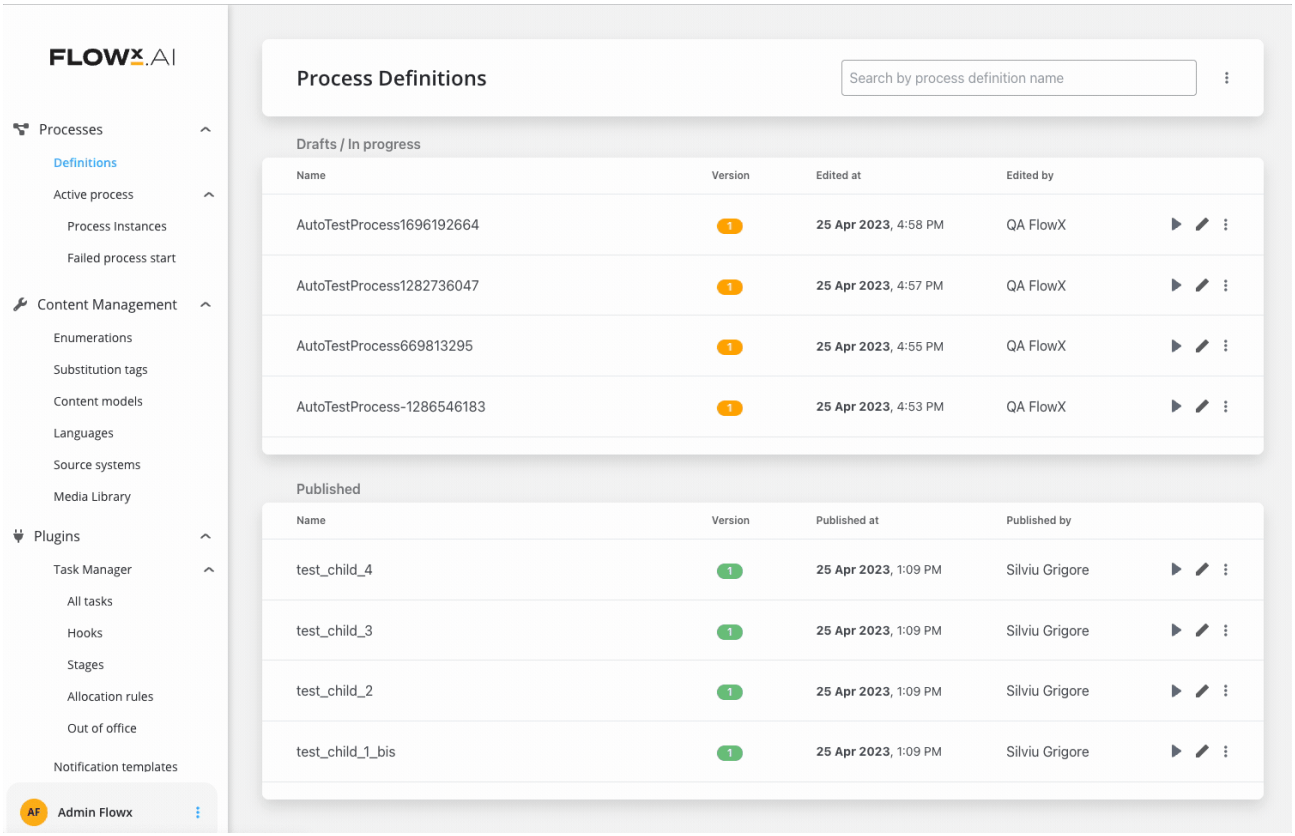
Generic parameters can be defined and used in many scenarios. Here are a few examples of useful generic parameters:

Parameter	Description
baseURL	This parameter can be used to define the base URL of an API or website, which can be utilized across multiple environments

Parameter	Description
redirectURL	This parameter can be used to define the URL to which a user should be redirected after completing a certain action or process. This can save time and effort by avoiding the need to hardcode multiple redirect URLs.
envFilePath	This parameter can be used to define the path of the environment file that stores a document uploaded

To add a new generic parameter, follow the next steps:

1. Go to **FLOWX Designer** and select the **General Settings** tab.
2. Select **Generic Parameters** from the list.
3. Click **New parameter**.
4. Fill in the details.
5. Click **Save**.

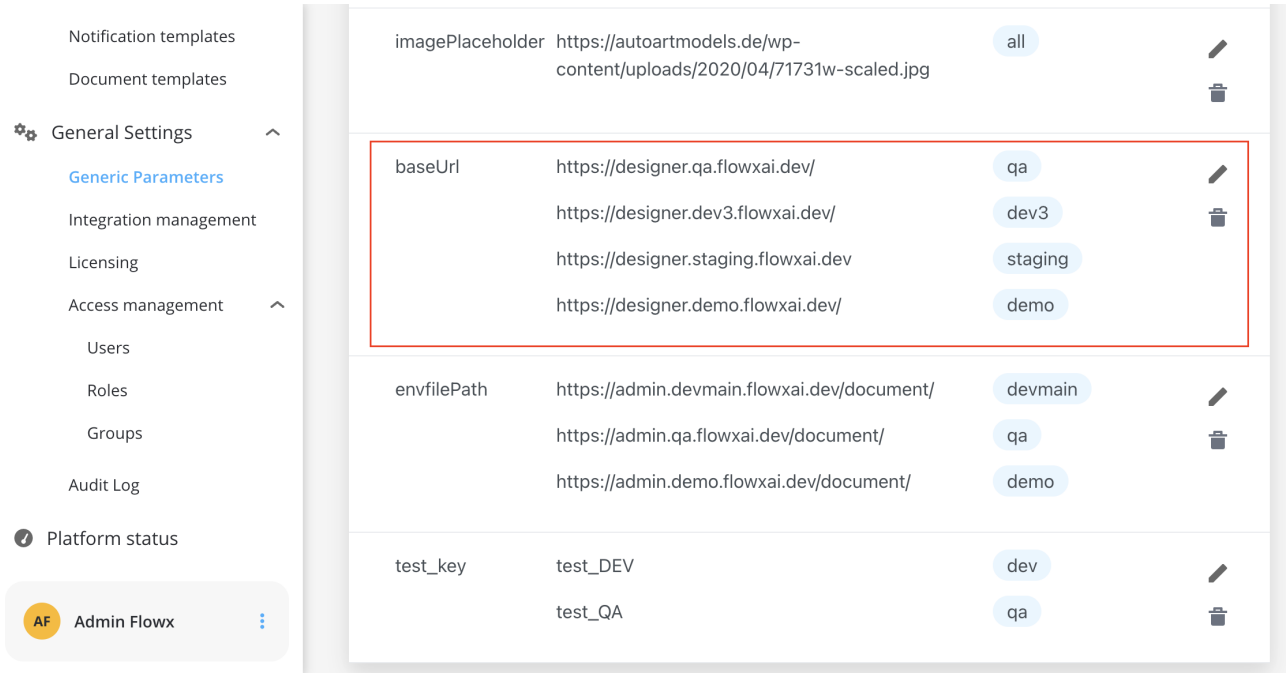


Configuring a generic parameter

To configure a generic parameter you need to fill in the following details:

- **Key** - the key that will be used in a process to call the generic parameter
- **Value** - the value that will replace the key depending on the defined parameters
- **Environment** - set the environment where you want to use the generic parameter (❗ leave empty to apply to all environments)
- **Add a new value** - to add a new value for the same key but for a different environment

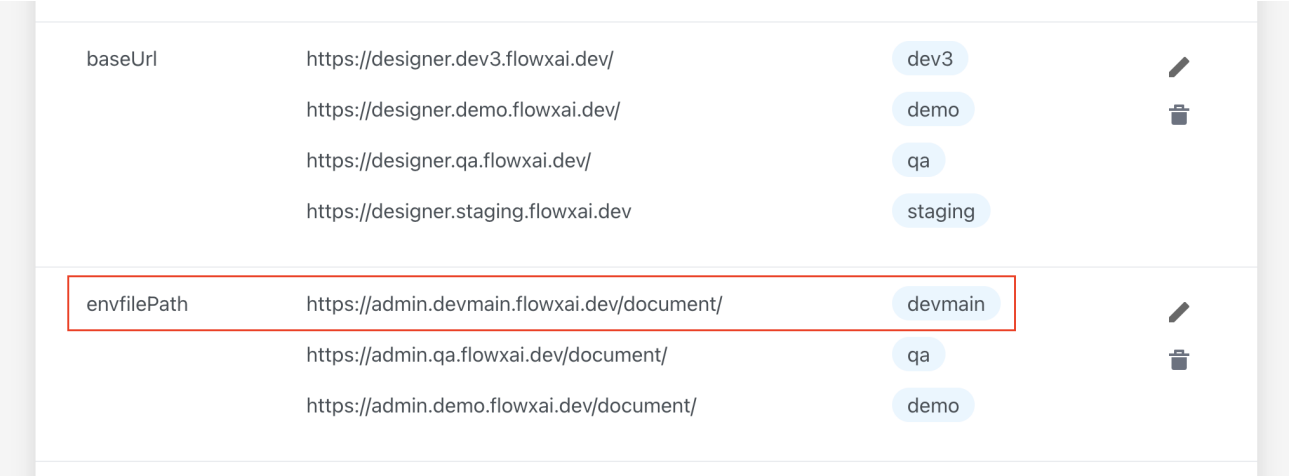
> **INFO**
>
> For example, if you want to set a `baseURL` generic parameter (the URL will be different, depending on the environmaent).



Using generic parameters

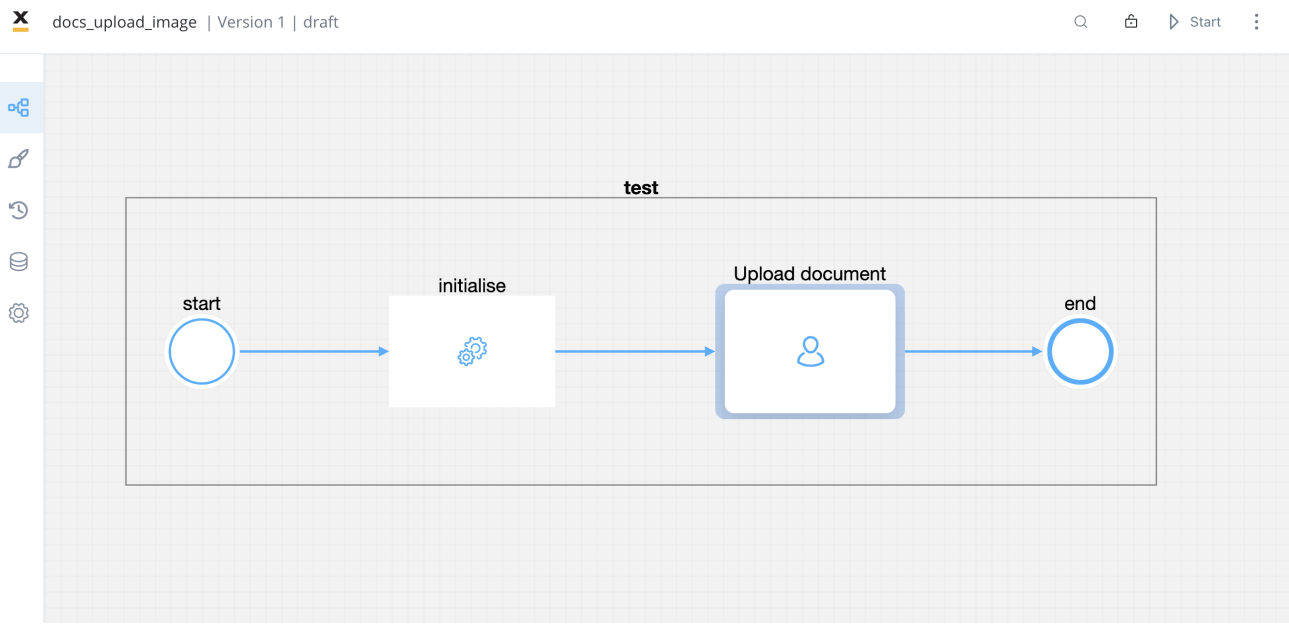
Use case

Imagine that you need to create a process in which you need to upload an image or a document. We will define a generic parameter called `envfilePath` that will represent the path where the document/image will be uploaded.



The minimum requirement to build an upload doc/image process:

- a start node
- a task node
- a user task node
- an end node



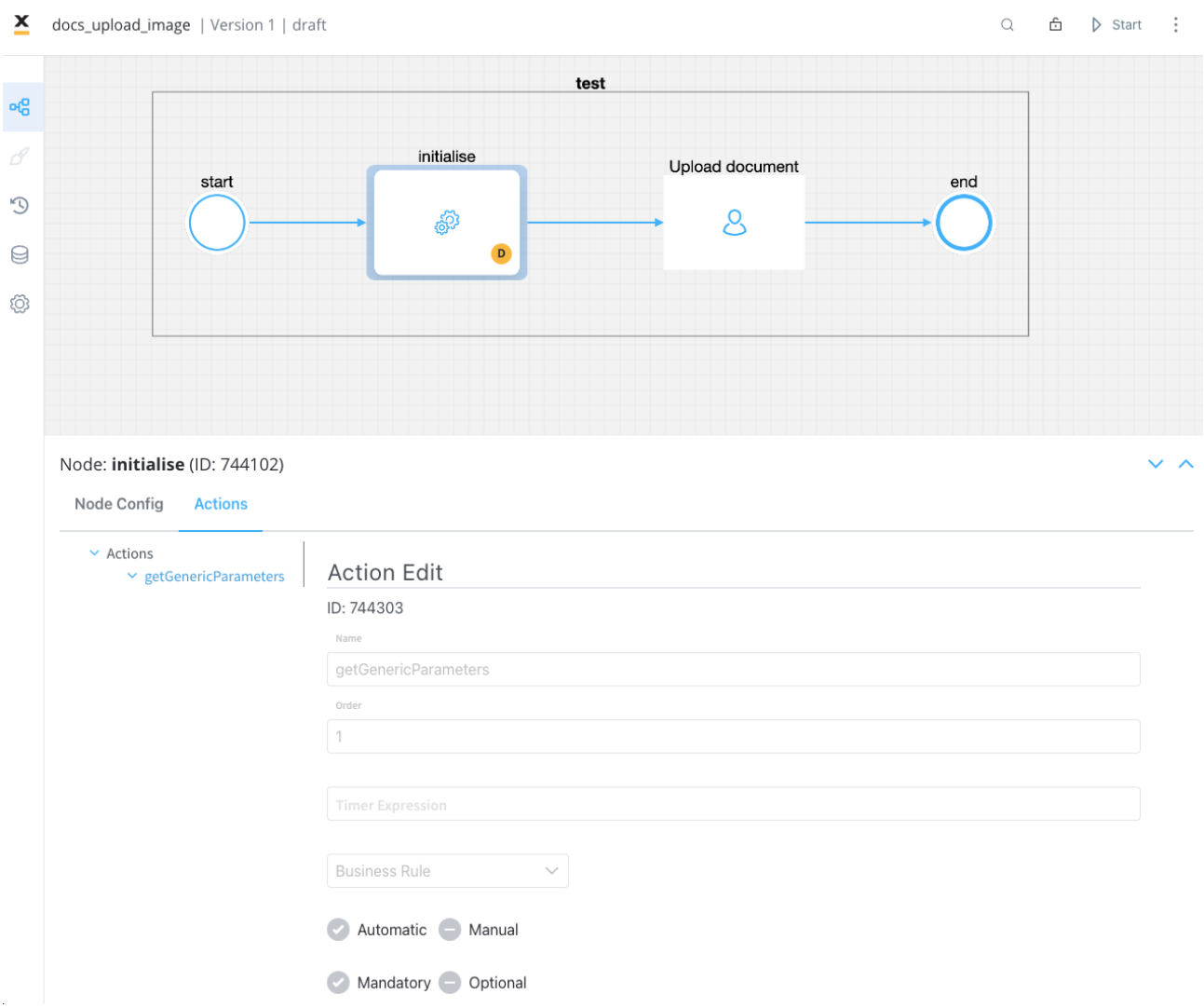
Configuring the task node

Set a

The fallback content to display on prerendering
action on the task node with the following properties:

Action Edit

- **Name** - used internally to make a distinction between different actions on nodes in the process - example *getGenericParameters*
- **Order** - if multiple actions are defined on the same node, the running order should be set using this option
- **Action type** - should be set to **Business Rule**
- **Trigger type** - Automatic - choose if this action should be triggered automatically (when the process flow reaches this step)
- **Required type** - automatic actions can only be defined as mandatory



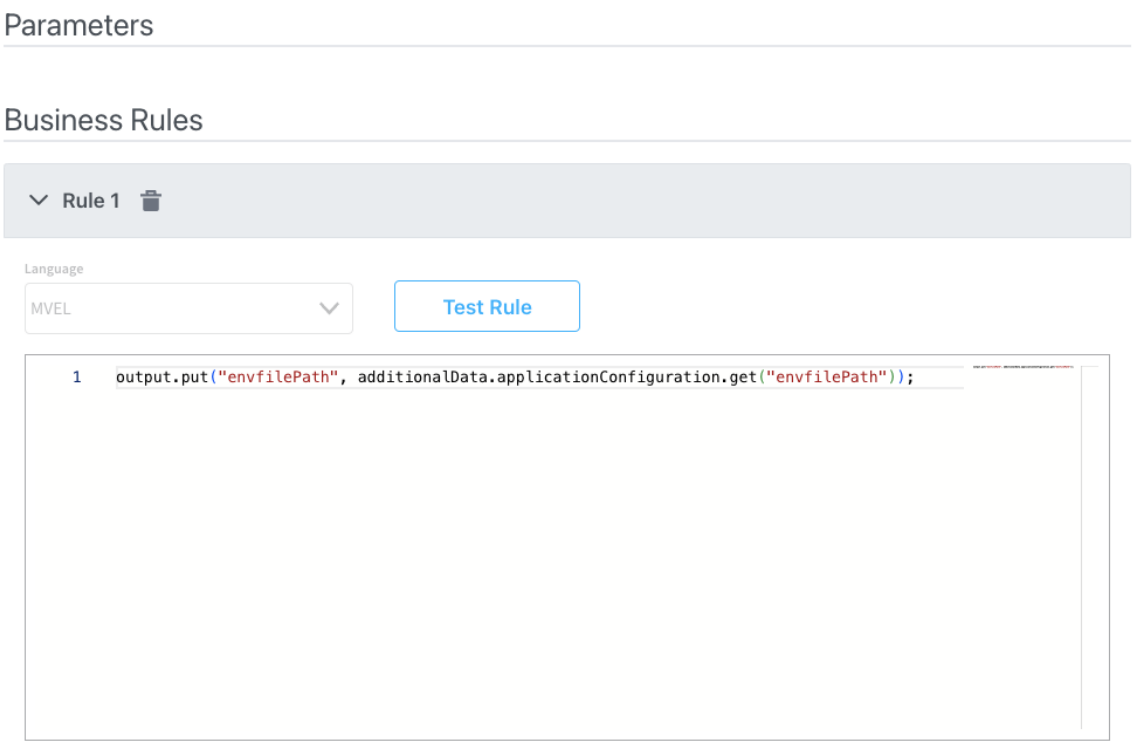
Parameters

- **Language** - we will choose **MVEL** for this business rule example

» MVEL details here

- **Message body** - MVEL expression

```
output.put("envfilePath",
additionalData.applicationConfiguration.get("envfilePath"));
```



This MVEL business rule assigns a value to a key, `envFilePath` (our defined generic parameter) in the "output" map object. The value assigned to the key is retrieved from another object, `additionalData.applicationConfiguration`, using the "get" method and passing the key `envFilePath` as the parameter.

In other words, this rule extracts the value of the `envFilePath` generic parameter from the `additionalData.applicationConfiguration` object and assigns it to the `envFilePath` key in the "output" map object.

It is important to note that the `additionalData.applicationConfiguration` object and the "output" map object must be previously defined and accessible in the current context for this rule to work.

Configuring the user task node

On this node we will define the following:

- an Upload File action with two child actions:
  - a Business Rule
  - a Send data to user interface action

> **INFO**
>
> Child actions can be marked as callbacks to be run after a reply from an external system is received. They will need to be set when defining the interaction with the external system (the Kafka send action).

For example, a callback function might be used to handle a user's interaction with a web page, such as upload a file. When the user performs the action, the callback function is executed, allowing the web application to respond appropriately.

Configuring Upload file action

Set an Upload file action on the task node with the following properties:

Action Edit

- **Name** - *uploadDocument*

- **Order** - if multiple actions are defined on the same node, the running order should be set using this option
- **Action type** - should be set to **Upload File**
- **Trigger type** - manually (triggered by the user)
- **Required type** - optional
- Repeatable - yes - should be checked if the action can be triggered multiple times
- **Autorun Children** - when this is switched on, the child actions (the ones defined as mandatory and automatic) will run immediately after the execution of the parent action is finalized

Parameters

- **Address** - the Kafka topic where the file will be posted - `ai.flowx.in.devmain.document.persist.v1`

> **CAUTION**
>
> In this example we used an environment called `devmain`, topic naming convention is different depending on what environment you are working.

- **Document Type** - other metadata that can be set (useful for the document plugin) - example: `BULK`
- **Folder** - allows you to configure a value by which the file will be identified in the future - example: `1234_${processInstanceId}`
- **Advanced configuration (Show headers)** - this represents a JSON value that will be sent on the headers of the Kafka message

```
{"processInstanceId": ${processInstanceId}, "destinationId":
"Upload document", "callbacksForAction": "uploadDocument"}
```

- `callbacksForAction` - the value of this key is a string that specifies a callback action associated with the `Upload document` destination ID (node). This is part of an event-driven system (Kafka send action) where this callback will be called once the `uploadDocument` action is completed.

Configuring Business rule action

```
envfilePath = input.?envfilePath;
uploadedDocument = input.?uploadedDocument;
if(uploadedDocument.?downloadPath != null &&
uploadedDocument.?downloadPath != ""){
    filePath = envfilePath + uploadedDocument.?downloadPath;
    uploadedDocument.filePath = filePath;

    output.put("uploadedDocument", uploadedDocument);
}
```

1. The business rule is expecting two inputs: `envfilePath` and `uploadedDocument`.
2. It is checking if the `downloadPath` property of the `uploadedDocument` input is not null and not an empty string. If it's not, then it proceeds to the next steps.
3. It concatenates the `envfilePath` and the `downloadPath` to form the full file path (filePath) where the uploaded document is expected to be located.
4. It updates the `uploadedDocument` input by adding a new property called `filePath` with the value of `filePath`.

5. It puts the updated `uploadedDocument` object into the output object as a key-value pair, with the key being "uploadedDocument".

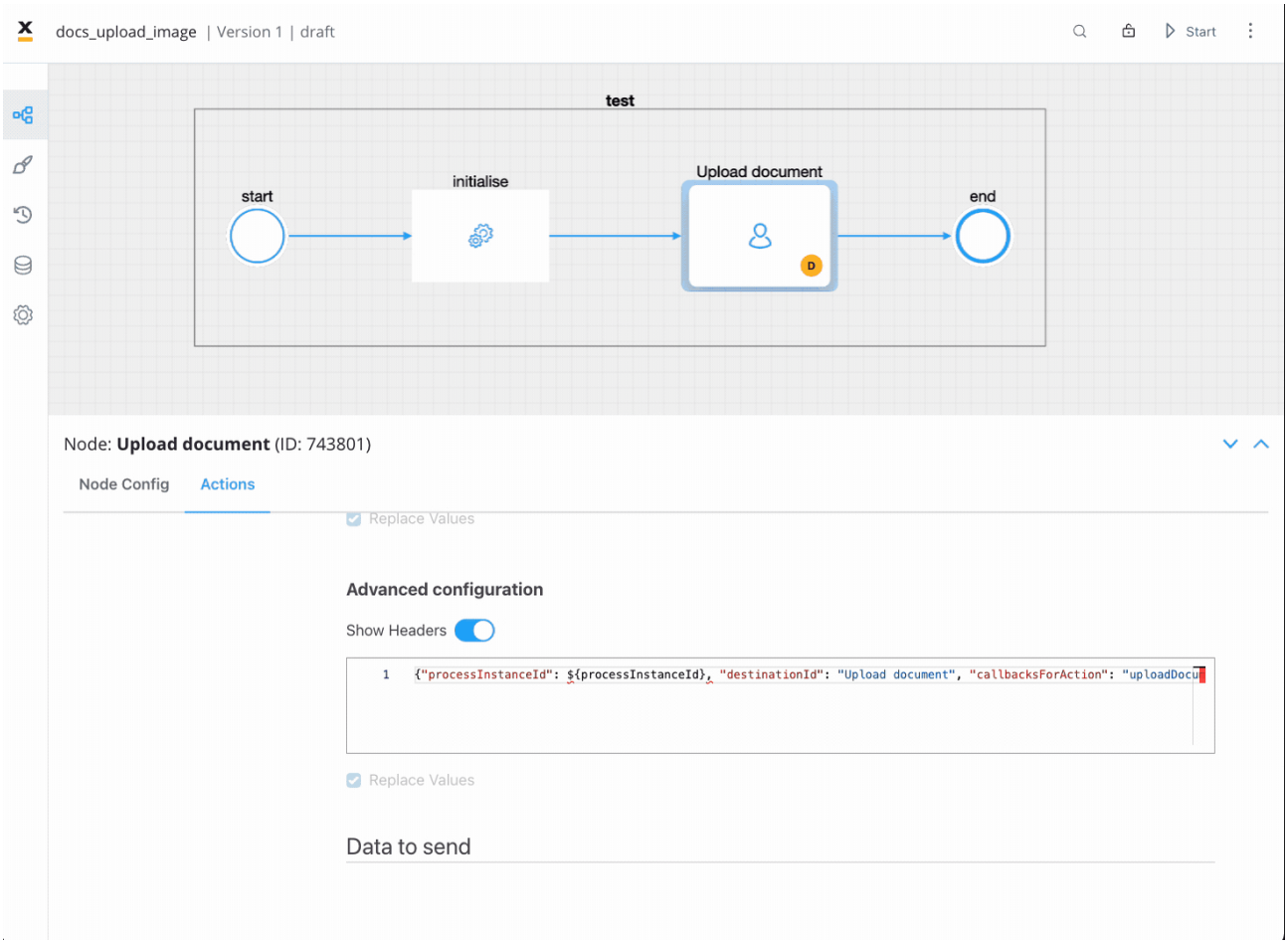
In summary, the code seems to be processing an uploaded document by checking its download path, constructing a full file path, and updating the document object with the new file path. Finally, it outputs the updated document object.

Configuring a Send data to user interface action

```
{"uploadedDocument":
    {
        "filePath": "${uploadedDocument.filePath}"
    }
}
```

"filePath": This is a key in the object which holds the value ${uploadedDocument.filePath}. The syntax ${...} suggests that it's a variable placeholder that will be replaced with the actual value at runtime.

After configuring all the nodes and parameters, run the process:



Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Integration management / Configuring access rights for Integration Management

Granular access rights can be configured for restricting access to the Integration Management plugin component. These access rights must be configured in the Designer (admin) deployment.

The following access authorizations are provided, with the specified access scopes:

1. **Manage-integrations** - for configuring access for managing integration management

Available scopes:

- import - users can import integrations
- read - users can view integrations
- edit - users can edit integrations
- admin - users can delete integrations

Integration management is preconfigured with the following default users roles for each of the access scopes mentioned above:

- manage-integrations
  - import:
    - ROLE_ADMIN_MANAGE_INTEGRATIONS_IMPORT
    - ROLE_ADMIN_MANAGE_INTEGRATIONS_EDIT
    - ROLE_ADMIN_MANAGE_INTEGRATIONS_ADMIN
  - read:
    - ROLE_ADMIN_MANAGE_INTEGRATIONS_READ
    - ROLE_ADMIN_MANAGE_INTEGRATIONS_IMPORT
    - ROLE_ADMIN_MANAGE_INTEGRATIONS_EDIT

        - ROLE_ADMIN_MANAGE_INTEGRATIONS_ADMIN
    - edit:
        - ROLE_ADMIN_MANAGE_INTEGRATIONS_EDIT
        - ROLE_ADMIN_MANAGE_INTEGRATIONS_ADMIN
    - admin:
        - ROLE_ADMIN_MANAGE_INTEGRATIONS_ADMIN

> **DANGER**
>
> These roles need to be defined in the chosen identity provider solution. It can be either kyecloak, RH-SSO, or another identity provider solution. For more details on how to define service accounts, check the Access rights section.

In case other custom roles are needed, you can configure them using environment variables. More than one role can be set for each access scope.

To configure access for each of the roles above, adapt the following input:

`SECURITY_ACCESSAUTHORIZATIONS_AUTHORIZATIONNAME_SCOPES_SCOPENAME_ROLESALLOWED: NEEDED_ROLE_NAMES`

Possible values for AUTHORIZATIONNAME: MANAGEDOCUMENTTEMPLATES.

Possible values for SCOPENAME: import, read, edit, admin.

For example, if you need to configure role access for read, insert this:

```
SECURITY_ACCESSAUTHORIZATIONS_MANAGEINTEGRATIONS_SCOPES_READ_R
ROLE_NAME_TEST
```

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Licensing

The License Engine is part of the core components of the

The fallback content to display on prerendering
. It is used for displaying reports regarding the usage of the platform in the
The fallback content to display on prerendering
.

It can be quickly deployed on the chosen infrastructure and then connected to the

The fallback content to display on prerendering
through
The fallback content to display on prerendering
events.

Let's go through the steps needed in order to deploy and set up the service:

» License engine setup guide

Multiple roles are available in the license engine, here are the steps for configuring them:

» Configuring access roles (old)

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Audit log

The Audit Log service provides a centralized location for all audit events. The following details are available for each event:

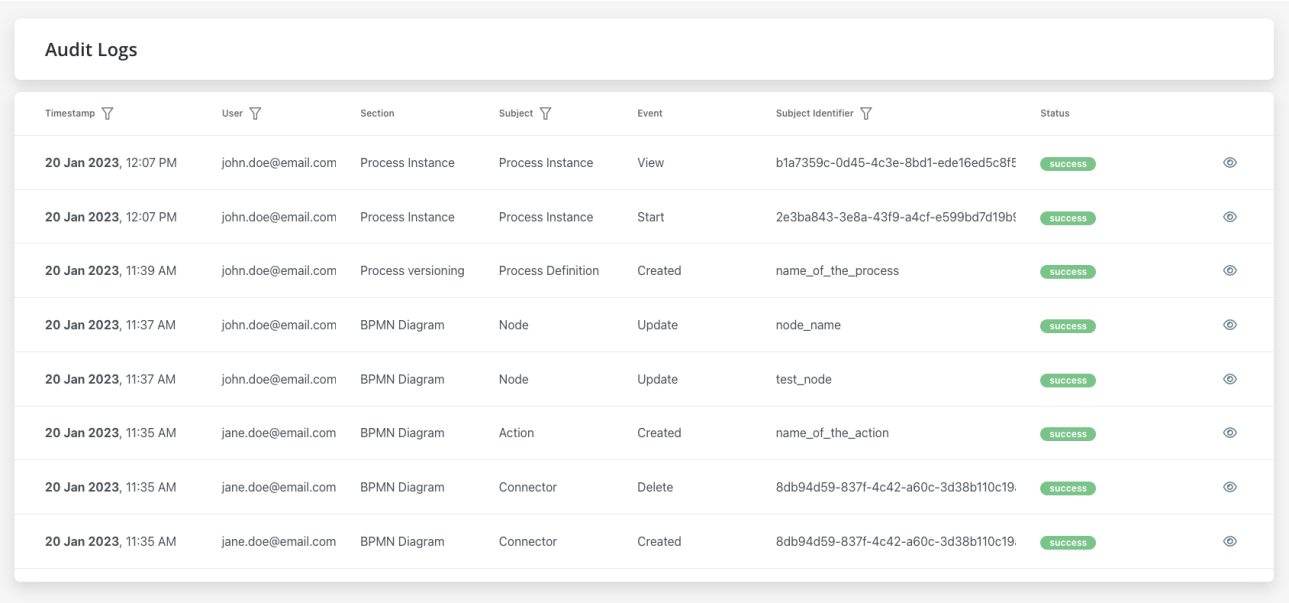
- **Timestamp** - the date and time the event occurred, the timestamp is displayed in a reversed chronologically order
- **User** - the entity who initiated the event, could be a username or a system
- **Subject** - the area or component of the system affected by the event

▸ Possible values

- **Event** - the specific action that occurred

▸ Possible values

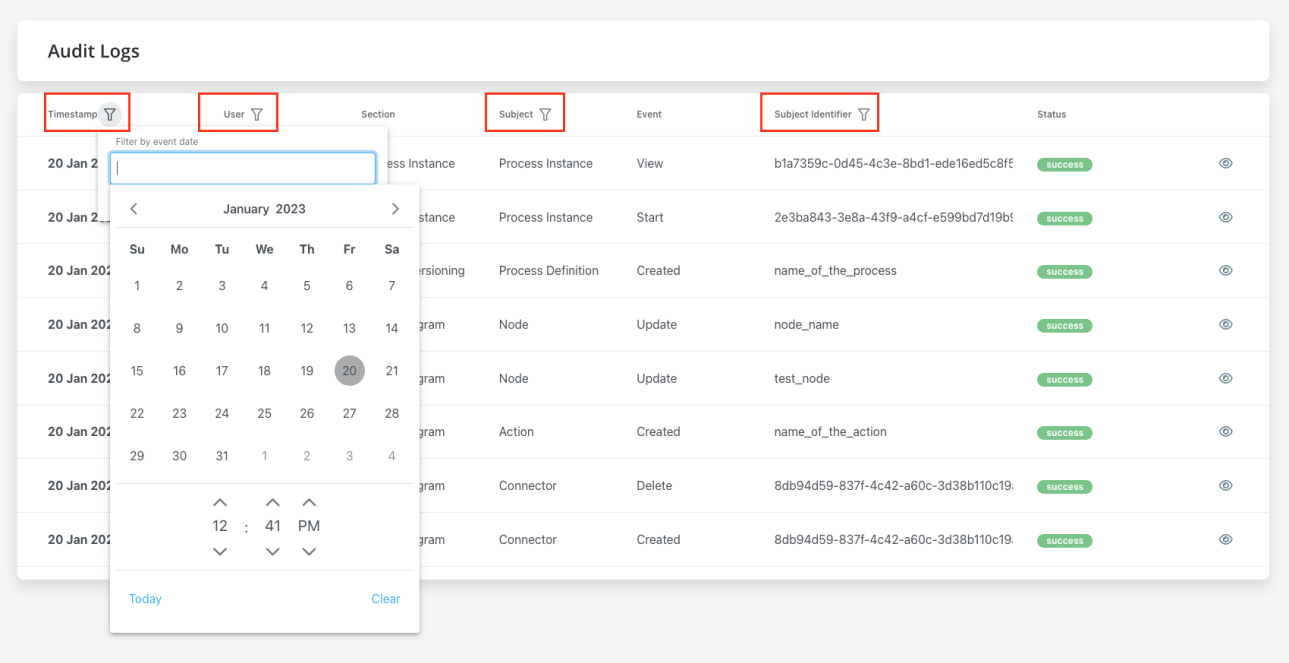
- **Subject identifier** - the name related to the subject, there are different types of identifiers based on the selected subject
- **Version** - the version of the process definition at the time of the event
- **Status** - the outcome of the event (e.g. success or failure)



Filtering

Users can filter audit records by event date and by selecting specific options for User, Subject, and Subject Identifier.

- Filter by event date

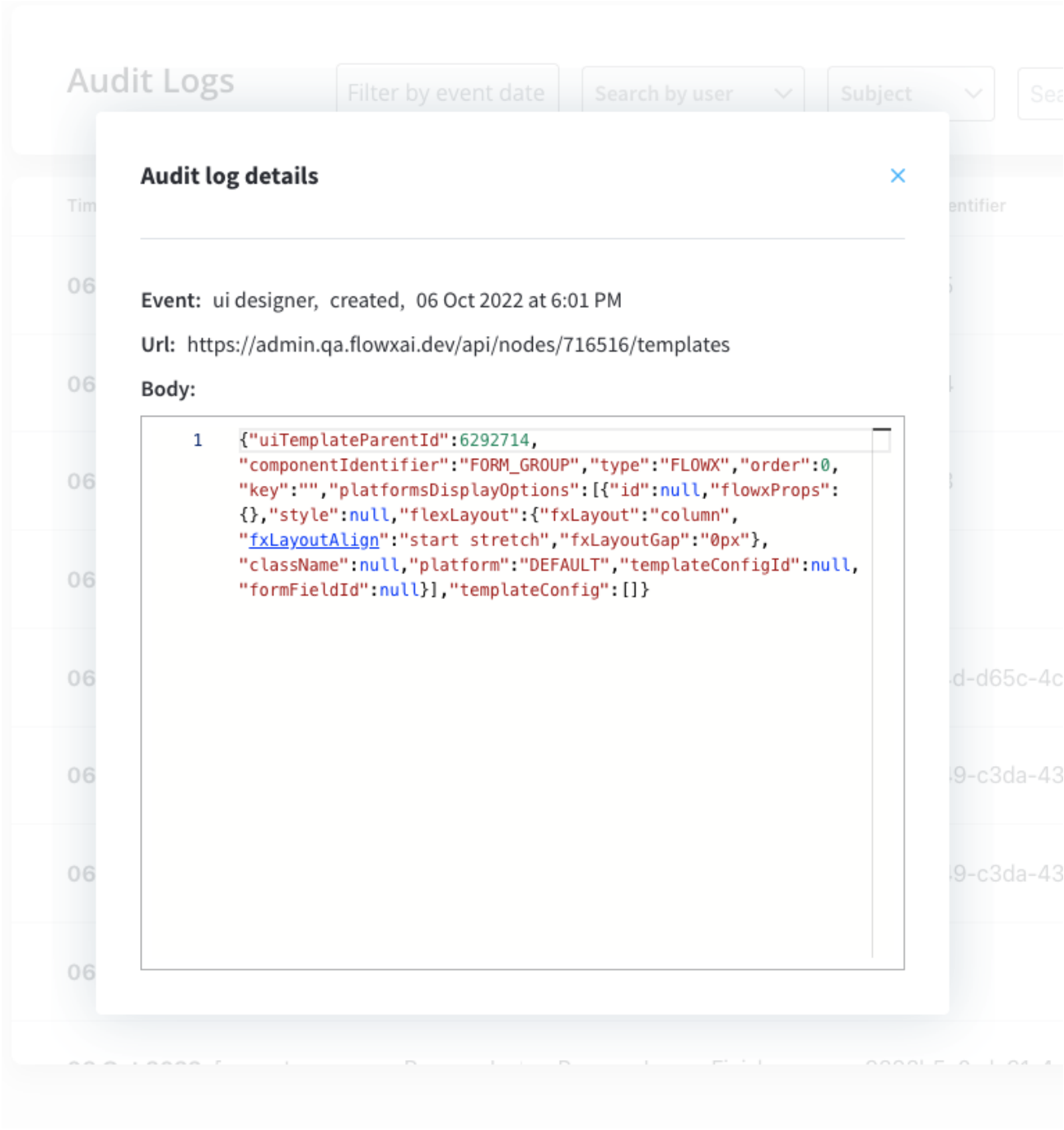


- User - single selection, type at least 4 characters
- Subject - single selection
- Subject identifier - exact match

Audit log details

To view additional details for a specific event, users can click the eye icon on the right of the event in the list. Additional information available in the audit log details window includes Here you have the following information:

- Event - the specific action that occured
- URL - the URL associated with the event
- Body - any additional data or information related to the event



Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Scheduler

Overview

The Scheduler is part of the core components of the

The fallback content to display on prerendering
. It can be easily added to your custom FLOWX deployment to **enhance the core platform capabilities with functionality specific to scheduling messages.**

The service offers the possibility to schedule a message that you only need to process after a configured time period.

It can be quickly deployed on the chosen infrastructure and then connected to the

The fallback content to display on prerendering
through Kafka events.

Let's go through the steps needed in order to deploy and set up the service:

» Scheduler setup guide

We've prepared some examples of various use cases where this service is useful:

Using the scheduler

After deploying the scheduler service in your infrastructure, you can start using it to schedule messages that you need to process at a later time.

One such example would be to use the scheduler service to expire processes that were started but haven't been finished.

> **DANGER**
>
> First you need to check the configured topics match the ones configured in the engine.

For example the engine topics KAFKA_TOPIC_PROCESS_SCHEDULE_OUT_SET and KAFKA_TOPIC_PROCESS_SCHEDULE_OUT_STOP **should be the same with the ones configured in the scheduler** (`KAFKA_TOPIC_SCHEDULE_IN_SET` and `KAFKA_TOPIC_SCHEDULE_IN_STOP`)

When a process is scheduled to expire, the engine sends the following message to the scheduler service (on the topic `KAFKA_TOPIC_SCHEDULE_IN_SET`):

```
{
  "applicationName": "onboarding",
  "applicationId": "04f82408-ee66-4c68-8162-b693b06bba00",
  "payload": {
    "scheduledEventType": "EXPIRE_PROCESS",
    "processInstanceUuid": "04f82408-ee66-4c68-8162-
b693b06bba00"
  },
  "scheduledTime": 1621412209.353327,
```

```
  "responseTopicName": "ai.flowx.process.expire.staging"
}
```

The scheduled time should be defined as `java.time.Instant`.

At the scheduled time, the payload will be sent back to the response topic defined in the message, like so:

```
{
  "scheduledEventType": "EXPIRE_PROCESS",
  "processInstanceUuid": "04f82408-ee66-4c68-8162-
b693b06bba00"
}
```

If you don't need the scheduled message anymore, you can discard it by sending the following message (on the topic `KAFKA_TOPIC_SCHEDULE_IN_STOP`)

```
{
  "applicationName": "onboarding",
  "applicationId": "04f82408-ee66-4c68-8162-b693b06bba00"
}
```

These fields, `applicationName` and `applicationId` are used to uniquely identify a scheduled message.

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Core extensions / Search data service

Search data is a microservice that searches for data in another process.

The new search data microservice enables you to create a process that can perform a search/look for data (using Kafka send / Kafka receive actions) in other processes.

> **TIP**
>
> Using elastic search, the new search microservice will be able to search for keys that are indexed in ES, via existing mechanics.

> **CAUTION**
>
> Elastic search indexing must be switched on the FLOWX.AI Engine configuration. You can find more details in the **Search data service setup guide**.

Using search data

Use case:

- search for data in other processes
- display results about other processes where the search key was found

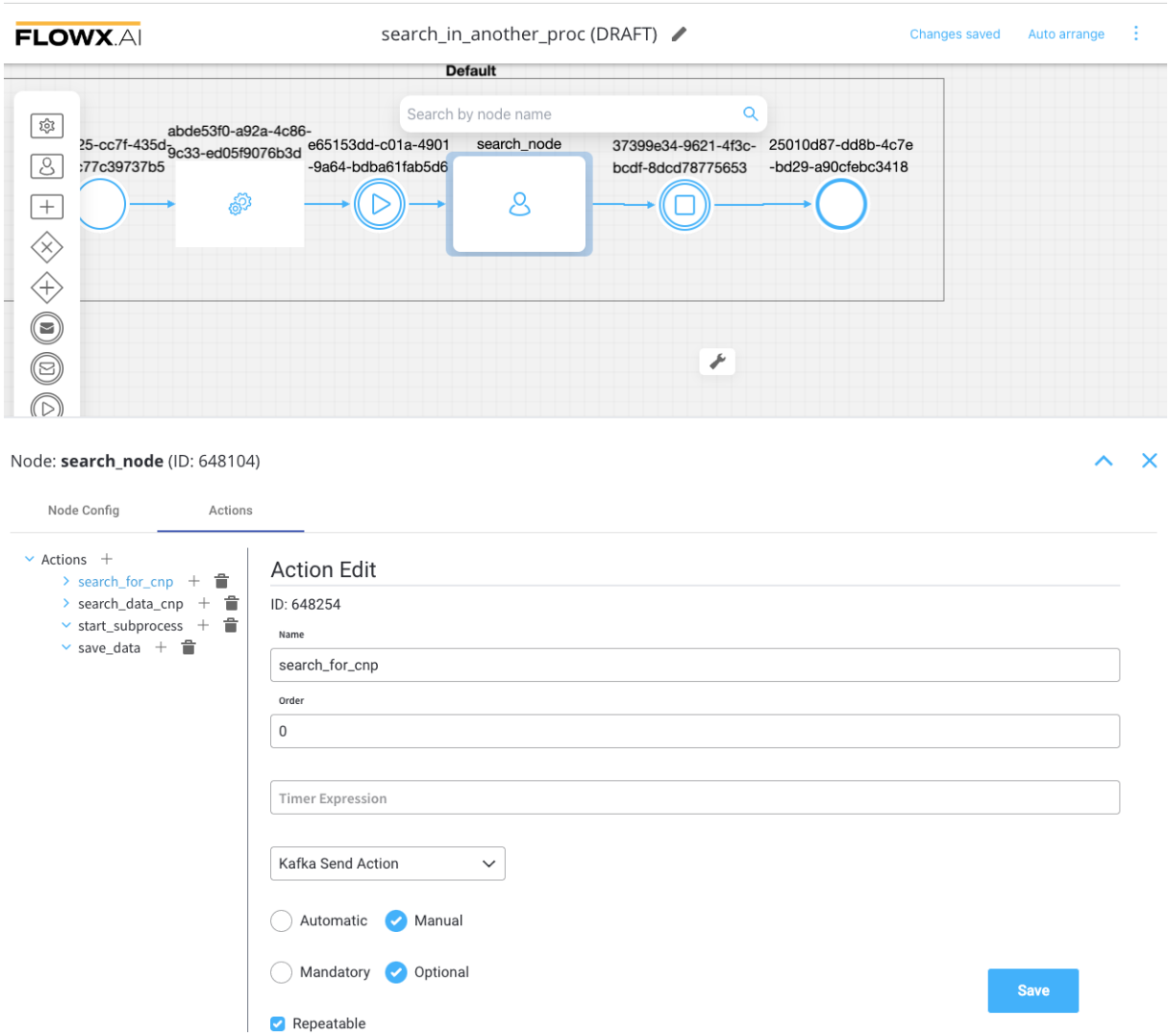
1. Create a process using

   The fallback content to display on prerendering

   .

2. From the newly created process where you want to perform the search, add a Task node.

3. Configure a send event via a Kafka send action.

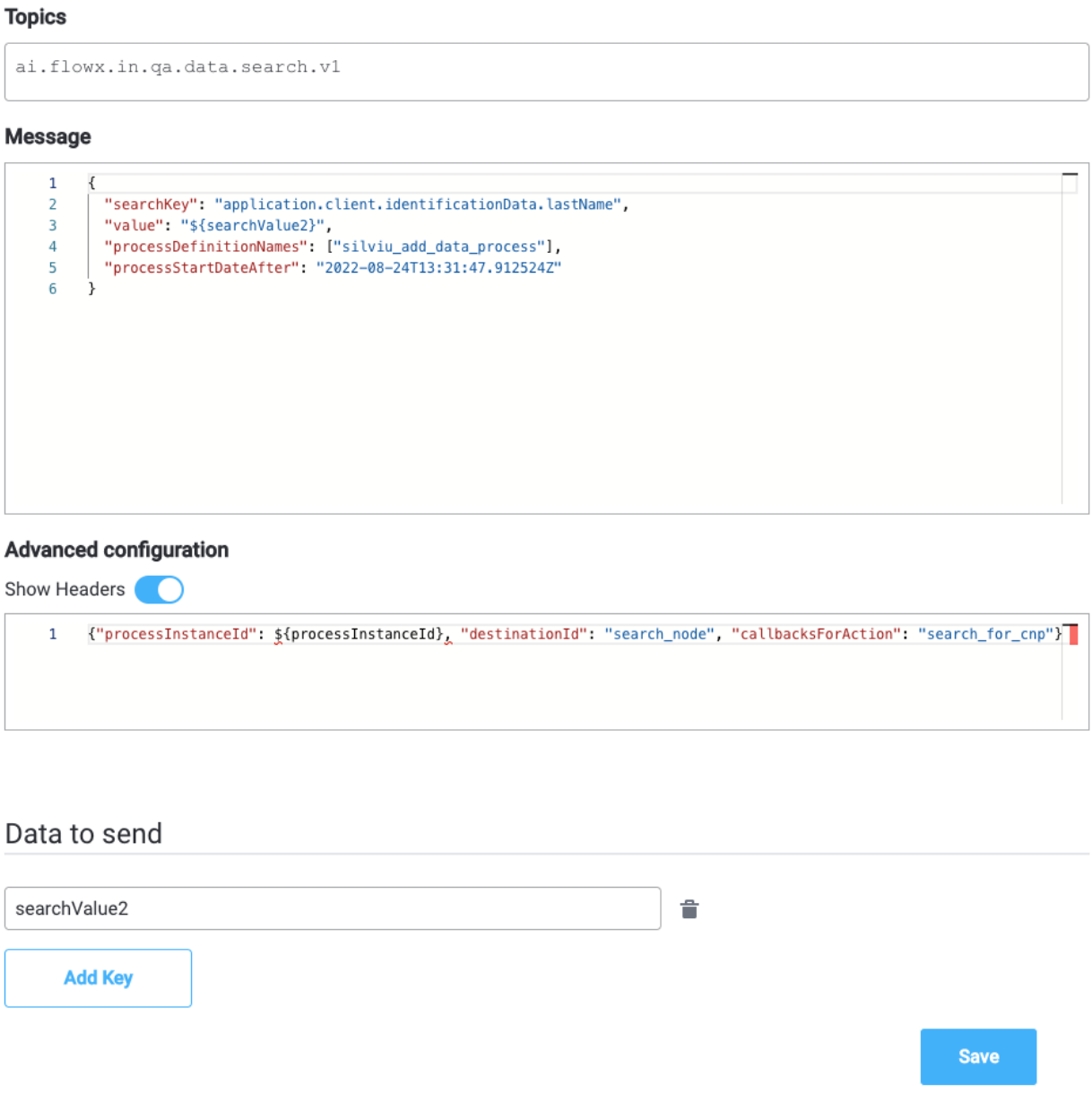


4. Configure the following items:

- **Topic name** - the Kafka topic on which the search service listens for requests; ❗ respect the naming pattern
- **Data to send** - (key) - used when data is sent from the frontend via an action to validate the data (you can find more information in the User Task configuration section)
- **Headers** - required
- **Body message**:
  - `searchKey` - it will hold the result received from the elastic search
  - `value` - value of the key
  - `processDefinitionNames` - the process definition names where to perform the search
  - `processStartDateAfter` - the service will look into process definitions created after the defined date

```
{
    "searchKey": "application.client.name",
    "value": "12344",
    "processStartDateAfter": "formatDeDataStandard", (opt)
    "processStartDateBefore": "formatDeDataStandard", (opt)
    "processDefinitionNames": [ "processDef1", "processDef2"
],
    "status": ["ANY",...]
}
```

- Example (dummy values extracted from a process):



5. A custom microservice (a core extension) will receive this event and will search the value of the process in the elastic search.
6. It will respond to the engine via a Kafka topic.

TIP

The topic must be defined in the **Node config** of the **User task** where you previously added the Kafka Send Action.

The **body message** of the response will look like this:

```
! If there is no result:
```

```
{
    "searchKey": "application.client.name",
    "result": [],
    "processStartDate": date,
    "tooManyResults": true|false
}
```

- Example (dummy values extracted from a process):

TIP

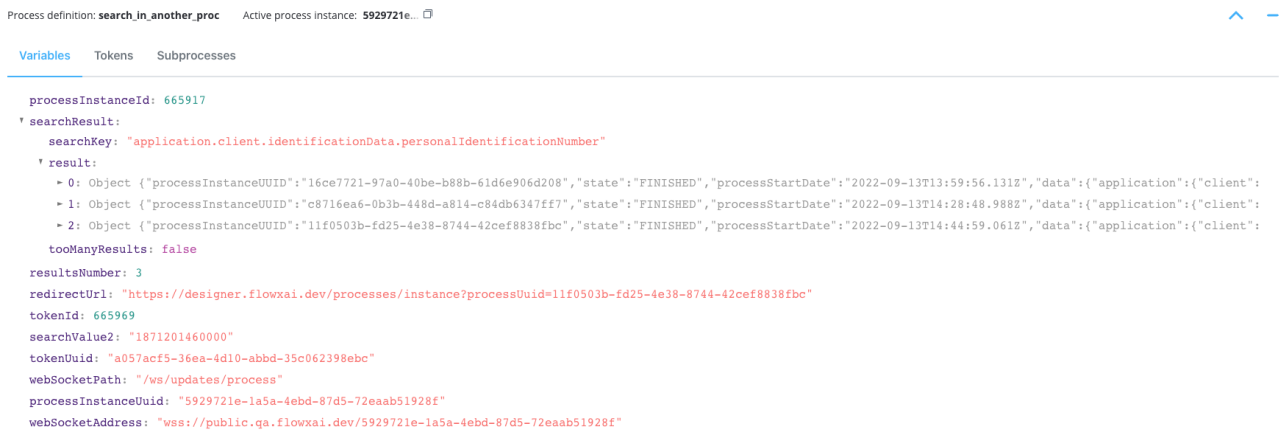
To access the view of your process variables, tokens and subprocesses go to **FLOWX.AI Designer > Active process > Process Instances**. Here you will find the response.

❗ If there is a list of results:

```
    "searchKey": "application.client.name"
    "result":[{
            "processInstanceUUID": "UUID",
            "status": "CREATED",
            "processStartDate": date,
            "data" : {"all data in elastic for that
process"}
    }],
    "tooManyResults": true|false
}
```

NOTE: You will receive up to 50 results - if `tooManyResults` is true.

- Example (dummy values extracted from a process):



Let's go now through the steps needed to deploy and set up the service:

» Search data service setup guide

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Renderer SDKs / Using the Angular Renderer

FlowxProcessRenderer is a low code library designed to render UI configured via the Flowx Process Editor.

Angular project requirements

Your app MUST be created using the NG app from the @angular/cli~15 package. It also MUST use SCSS for styling.

```
npm install -g @angular/cli@15.0
ng new my-flowx-app
```

INFO

To install the npm libraries provided by FLOWX you will need to obtain access to the private FLOWX Nexus registry. Please consult with your project DevOps.

CAUTION

The library uses Angular version **@angular~15**, **npm v8.1.2** and **node v16.13.2**.

CAUTION

If you are using an older version of Angular (for example, v14), please consult the following link for update instructions:

Update Angular from v14.0 to v15.0

Installing the library

Use the following command to install the **renderer** library and its required dependencies:

```
npm install @flowx/ui-sdk@3.21.0
@flowx/ui-toolkit@3.21.0
@flowx/ui-theme@3.21.0
paperflow-web-components
vanillajs-datepicker@1.3.1
moment@^2.27.0
@angular/flex-layout@15.0.0-beta.42
@angular/material@15.2.0
@angular/material-moment-adapter@15.2.0
@angular/cdk@15.2.0
ng2-pdfjs-viewer@15.0.0
event-source-polyfill@1.0.31
```

Also, in order to successfully link the pdf viewer, add the following declaration in the assets property of you project's angular.json:

```
{
  "glob": "**/*",
  "input": "node_modules/ng2-pdfjs-viewer/pdfjs",
  "output": "/assets/pdfjs"
}
```

Using the library

Once installed, FlxProcessModule will be imported in the `AppModule` `FlxProcessModule.forRoot({})`.

You MUST also import the dependencies of `FlxProcessModule: HttpClientModule` from `@angular/common/http` and **IconModule** from `@flowxai/ui-toolkit`.

Using Paperflow web components

Add path to component styles to stylePreprocessesOptions object in **angular.json file**

```
"stylePreprocessorOptions": {
  "includePaths": [
  "./node_modules/paperflow-web-components/src/assets/scss",
  "./node_modules/flowx-process-
renderer/src/assets/scss/style.scss",
  "src/styles"]
}
```

> **INFO**
>
> Because the datepicker module is build on top of angular material datepicker module, using it requires importing one predefined material theme in you **angular.json** configuration.

```
 "styles": ["...,
"./node_modules/@angular/material/prebuilt-themes/indigo-
pink.css"],
```

Theming

Component theming is done through two json files (`theme_tokens.json`, `theme_components.json`) that need to be added in the assets folder of your project The file paths need to be passed to the `FlxProcessModule.forRoot()` method through the `themePaths` object.

```
themePaths: {
    components: 'assets/theme/theme_components.json',
    tokens: 'assets/theme/theme_tokens.json',
  },
```

The **assets/theme/theme_tokens.json** - should hold the design tokens (e.g. colors, fonts) used in the theme. An example can be found here.

The **assets/theme/theme_components.json** - holds metadata used to describe component styles. An example can be found here.

For **Task Management** theming is done through the ppf-theme mixin that accepts as an argument a list of colors grouped under **primary**, **status** and **background**

```
@use 'ppf-theme';

@include ppf-theme.ppf-theme((
  'primary': (
    'color1': vars.$primary,
    'color2': vars.$secondary,
    'color3': vars.$text-color,
  ),
  'status': (
    'success': vars.$success,
    'warning': vars.$warning,
    'error': vars.$error,
  ),
  'background': (
    'background1': vars.$background1,
    'background2': vars.$background2,
    'background3': vars.$background3,
```

```
  ),
));
```

Authorization

> **INFO**
>
> Every request from the **FLOWX** renderer SDK will be made using the **HttpClientModule** of the client app, which means those requests will go through every interceptor you define here. This is most important to know when building the auth method as it will be the job of the client app to intercept and decorate the requests with the necessary auth info (eg. `Authorziation: Bearer ...`).

> **NOTE**
>
> It's the responsibility of the client app to implement the authorization flow (using the **OpenID Connect** standard). The renderer SDK will expect to find the **JWT** saved in the browser **localStorage** object at the key named access_token.

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import { HttpClientModule, HTTP_INTERCEPTORS } from
'@angular/common/http';
import {FlxProcessModule} from 'flowx-process-renderer';
import { IconModule } from 'paperflow-web-components';

import {AppRoutingModule} from './app-routing.module';
import {AppComponent} from './app.component';
```

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    // will be used by the renderer SDK to make requests
    HttpClientModule,
    // needed by the renderer SDK
    IconModule.forRoot(),
    FlxProcessModule.forRoot({
      components: {},
      services: {},
      themePaths: {
        components: 'assets/theme/theme_components.json',
        tokens: 'assets/theme/theme_tokens.json',
      },
    }),
  ],
  // this interceptor with decorate the requests with the
Authorization header
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor,
multi: true },
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

The `forRoot()` call is required in the application module where the process will be rendered. The `forRoot()` method accepts a config argument where you can

pass extra config info, register a **custom component**, **service**, or **custom validators**.

Custom components will be referenced by name when creating the template config for a user task.

Custom validators will be referenced by name (`currentOrLastYear`) in the template config panel in the validators section of each generated form field.

```
// example
FlxProcessModule.forRoot({
  components: {
    YourCustomComponentIdenfier: CustomComponentInstance,
  },
  services: {
    NomenclatorService,
    LocalDataStoreService,
  },
  validators: {currentOrLastYear },
})
```

```
# example with custom component and custom validator
FlxProcessModule.forRoot({
  components: {
    YourCustomComponentIdenfier: CustomComponentInstance,
  },
  services: {
    NomenclatorService,
    LocalDataStoreService,
  },
  validators: {currentOrLastYear },
})
```

```
  // example of a custom validator that restricts data
selection to
  // the current or the previous year

  currentOrLastYear: function currentOrLastYear(AC:
AbstractControl): { [key: string]: any } {
    if (!AC) {
      return null;
    }

    const yearDate = moment(AC.value, YEAR_FORMAT, true);
    const currentDateYear = moment(new
Date()).startOf('year');
    const lastYear = moment(new Date()).subtract(1,
'year').startOf('year');

    if (!yearDate.isSame(currentDateYear) &&
!yearDate.isSame(lastYear)) {
      return { currentOrLastYear: true };
    }

    return null;
  }
```

CAUTION

The error that the validator returns **MUST** match the validator name.

The component is the main container of the UI, which will build and render the components configured via the **FlowX Designer**. It accepts the following inputs:

```
<flx-process-renderer
  [apiUrl]="baseApiUrl"
  [processApiPath]="processApiPath"
  [processName]="processName"
  [processStartData]="processStartData"
  [debugLogs]="debugLogs"
  [keepState]="keepState"
  [language]="language"
></flx-process-renderer>
```

Parameters:

Name	Description	Type	Mandatory	Default value	
baseApiUrl	Your base url	string	true	-	https
processApiPath	Engine API prefix	string	true	-	/onb
processName	Identifies a process	string	true	-	clien
processStartData	Data required to start the process	json	true	-	{ "firs "last

Name	Description	Type	Mandatory	Default value	
debugLogs	When set to true this will print WS messages in the console	boolean	false	false	-
language	Language used to localize the application.	string	false	ro-RO	-

Name	Description	Type	Mandatory	Default value	
keepState	By default all process data is reset when the process renderer component gets destroyed. Setting this to true will keep process data even if the viewport gets destroyed	boolean	false	false	-

Name	Description	Type	Mandatory	Default value	
isDraft	When true allows starting a process in draft state. *Note that isDraft = true requires that processName be the **id** (number) of the process and NOT the name.	boolean	false	false	-

Data and actions

Custom components will be hydrated with data through the $data input observable which must be defined in the custom component class.

```
@Component({
  selector: 'my-custom-component',
  templateUrl: './custom-component.component.html',
  styleUrls: ['./custom-component.component.scss'],
})
```

```
export class CustomComponentComponent  {
  @Input() data$: Observable<any>;
  }
```

Component actions are always found under `data` -> `actionsFn` key.

Action names are configurable via the process editor.

```
# data object example
data: {
   actionsFn: {
      action_one: () => void;
      action_two: () => void; }
   }
```

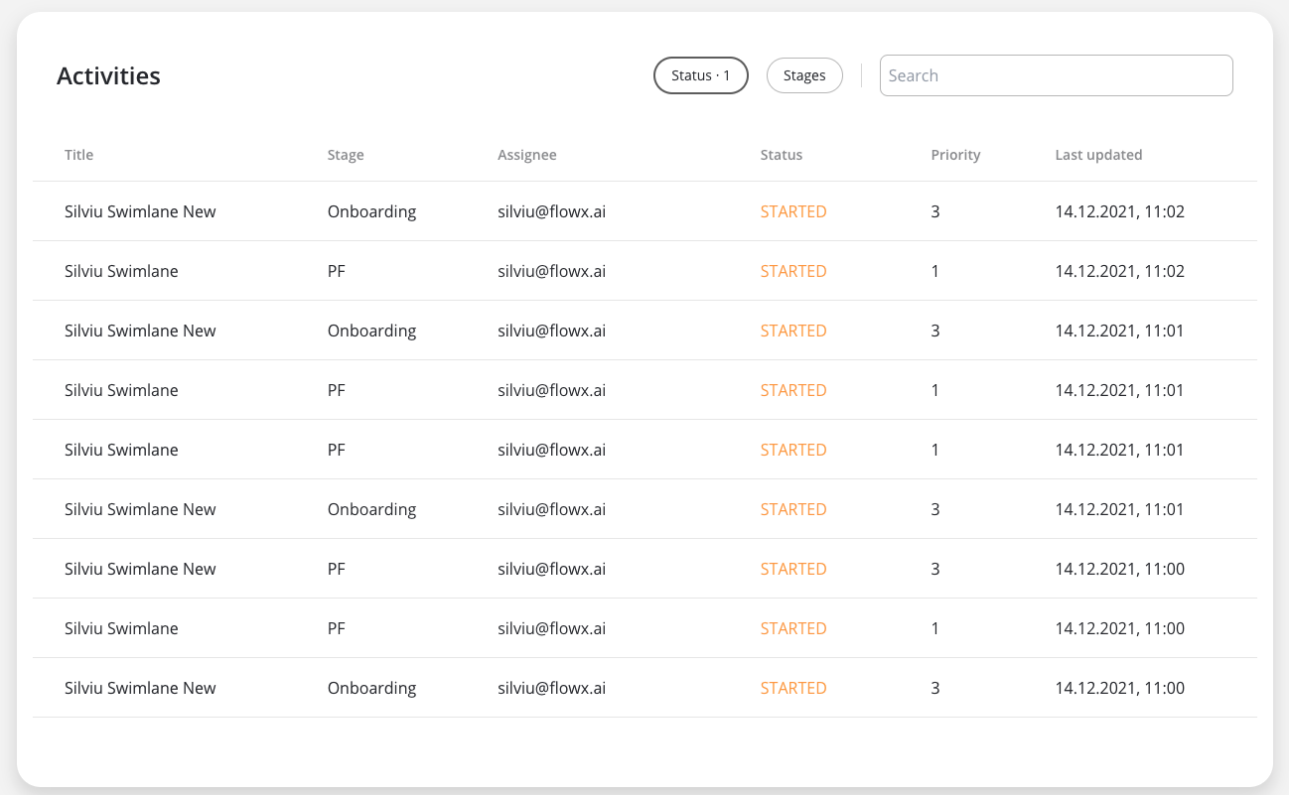
Interacting with the process

Data from the process is communicated via **SSE** protocol under the following keys:

Name	Description	Example	
Data	data updates for process model bound to default/custom components		
ProcessMetadata	updates about process metadata, ex: progress update, data about how to render components		

Name	Description	Example	
RunAction	instructs the UI to perform the given action		

Task management component



The flx-task-management component is found in the FlxTaskManagementModule. In order to have access to it, import the module where needed:

```
import {FlxTaskManagementModule} from 'flowx-process-
renderer';
@NgModule({
  declarations: [
```

```
    ...,
  ],
  imports: [
    ...,
    FlxTaskManagementModule
  ],

})
export class MyModule {
}
```

Then in the template:

```
<flx-task-management [baseUrl]="baseUrl" [title]="'Tasks'">
</flx-task-management>
```

Parameters:

Name	Description	Type	Default	Mandatory	
apiUrl	Endpoint where the tasks are available	string	-	true	https://yo
title	Table header value	string	Activities	false	Tasks

Name	Description	Type	Default	Mandatory	
pollingInterval	Interval for polling task updates	number	5000 ms	false	10000

Development

When modifying the library source code and testing it inside the designer app use the following command which rebuilds the flx-process-renderer library, recreates the link between the library and the designer app and recompiles the designer app:

```
npm run build && cd dist/flowx-process-renderer/ && npm link
&& cd ../../ && npm link flowx-process-renderer && npm run
start:designer
```

or alternatively run

`./start_with_build_lib.sh`

If you want to start the designer app and the flx-process-renderer library in development mode (no need to recompile the lib for every change) run the following command:

`npm run start:designer-dev`

CAUTION

> Remember to test the final version of the code by building and bundling the renderer library to check that everything works e2e

Trying to use this lib with npm link from another app will most probably fail. If (when) that happens, there are two alternatives that you can use:

1. Use the build-and-sync.sh script, that builds the lib, removes the current build from the client app **node_modules** and copies the newly build lib to the node_modules dir of the client app:

```
./build-and-sync.sh ${path to the client app root}

# example (the client app is demo-web):
./build-and-sync.sh ../../demo-web
```

NOTE: This method uses under the hood the build-and-sync.sh script from the first version and the chokidar-cli library to detect file changes.

2. Use the build-and-sync:watch npm script, that builds the library and copies it to the client app's **node_module** directory every time a file changes:

```
npm run build-and-sync:watch --target-path=${path to the
client app root}

# example (the client app is demo-web):
npm run build-and-sync:watch --target-path=../../demo-web
```

Running the tests

```
ng test
```

Coding style tests

Always follow the Angular official coding styles.

Below you will find a Storybook which will demonstrate how components behave under different states, props, and conditions, it allows you to preview and interact with individual UI components in isolation, without the need for a full-fledged application:

» Storybook

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Renderer SDKs / Using the iOS Renderer

iOS Project Requirements

The minimum requirements are:

- iOS 14
- Swift 5.0

Installing the library

The iOS Renderer is available through Cocoapods and Swift Package Manager.

Swift Package Manager

In Xcode, click `File` → `Add Packages...`, enter FlowX repo's URL `https://github.com/flowx-ai/flowx-ios-sdk`. Set the dependency rule to `Up To Next Major` and add package.

If you are developing a framework and use FlowX as a dependency, add to your `Package.swift` file:

```
dependencies: [
    .package(url: "https://github.com/flowx-ai/flowx-ios-
sdk", .upToNextMajor(from: "0.96.0"))
]
```

Cocoapods

Prerequisites

- Cocoapods gem installed

Cocoapods private trunk setup

Add the private trunk repo to your local Cocoapods installation with the command:

```
pod repo add flowx-specs git@github.com:flowx-ai/flowx-ios-
specs.git
```

Adding the dependency

Add the source of the private repository in the Podfile

```
source 'git@github.com/flowx-ios-specs.git'
```

Add the pod and then run `pod install`

```
pod 'FlowX'
```

Library dependencies

The iOS Renderer library depends on the following libraries:

- Socket.IO-Client-Swift
- Alamofire
- SVProgressHUD
- SDWebImageSwiftUI

Configuring the library

The SDK has 2 configurations, available through shared instances.

It is recommended to call the configuration methods at app launch.

Otherwise, make sure you do it before the start of any FlowX process.

FXConfig

This config is used for general purpose properties.

Properties

Name	Description	Type	Requirement
baseURL	The base URL used for REST networking	String	Mandatory
imageBaseURL	The base URL used for media library images	String	Mandatory
language	The language used for retrieving enumerations and substitution tags	String	Mandatory. Defaults to "en"

Name	Description	Type	Requirement
stepViewType	The type of the custom step view class	FXStepViewProtocol.Type	Optional
logEnabled	Value indicating whether console logging is enabled. Default is false	Bool	Optional

Sample

```
FXConfig.sharedInstance.configure { (config) in
    config.baseURL = myBaseURL
    config.imageBaseURL = myImageBaseURL
    config.language = "en"
    config.logEnabled = true
    config.stepViewType = CustomStepView.self
}
```

FXSessionConfig

This config is used for providing networking or auth session-specific properties.

The library expects a session instance managed by the container app. Request adapting and retrying are handled by the container app.

Properties

Name	Description	Type
sessionManager	Alamofire session instance used for REST networking	Session
token	JWT authentication access token	String

Sample

```
FXSessionConfig.sharedInstance.configure { config in
    config.sessionManager = mySessionManager
    config.token = myAccessToken
}
```

Using the library

The library's public APIs are called using the shared instance of FlowX, `FlowX.sharedInstance`.

How to start and end FlowX session

After all the configurations are set, you can start a FlowX session by calling the `startSession()` method.

This is optional, as the session starts lazily when the first process is started.

`FlowX.sharedInstance.startSession()`

When you want to end a FlowX session, you can call the `endSession()` method. This also does a complete clean-up of the started processes.

You might want to use this method in a variety of scenarios, for instance when the user logs out.

`FlowX.sharedInstance.endSession()`

How to start a process

You can start a process by calling the method below.

The container app is responsible with presenting the navigation controller holding the process navigation.

```
public func startProcess(navigationController:
UINavigationController,
                         name: String,
                         params: [String: Any]?,
                         isModal: Bool = false,
                         showLoader: Bool = false)
```

`navigationController` - the instance of UINavigationController which will hold the process navigation stack

`name` - the name of the process

`params` - the start parameters, if any

`isModal` - a boolean indicating whether the process navigation is modally displayed. When the process navigation is displayed modally, a close bar button item is displayed on each screen displayed throughout the process navigation.

`showLoader` - a boolean indicating whether the loader should be displayed when starting the process.

Sample

```
FlowX.sharedInstance.startProcess(navigationController:
processNavigationController,
                                  name: processName,
                                  params: startParams,
                                  isModal: true
                                  showLoader: true)

self.present(processNavigationController, animated: true,
completion: nil)
```

How to resume a process

You can resume a process by calling the method below.

```
public func continueExistingProcess(uuid: String,
                                    name: String,
                                    navigationController:
UINavigationController,
                                    isModal: Bool = false) {
```

`uuid` - the UUID string of the process

`name` - the name of the process

`navigationController` - the instance of UINavigationController which will hold the process navigation stack

`isModal` - a boolean indicating whether the process navigation is modally displayed. When the process navigation is displayed modally, a close bar button item is displayed on each screen displayed throughout the process navigation.

How to end a process

You can manually end a process by calling the `stopProcess(name: String)` method.

This is useful when you want to explicitly ask the FlowX shared instance to clean up the instance of the process sent as parameter.

For example, it could be used for modally displayed processes that are dismissed by the user, in which case the `dismissRequested(forProcess process: String, navigationController: UINavigationController)` method of the FXDataSource will be called.

Sample

```
FlowX.sharedInstance.stopProcess(name: processName)
```

How to run an action from a custom component

The custom components which the container app provides will contain FlowX actions to be executed. In order to run an action you need to call the following method:

```
public func runAction(action: ProcessActionModel,
                      params: [String: Any]? = nil)
```

`action` - the `ProcessActionModel` action object

`params` - the parameters for the action

How to run an upload action from a custom component

```
public func runUploadAction(action: ProcessActionModel,
                            image: UIImage)
```

`action` - the `ProcessActionModel` action object

`image` - the image to upload

```
public func runUploadAction(action: ProcessActionModel,
                            fileURL: URL)
```

`action` - the `ProcessActionModel` action object

`fileURL` - the local URL of the image

Getting a substitution tag value by key

```
public func getTag(withKey key: String) -> String?
```

All substitution tags will be retrieved by the SDK before starting the first process and will be stored in memory.

Whenever the container app needs a substitution tag value for populating the UI of the custom components, it can request the substitution tag using the method above, providing the key.

Getting a media item url by key

```
public func getMediaItemURL(withKey key: String) -> String?
```

All media items will be retrieved by the SDK before starting the first process and will be stored in memory.

Whenever the container app needs a media item url for populating the UI of the custom components, it can request the url using the method above, providing the key.

```
public func getTag(withKey key: String) -> String?
```

All substitution tags will be retrieved by the SDK before starting the first process and will be stored in memory.

Whenever the container app needs a substitution tag value for populating the UI of the custom components, it can request the substitution tag using the method

above, providing the key.

Handling authorization token changes

When the access token of the auth session changes, you can update it in the renderer using the `func updateAuthorization(token: String)` method.

FXDataSource

The library offers a way of communication with the container app through the `FXDataSource` protocol.

The data source is a public property of FlowX shared instance.

`public weak var dataSource: FXDataSource?`

```
public protocol FXDataSource: AnyObject {
    func controllerFor(componentIdentifier: String) ->
FXController?

    func viewFor(componentIdentifier: String) -> FXView?

    func viewFor(componentIdentifier: String,
customComponentViewModel: FXCustomComponentViewModel) ->
AnyView?

    func navigationController() -> UINavigationController?

    func errorReceivedForAction(name: String?)

    func validate(validatorName: String, value: String) ->
Bool
```

```
    func dismissRequested(forProcess process: String,
navigationController: UINavigationController)

}
```

- `func controllerFor(componentIdentifier: String) -> FXController?`

This method is used for providing a custom component UIKit view controller, identified by the componentIdentifier argument.

- `func viewFor(componentIdentifier: String) -> FXView?`

This method is used for providing a custom UIKit view, identified by the componentIdentifier argument.

- `func viewFor(componentIdentifier: String, customComponentViewModel: FXCustomComponentViewModel) -> AnyView?`

This method is used for providing a custom SwiftUI view, identified by the componentIdentifier argument. A view model is provided as an ObservableObject to be added as @ObservedObject inside the SwiftUI view.

- `func navigationController() -> UINavigationController?`

This method is used for providing a navigation controller. It can be either a custom `UINavigationController` class, or just a regular `UINavigationController` instance themed by the container app.

- `func errorReceivedForAction(name: String?)`

This method is called when an error occurs after an action is executed.

- `func validate(validatorName: String, value: String) -> Bool`

This method is used for custom validators. It provides the name of the validator and the value to be validated. The method returns a boolean indicating whether the value is valid or not.

- `func dismissRequested(forProcess process: String, navigationController: UINavigationController)`

This method is called, on a modally displayed process navigation, when the user attempts to dismiss the modal navigation. Typically it is used when you want to present a confirmation pop-up.

The container app is responsible with dismissing the UI and calling the stop process APIs.

FXController

FXController is an open class, which helps the container app provide UIKit custom component screens to the renderer. It needs to be subclassed for each custom screen.

```
open class FXController: UIViewController {

    internal(set) public var data: [String: Any]?
    internal(set) public var actions: [ProcessActionModel]?
```

```
    open func titleForScreen() -> String? {
        return nil
    }

    open func populateUI(data: [String: Any]) {

    }

    open func updateUI(data: [String: Any]) {

    }

}
```

- `internal(set) public var data: [String: Any]?`

`data` is a dictionary property, containing the data model for the custom component.

- `internal(set) public var actions: [ProcessActionModel]?`

`actions` is the array of actions provided to the custom component.

- `func titleForScreen() -> String?`

This method is used for setting the screen title. It is called by the renderer when the view controller is displayed.

- `func populateUI(data: [String: Any])`

This method is called by the renderer, after the controller has been presented, when the data is available.

This will happen asynchronously. It is the container app's responsibility to make sure that the initial state of the view controller does not have default/residual values displayed.

- `func updateUI(data: [String: Any])`

This method is called by the renderer when an already displayed view controller needs to update the data shown.

FXView

FXView is a protocol that helps the container app provide custom UIKit subviews of a generated screen to the renderer. It needs to be implemented by `UIView` instances. Similar to `FXController` it has data and actions properties and a populate method.

```
public protocol FXView: UIView {
    var data: [String: Any]? { get set }
    var actions: [ProcessActionModel]? { get set }

    func populateUI(data: [String: Any]?)
}
```

- `var data: [String: Any]?`

`data` is a dictionary property containing the data model needed by the custom view.

- `var actions: [ProcessActionModel]?`

`actions` is the array of actions provided to the custom view.

- `func populateUI(data: [String: Any]?)`

This method is called by the renderer after the screen containing the view has been displayed.

It is the container app's responsibility to make sure that the initial state of the view does not have default/residual values displayed.

NOTE: It is mandatory for views implementing the FXView protocol to provide the intrinsic content size. Sample:

```
override var intrinsicContentSize: CGSize {
    return CGSize(width: UIScreen.main.bounds.width, height:
100)
}
```

FXCustomComponentViewModel

`FXCustomComponentViewModel` is a class implementing the `ObservableObject` protocol. It is used for managing the state of custom SwiftUI views. It has two published properties, for data and actions.

```
    @Published public var data: [String: Any] = [:]
    @Published public var actions: [ProcessActionModel] = []
```

Example

```
struct SampleView: View {

    @ObservedObject var viewModel:
```

```
FXCustomComponentViewModel

    var body: some View {
        Text("Lorem")
    }
}
```

Was this page helpful?

PLATFORM DEEP DIVE / Core components / Renderer SDKs / Using the Android Renderer

Android project requirements

To use the Android Renderer library, ensure that your Android project meets the following minimum requirements:

- minSdk 26

Installing the library

1. Add the following code to your Android project's `settings.gradle` file::

```
dependencyResolutionManagement {
    ...
```

```
    repositories {
        ...
        maven {
            credentials {
                username "YOUR_USERNAME_HERE"
                password "YOUR_PASSWORD_HERE"
            }
            url 'https://nexus-
jx.dev.rd.flowx.ai/repository/flowx-maven-releases/'
        }
    }
}
```

2. Add the following code to your `app/build.gradle` file:

```
dependencies {
    ...
    implementation "ai.flowx.android:android-sdk:2.0.1"
    ...
}
```

Library dependencies

The Android Renderer library depends on the following libraries:

- Koin
- Retrofit
- Coil

Accessing the documentation

To access the Android Renderer library's documentation, follow these steps:

1. Download the **javadoc.jar** file from the same repository as the library.
2. Extract the **javadoc.jar** file.
3. Open the **index.html** file in your browser.
4. Navigate to `ai.flowx.android.sdk.FlowxSdkApi`.

Was this page helpful?