FLOWX.AI

BUILDING BLOCKS / UI Designer

Contents

- BUILDING BLOCKS / UI Designer / UI component types / Root components / Container
- BUILDING BLOCKS / UI Designer / UI component types / Root components / Card
- BUILDING BLOCKS / UI Designer / UI component types / Root components / Custom
- BUILDING BLOCKS / UI Designer / UI component types / Collection / Collection Prototype
  - Description
  - Configurable properties:
  - Example
  - Adding elements with UI Actions
    - Step 1 - Defining the Node Action
    - Step 2 - Adding the Button & UI Action
    - Result
- BUILDING BLOCKS / UI Designer / UI component types / Buttons
  - Basic button
    - Configuring a basic button
    - Button styling
  - File upload
    - Configuring a file upload button
    - Button styling
- BUILDING BLOCKS / UI Designer / UI component types / File Preview
  - Configuring a File Preview element
    - File Preview properties (web)
    - File Preview properties (mobile)
    - File preview styling

    - File Preview example
- BUILDING BLOCKS / UI Designer / UI component types / Form elements / Input
    - Configuring the input element
        - Input settings
        - Input styling
- BUILDING BLOCKS / UI Designer / UI component types / Form elements / Text area
    - Configuring the text area element
        - Text area settings
        - Text area styling
- BUILDING BLOCKS / UI Designer / UI component types / Form elements / Select
    - Configuring the Select element
        - Select Settings
        - Select styling
    - Example - Dynamic dropdowns
        - Creating the process
        - Configuring the nodes
        - Configuring the UI
- BUILDING BLOCKS / UI Designer / UI component types / Form elements / Checkbox
    - Configuring the checkbox element
        - Checkbox settings
        - Checkbox styling
- BUILDING BLOCKS / UI Designer / UI component types / Form elements / Radio
    - Configuring the radio field element
        - Radio settings

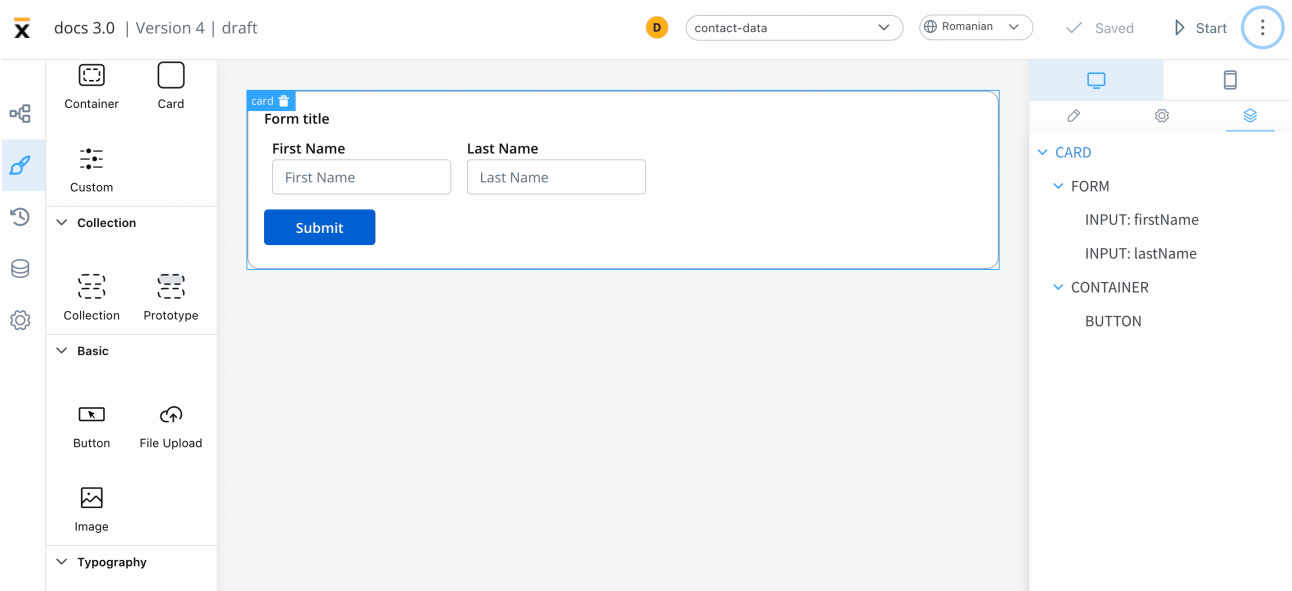
        - Radio styling
- BUILDING BLOCKS / UI Designer / UI component types / Form elements / Switch
    - Configuring the switch element
        - Switch settings
- BUILDING BLOCKS / UI Designer / UI component types / Form elements / Datepicker
    - Configuring the datepicker element
        - Datepicker settings
        - Datepicker styling
- BUILDING BLOCKS / UI Designer / UI component types / Form elements / Slider
    - Configuring the slider element
        - Slider settings
        - Multiple sliders
        - Slider styling
- BUILDING BLOCKS / UI Designer / UI component types / Form elements / Segmented button
    - Configuring the segmented button
        - Segmented button settings
        - Segmented button styling
- BUILDING BLOCKS / UI Designer / UI component types / Image
    - Configuring an image
        - Image settings
        - Media library
        - Process Data
        - External
    - UI actions
    - Image styling

- BUILDING BLOCKS / UI Designer / UI actions
  - Process UI actions
    - Manual action configuration example - Save Data
    - UI action configuration example
  - UI actions elements
    - Events
    - Action types
  - External UI actions
- BUILDING BLOCKS / UI Designer / Validators
  - Predefined validators
    - required validator
    - minlength validator
    - maxlength validator
    - min validator
    - max validator
    - email validator
    - pattern validator
    - datepicker - isSameOrBeforeToday
    - datepicker - isSameOrAfterToday
  - Custom validators
  - sidebar_position: 3
- BUILDING BLOCKS / UI Designer / Dynamic & computed values
  - Dynamic values
    - Example using Substitution tags
    - Example using process parameters
  - Computed values
    - Slider example
    - Usage
- BUILDING BLOCKS / UI Designer / Layout configuration

- BUILDING BLOCKS / UI Designer / Rendering and UI Designer changelog
  - Notes for post-migration

BUILDING BLOCKS / UI Designer / UI component types / Root components / Container

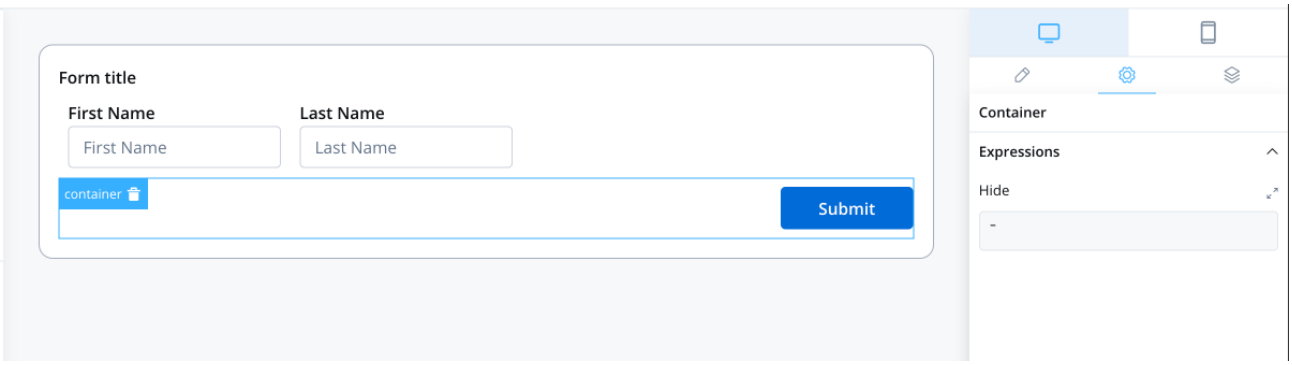
A container is a versatile element that allows you to group components and align them as desired.



The following properties can be configured in the container:

Settings

- **Expressions (Hide)** - JavaScript expressions used to hide components when they evaluate to true



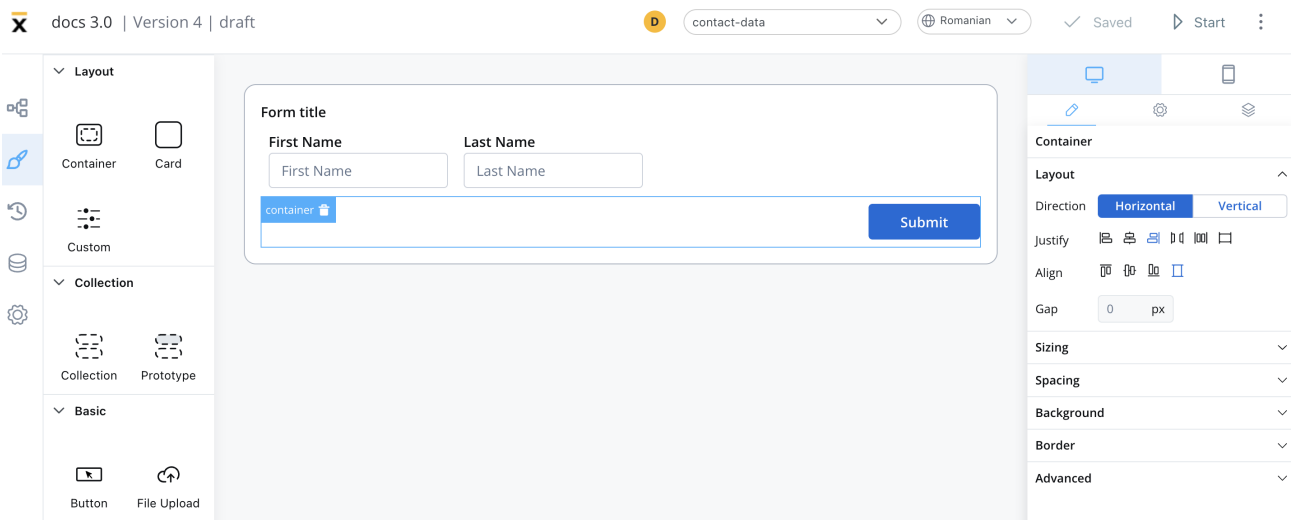
Styling

- **Layout** - this property is available for components that group children and includes the following options:
    - Direction - Horizontal / Vertical (for example, select *Horizontal*)
    - Justify (H) - (for example, select *end*)
    - Align (V) - this option allows you to align components vertically
    - Gap - you can set the gap between components

More layout demos available below:

» Layout Demos

When you apply the above properties, you can generate the following output, with the button appearing on the right side of the container, underneath the form with three form elements:



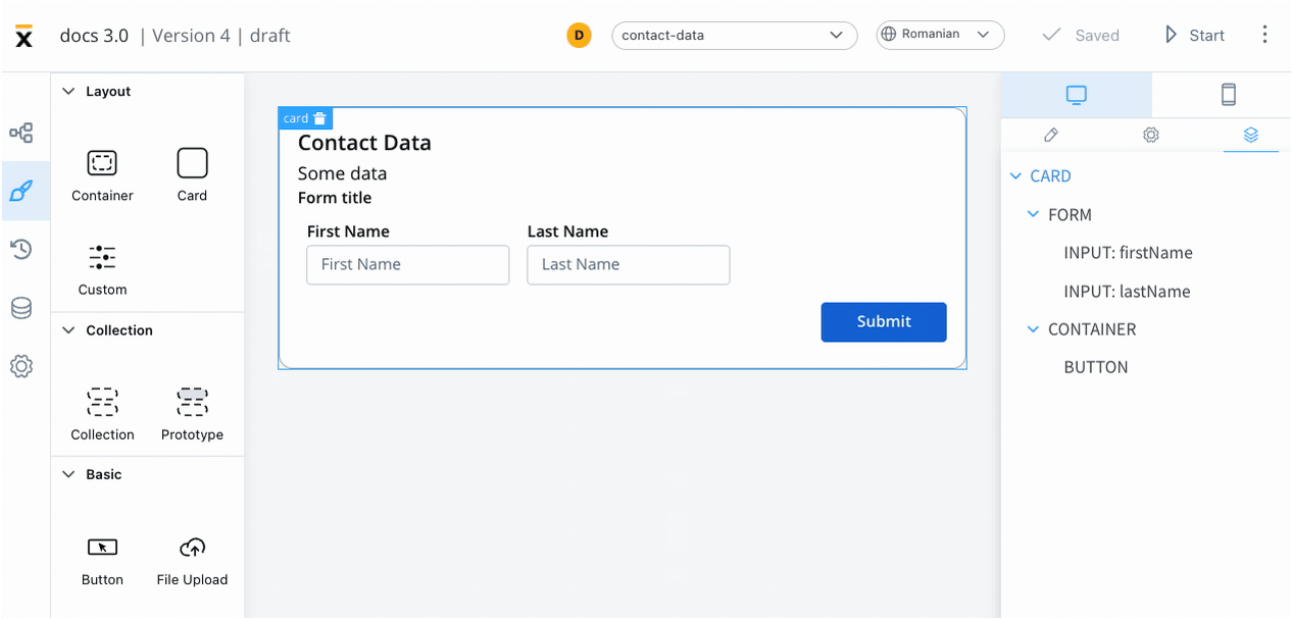
For more information about styling and layout configuration, check the following section:

» UI Designer

Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Root components / Card

A card is a graphical component that allows grouping and alignment of other components. It can also include an accordion element for expanding and collapsing content.

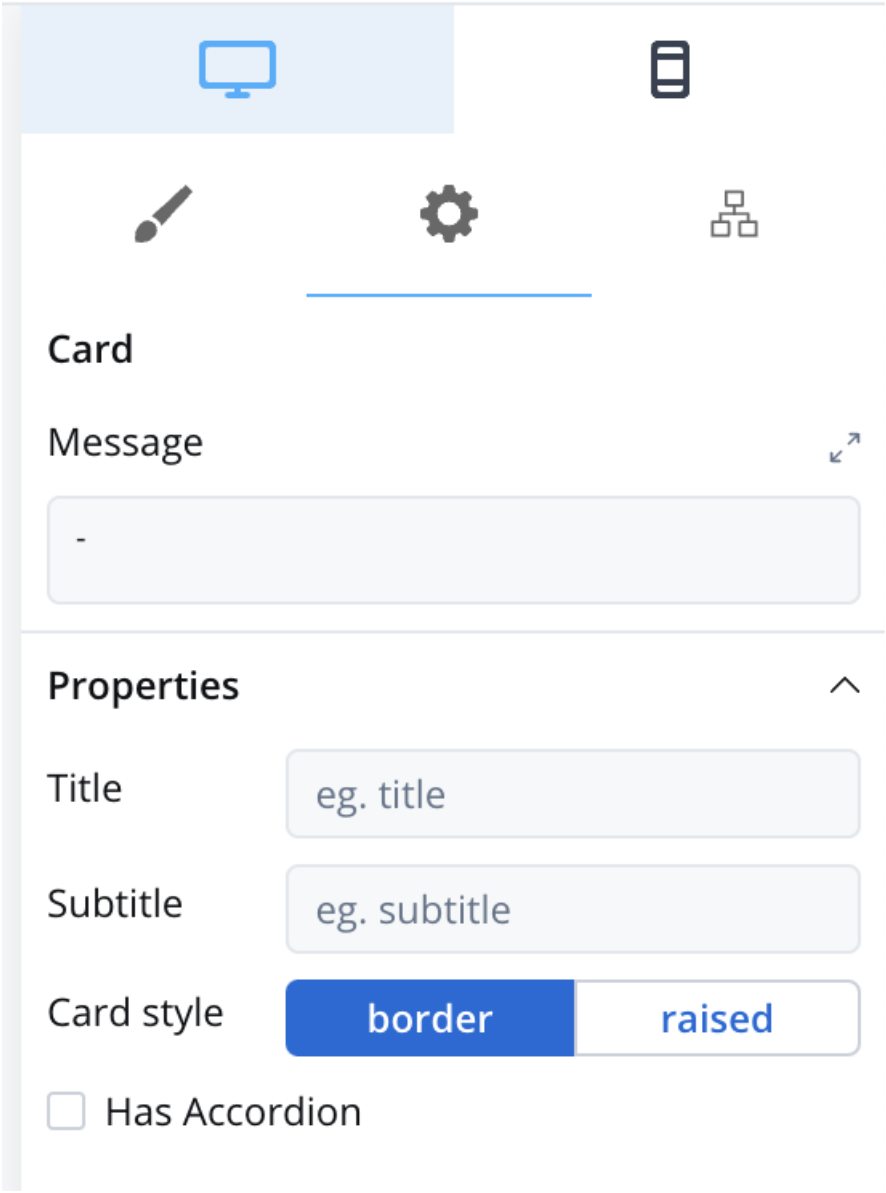


The following properties that can be configured:

Settings

- **Message** - a valid JSON that describes the data pushed to the frontend application when the process reaches a specific user task
- **Title** - the title of the card
- **Subtitle** - the subtitle of the card
- **Card style** - you can choose between a border or raised style
- **Has accordion?** - this feature allows you to add a Bootstrap accordion, which organizes content within collapsible items and displays only one collapsed item at a time

> **CAUTION**
>
> Accordion element is not available for mobile.


Card
Message
-
Properties
Title
eg. title
Subtitle
eg. subtitle
Card style
border
raised
Has Accordion

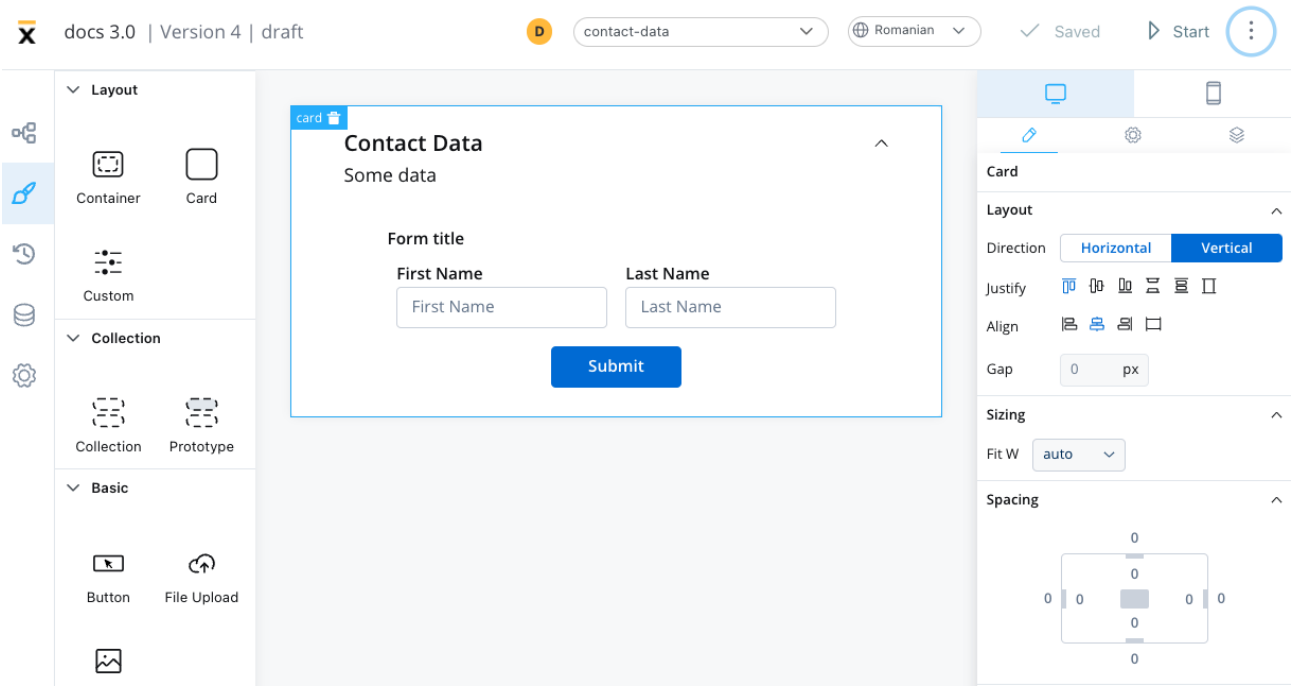
Styling

- **Layout** - This property is available for components that group children and includes the following options:
  - Direction - Horizontal / Vertical (for example, select *Vertical*)
  - Justify (H) - (for example, select *center*)
  - Align (V) - this option allows you to align components vertically
  - Gap - you can set the gap between components

More layout demos available below:

» Layout Demos

This example will generate a card with the following layout configuration:

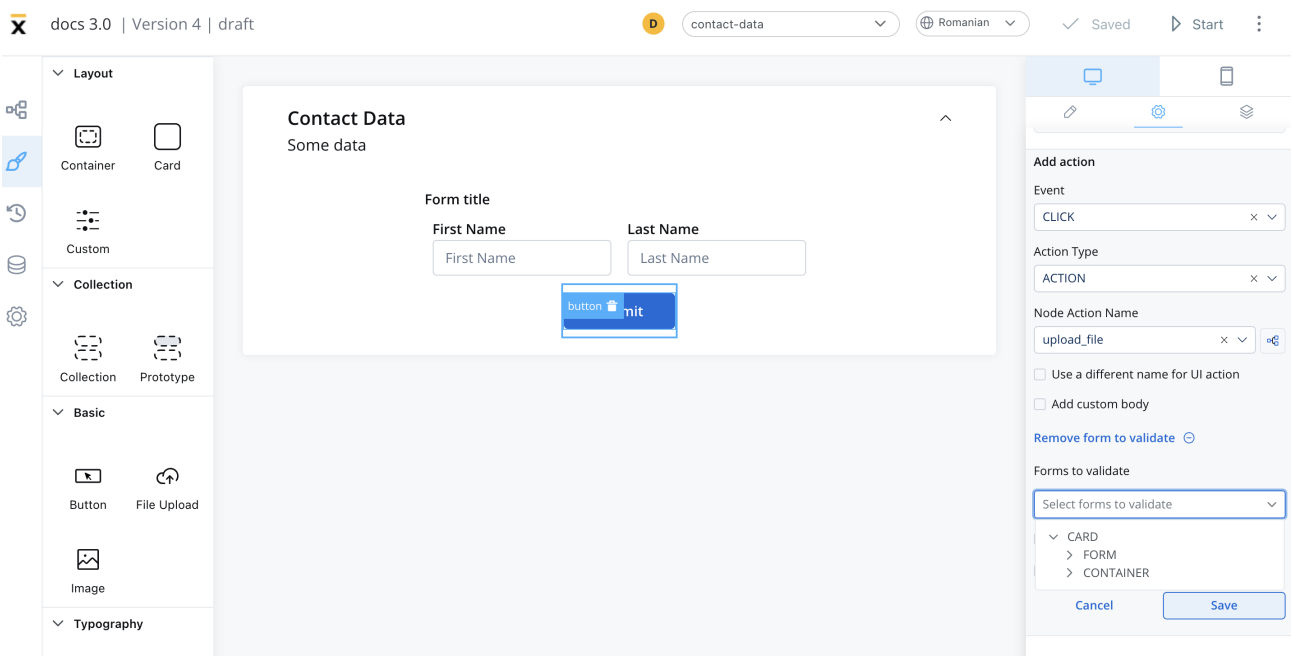


For more information about styling and layout configuration, check the following section:

» UI Designer

Validating elements

To validate all form elements under a card, you need to set the key of the form/element on the property of the button: *Forms To Validate*.

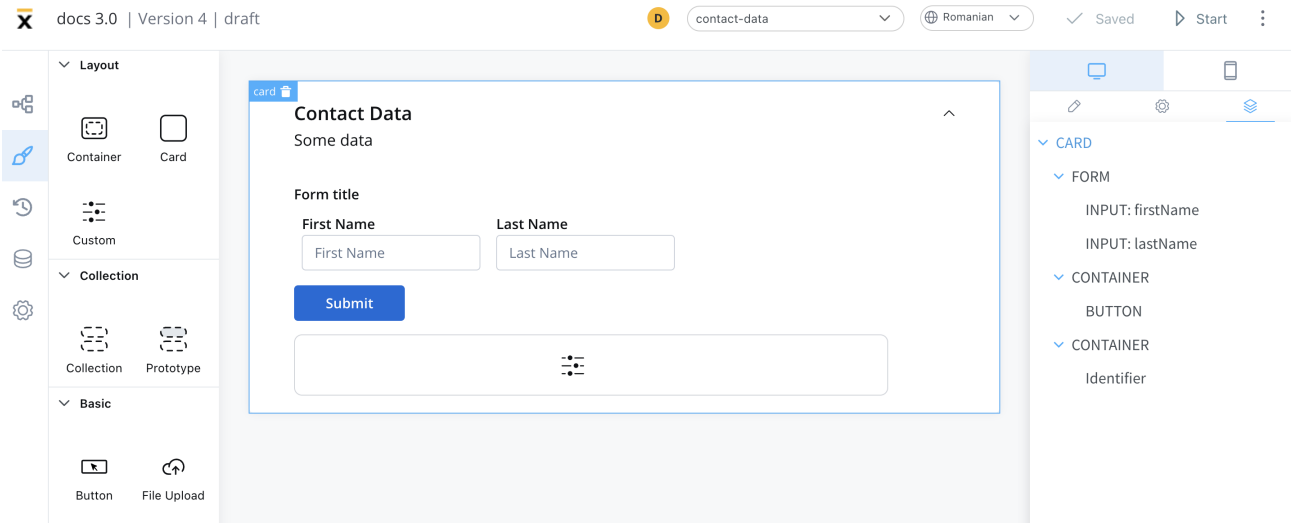


Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Root components / Custom

Custom components are developed in the web application and referenced here by component identifier. This will dictate where the component is displayed in the component hierarchy and what actions are available for the component.

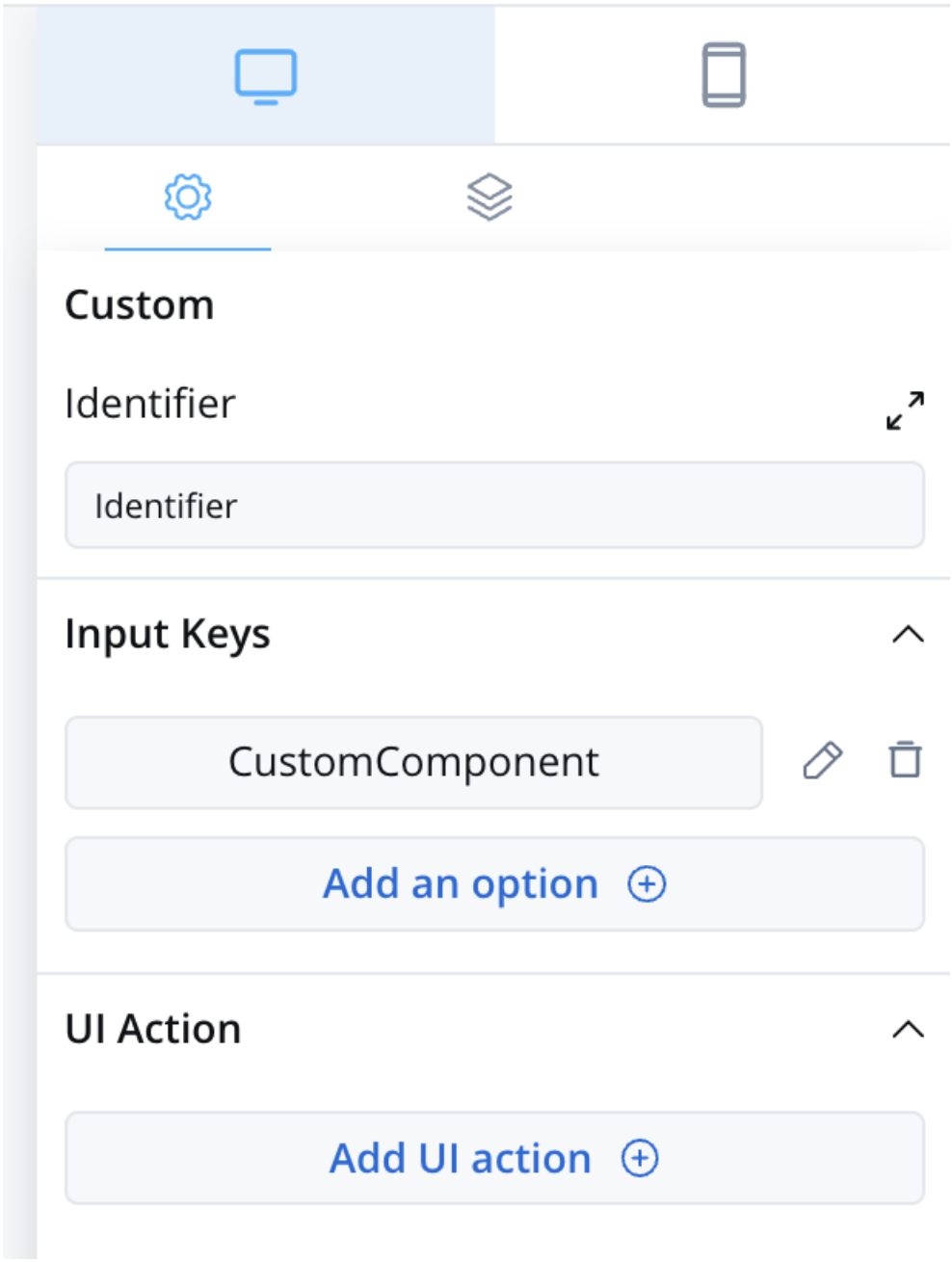
To add a custom component in the template config tree, we need to know its unique identifier and the data it should receive from the process model.



The properties that can be configured are as follows:

- **Identifier** - this will enable the custom component to be displayed in the component hierarchy and what actions are available for the component
- **Input keys** - used to define the process model paths from which the components will receive their data

- **UI Actions** - actions defined here will be made available to the custom component


Custom
Identifier
Identifier
Input Keys
CustomComponent
Add an option
UI Action
Add UI action

Display of User Interface Elements

When a process instance is initiated, the web application receives all the UI elements that can be displayed in the process under the `templateConfig` key.

When a user task is reached in the process instance, the **events-gateway** receive requests, triggering it to display the associated UI element.

Example:

1. Starting a process:

- The following is an example of starting a process instance via a **POST** request to `{{processUrl}}/api/internal/process/DemoProcess/start`:


```
{
  "processDefinitionName" : "DemoProcess",
  "tokens" : [ {
    "id" : 662631,
    "startNodeId" : null,
    "currentNodeId" : 662807,
    "currentNodeName" : null,
    "state" : "ACTIVE",
    "statusCurrentNode" : "ARRIVED",
    "dateUpdated" : "2023-02-09T12:23:19.464155Z",
    "uuid" : "ae626fda-8166-49e8-823b-fe24f36524a7"
  } ],
  "state" : "CREATED",
  "templateConfig" : [ {
    "id" : 630831,
    "flowxUuid" : "80ea0a85-2b0b-442a-a123-2480c7aa2dce",
    "nodeDefinitionId" : 662856,
```

```
    "componentIdentifier" : "CONTAINER",
    "type" : "FLOWX",
    "order" : 1,
    "canGoBack" : true,
    "displayOptions" : {
      "flowxProps" : { },
      "style" : null,
      "flexLayout" : {
        "fxLayoutGap" : 0,
        "fxLayoutAlign" : "start stretch",
        "fxLayout" : "column"
      },
      "className" : null,
      "platform" : "DEFAULT"
    },
    "templateConfig" : [ {
      "id" : 630832,
      "flowxUuid" : "38e2c164-f8cd-4f6e-93c8-39b7cdd734cf",
      "nodeDefinitionId" : 662856,
      "uiTemplateParentId" : 630831,
      "componentIdentifier" : "TEXT",
      "type" : "FLOWX",
      "order" : 0,
      "key" : "",
      "canGoBack" : true,
      "displayOptions" : {
        "flowxProps" : {
          "text" : "Demo text"
        },
        "style" : null,
        "flexLayout" : null,
        "className" : null,
        "platform" : "DEFAULT"
      },
      "expressions" : {
```

```
        "hide" : ""
      },
      "templateConfig" : [ ],
      "dataSource" : {
        "processData" : {
          "parentFlowxUuid" : null
        },
        "nomenclator" : {
          "parentFlowxUuid" : null
        }
      }
    } ]
  } ],
  "uuid" : "d985d128-ae45-4408-a643-1dd026a644d3",
  "generalData" : null,
  "backCounter" : 0,
  "startedByActionId" : null,
  "subProcesses" : null,
  "subprocessesUuids" : null,
  "baseUrl" : null
}
```


2. The following is an example of a progress message:


```
{
  "progressUpdateDTO": {
    "processInstanceUuid": "5f24c66f-04a7-433a-b64a-
a765d3b8121a",
    "tokenUuid": "11c32ba6-b3e7-4267-9383-25d69b26492c",
    "currentNodeId": 662856

  }
}
```

3. **ProgressUpdateDto** will trigger the **SDK** to search for the UI element having the same **nodeId** as the one from the web socket progress event

4. Additionally, it will ask for data and actions that are required for this component via a GET request `{{processUrl}}/api/process/5f24c66f-04a7-433a-b64a-a765d3b8121aa/data/662856`

Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Collection / Collection Prototype

Description

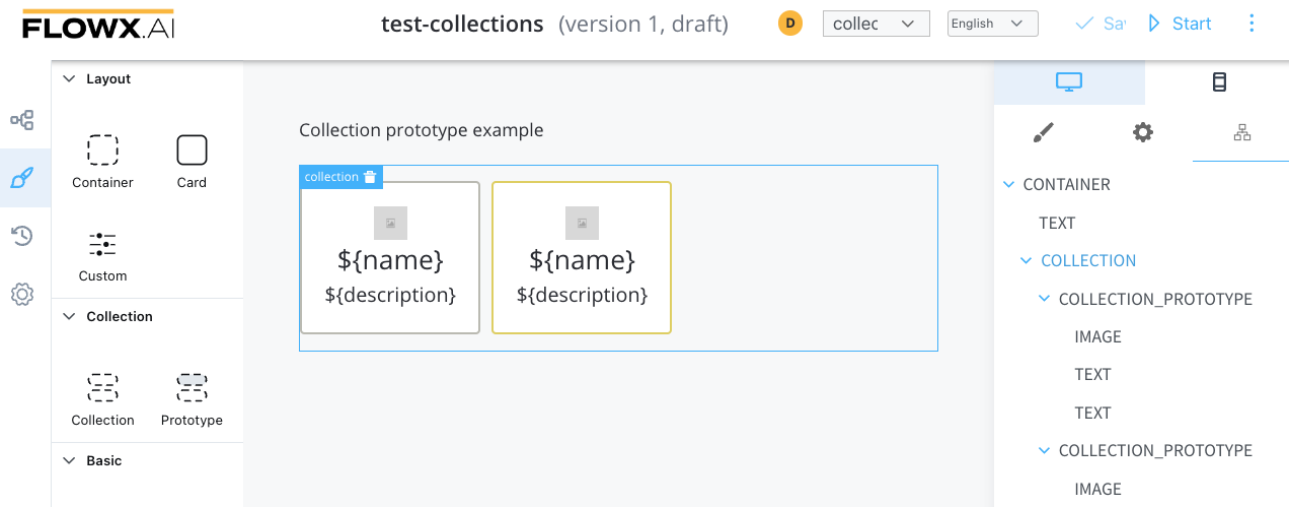
This additional container type is needed to allow multiple prototypes to be defined for a single Collection. This allows elements from the same collection to be displayed differently.

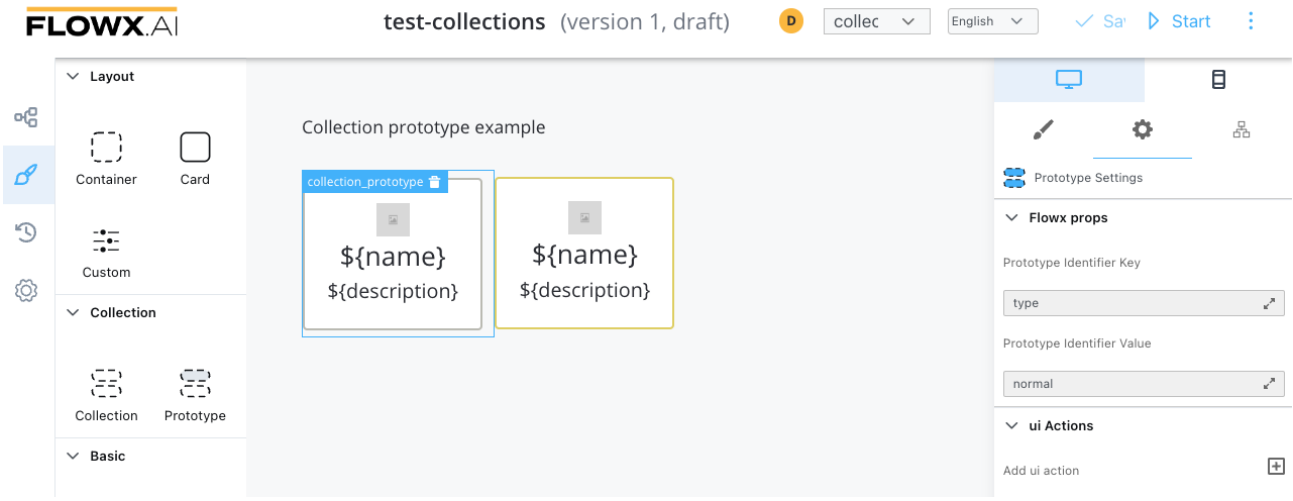
For example, suppose you are creating a piece of UI in which the user is presented a list of possible products from which to choose, but you want one of the products to be highlighted as the recommended one. This example requires a collection with two **collection prototypes** (one for the normal product and one for the recommended one).

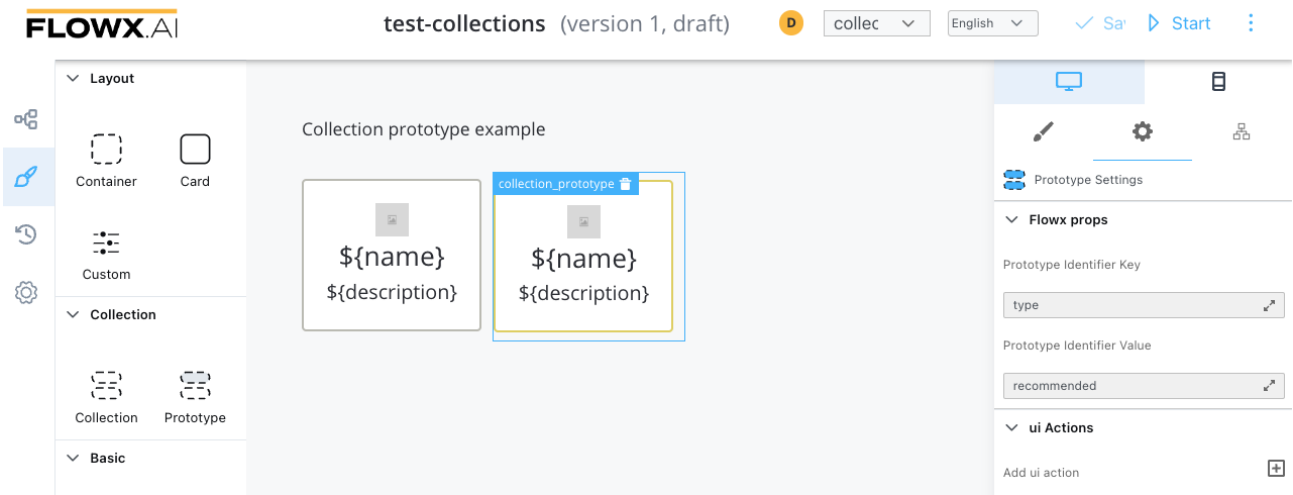
Configurable properties:

1. **Prototype Identifier Key** - the key where to look in the iterated object to determine the prototype to be shown - in the below example the key is "type"
2. Prototype Identifier Value - the value that should be present at the **Prototype Identifier Key** when this `COLLECTION_PROTOTYPE` should be displayed - in the below example the value is "normal" or "recommended"

Example





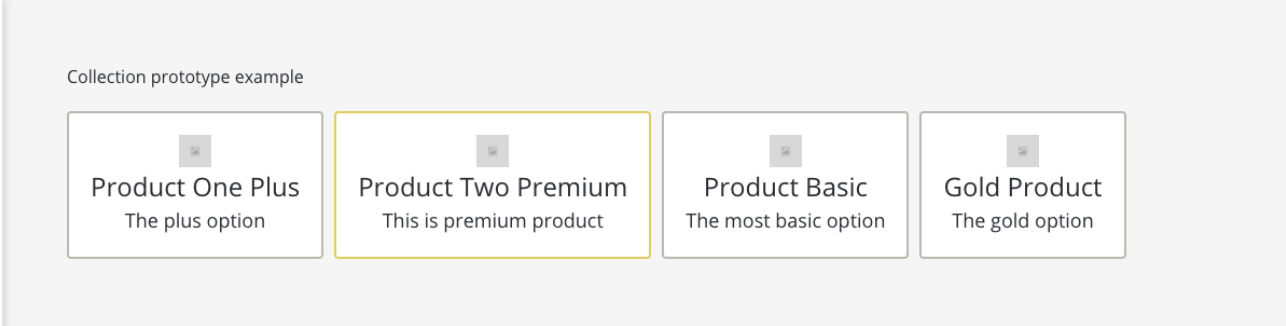


Source collection data example for products:

```
products: [
    {
      name: 'Product One Plus',
      description: 'The plus option'
      type: 'normal'
    },
    {
      name: 'Product Two Premium',
```

```
      description: 'This is premium product'
      type: 'recommended',
    },
    {
      name: 'Product Basic',
      description: 'The most basic option'
      type: 'normal'
    },
    {
      name: 'Gold Product',
      description: 'The gold option'
      type: 'normal',
    }
  ]
```

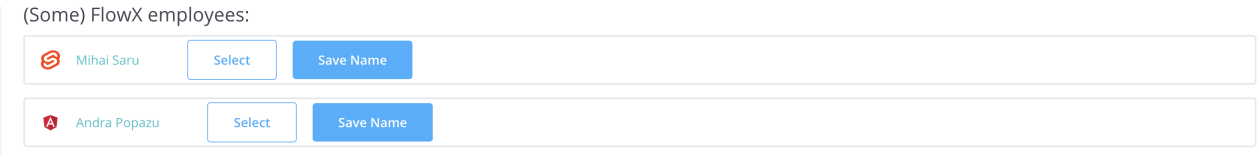
The above configuration will render:



Adding elements with UI Actions

There are a few differences you need to take into consideration when configuring elements that make use of **UI Actions** inside a **Collection Prototype**.

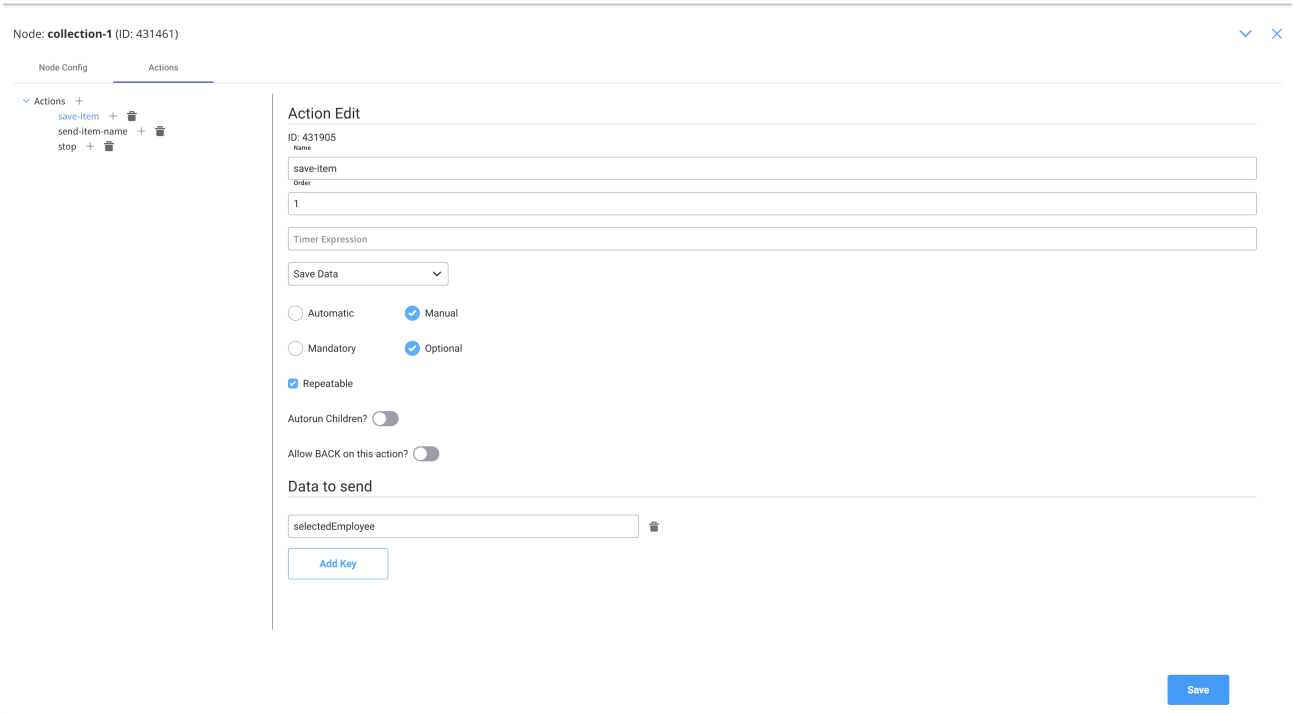
To showcase these differences, we'll use the next example:



We have a Collection with two employees and we want to provide the user with the option of selecting one of the employees (eg. to allow for further processing in the next steps of the process).

Step 1 - Defining the Node Action

To select one employee from the list, we first must add an Action to the User Task Node this UI is attached to:



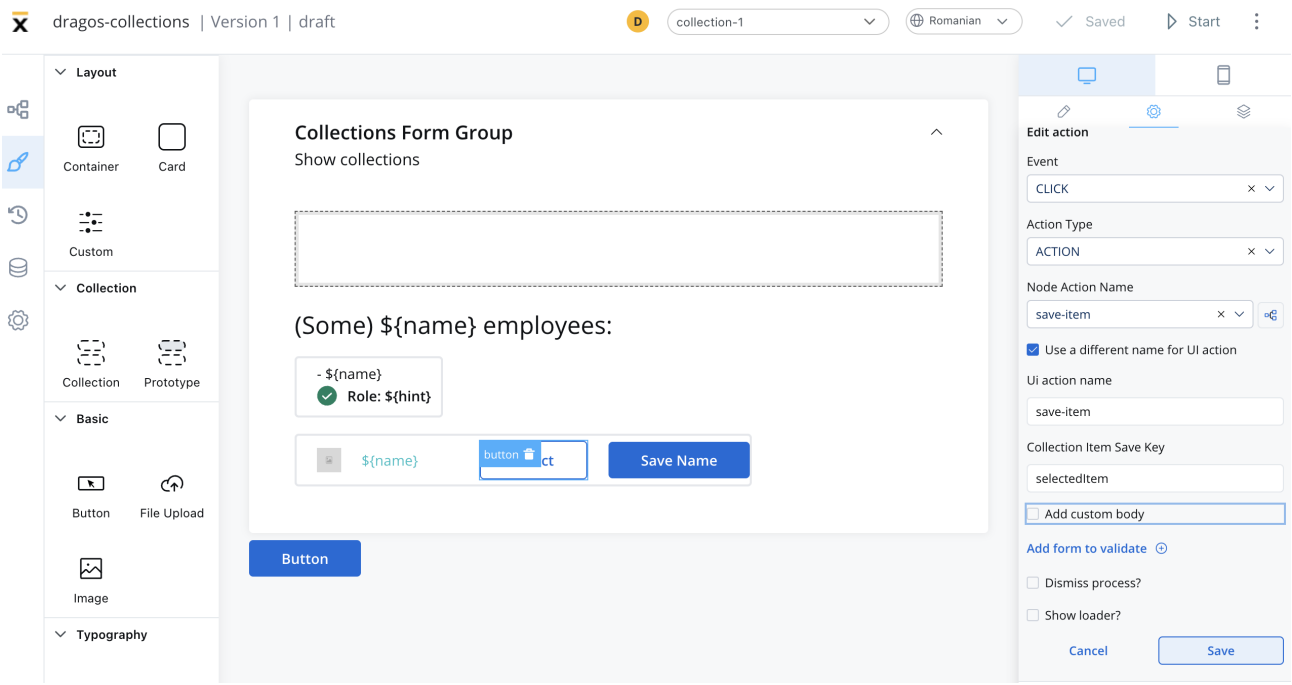
This **save-item** action is **manual** (since it will be triggered by the user) and **optionally** (since selecting an employee is not a requirement to go to the next Node in the process).

To allow the user to change his mind about the selected employee, this action is also marked as **Repeatable**.

Keep in mind to check the **Data to send** section. Here we are telling the platform where we want the selected employee (for which the user pressed the **Select** button) to be saved in the **process data**. In this example, we want it to be saved under the `selectedEmployee` key.

Step 2 - Adding the Button & UI Action

Now that we have a Node Action defined, we can go ahead and add the **Select** button in the UI of the User Task which contains the Employees Collection.



Collection Item Save Key field has an important role in the UI Action configuration of the **Select** button. This field represents how we pass the value of the **Employee** that the user has selected to the Node Action that we created in **Step 1**, named *save-item*.

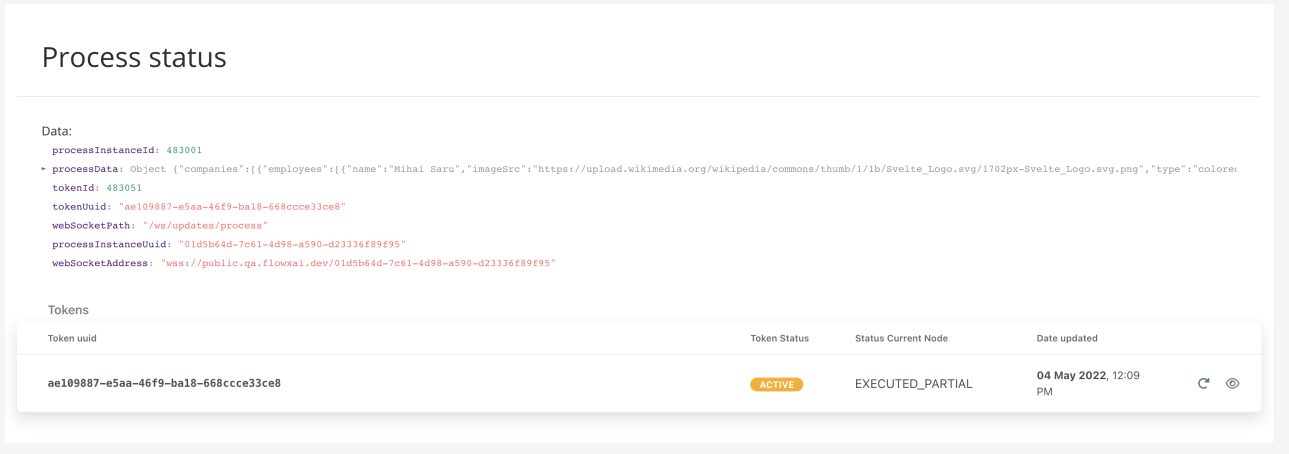
In our example, we set **Collection Item Save Key** to be `selectedEmployee`.

> **DANGER**
>
> **IMPORTANT:** `selectedEmployee` key is how we expose the data from the **Collection** to the Node Action It is **imperative** that the name in the **Collection Item Save Key** is the same as the one used in the **Data to send** input in the Node Action.

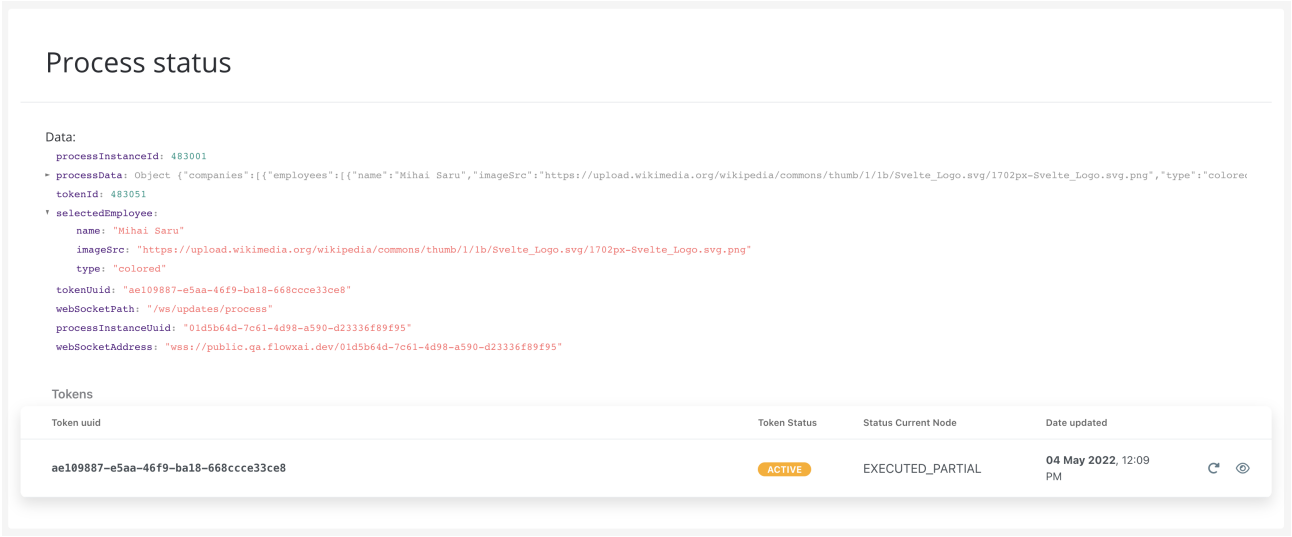
The button and UI action are mostly configured as any other Button and UI Action would be configured.

Result

This is how the process data looked before we pressed the **Select** button for an employee:



This is how the process data looks after we selected an employee from the list (notice the new field `selectedEmployee`):



Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Buttons

There are two types of buttons available, each with a different purpose. These types are:

- Basic button
- File upload button

Basic button

Basic buttons are used to perform an action such as unblocking a token to move forward in the process, sending an OTP, and opening a new tab.

Configuring a basic button

When configuring a basic button, you can customize the button's settings by using the following options:

- **Properties**
- **UI action**
- **Button styling**

Sections that can be configured regarding general settings:

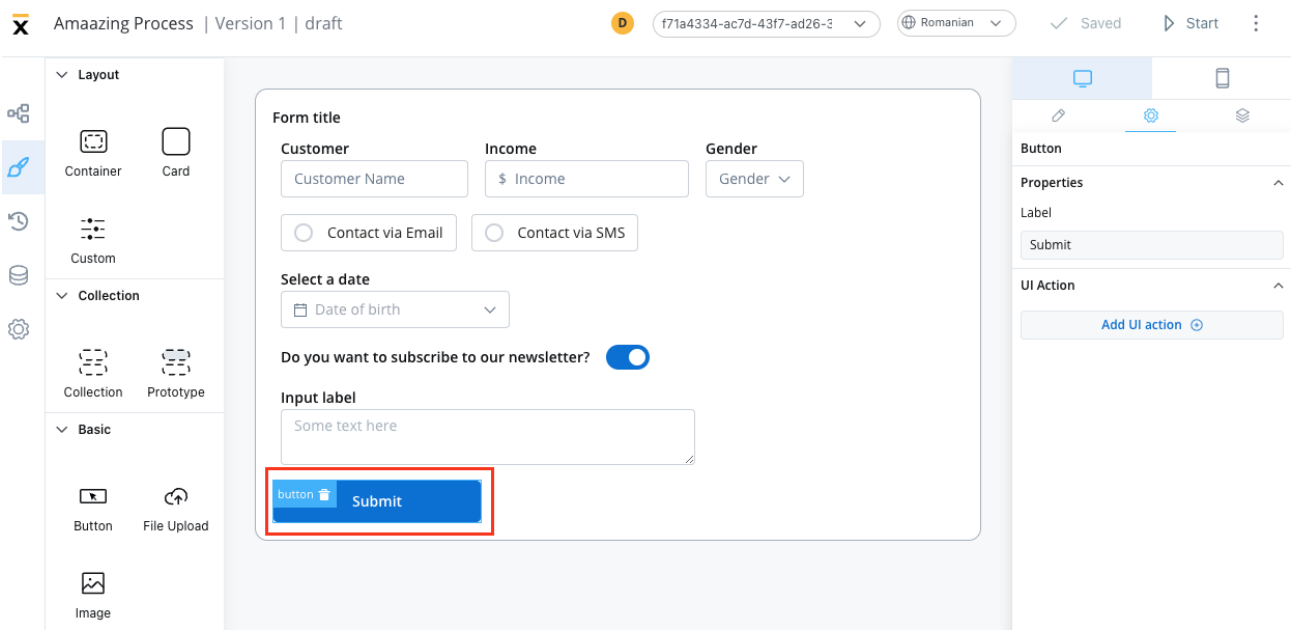
Properties

- **Label** - it allows you to set the label that appears on the button

UI action

Here, you can define the UI action that the button will trigger.

- **Event** - possible value: `CLICK`
- **Action Type** - select the action type



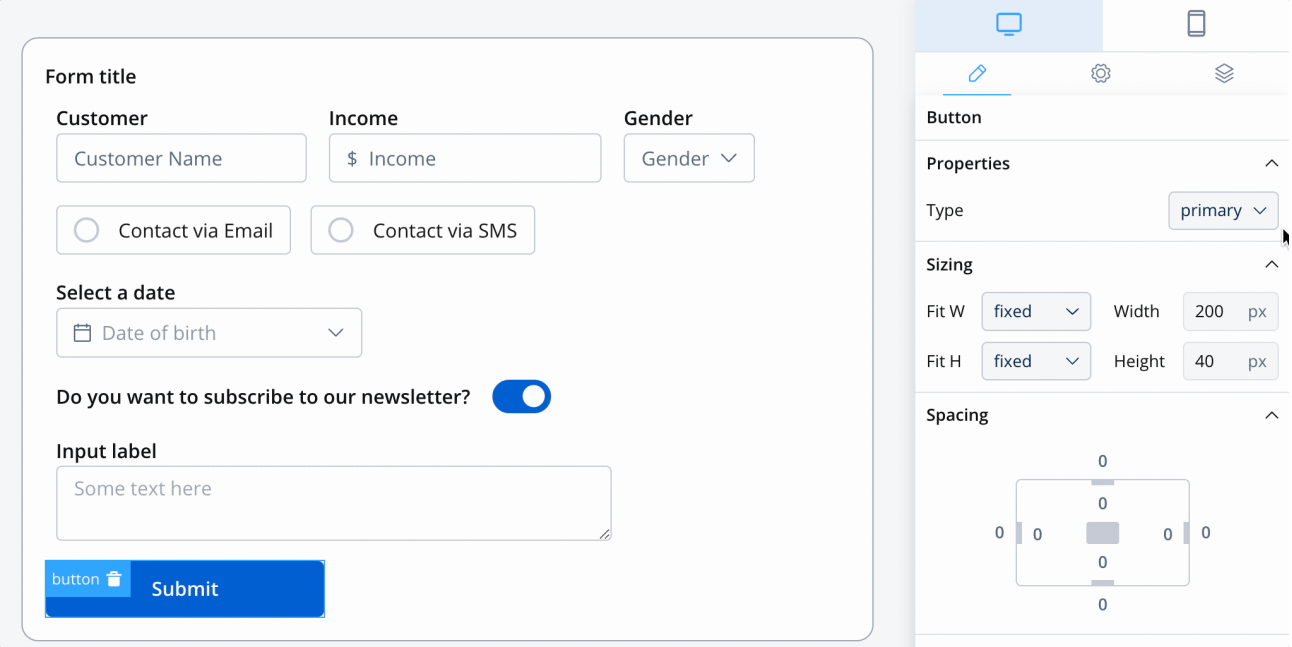
More details on how to configure UI actions can be found here.

Button styling

Properties

This section enables you to select the type of button using the styling tab in the UI Designer. There are four types available:

- Primary
- Secondary
- Ghost
- Text



> **INFO**
>
> For more information on valid CSS properties, click here

Icons

To further enhance the Button UI element with icons, you can include the following properties:

- **Icon Key** - the key associated in the Media library, select the icon from the **Media Library**
- **Icon Color** - select the color of the icon using the color picker

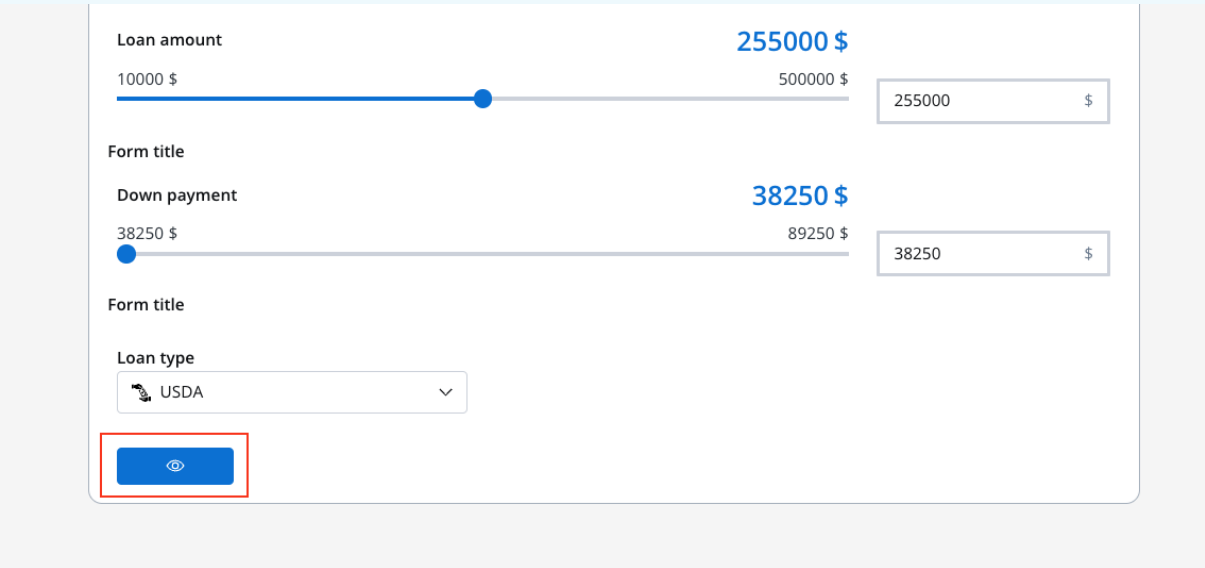
INFO

When setting the color, the entire icon is filled with that color, the SVG's fill. Avoid changing colors for multicolor icons.

- **Icon Position** - define the position of the icon within the button:c
  - Left
  - Right
  - Center

INFO

When selecting the center position for an icon, the button will display the icon only.



By utilizing these properties, you can create visually appealing Button UI elements with customizable icons, colors, and positions to effectively communicate their purpose and enhance the user experience.

File upload

This button will be used to select a file and do custom validation on it. Only the Flowx props will be different.

Configuring a file upload button

When configuring a file upload button, you can customize the button's settings by using the following options:

- **Properties**
- **UI action**
- **Button styling**

Sections that can be configured regarding general settings:

Properties

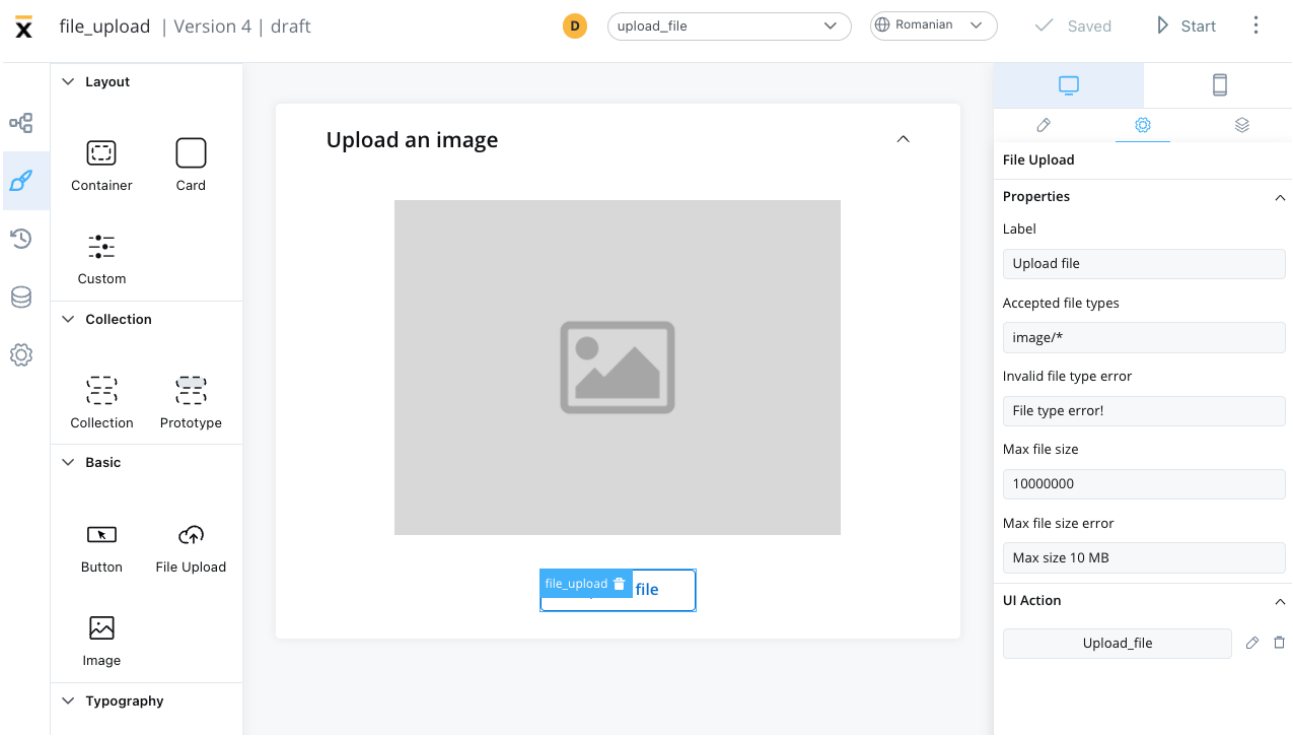
- **Label** - it allows you to set the label that appears on the button
- **Accepted file types** - the accept attribute takes as its value a string containing one or more of these unique file type specifiers, separated by commas, may take the following forms:

Value	Defintion
audio/*	Indicates that sound files are accepted

Value	Defintion
image/*	Indicates that image files are accepted
video/*	Indicates that video files are accepted
MIME type with no params	Indicates that files of the specified type are accepted
string starting with U+002E FULL STOP character (.) (for example, .doc, .docx, .xml)	Indicates that files with the specified file extension are accepted

- **Invalid file type error**
- **Max file size**
- **Max file size error**

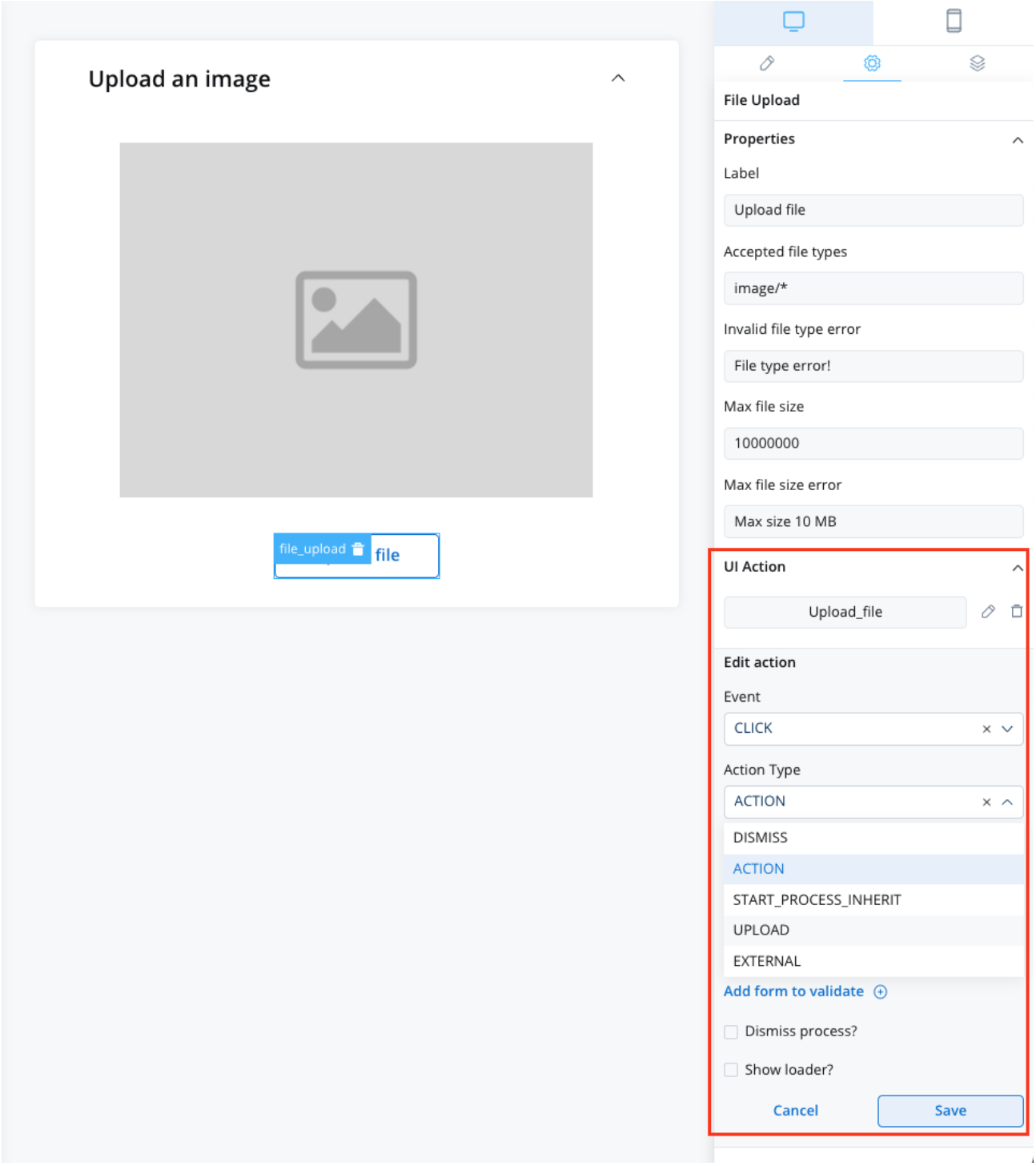
Example of an upload file button that accepts image files:



UI action

Here, you can define the UI action that the button will trigger.

- **Event** - possible value: `CLICK`
- **Action Type** - select the action type



INFO

More details on how to configure UI actions can be found here.

Button styling

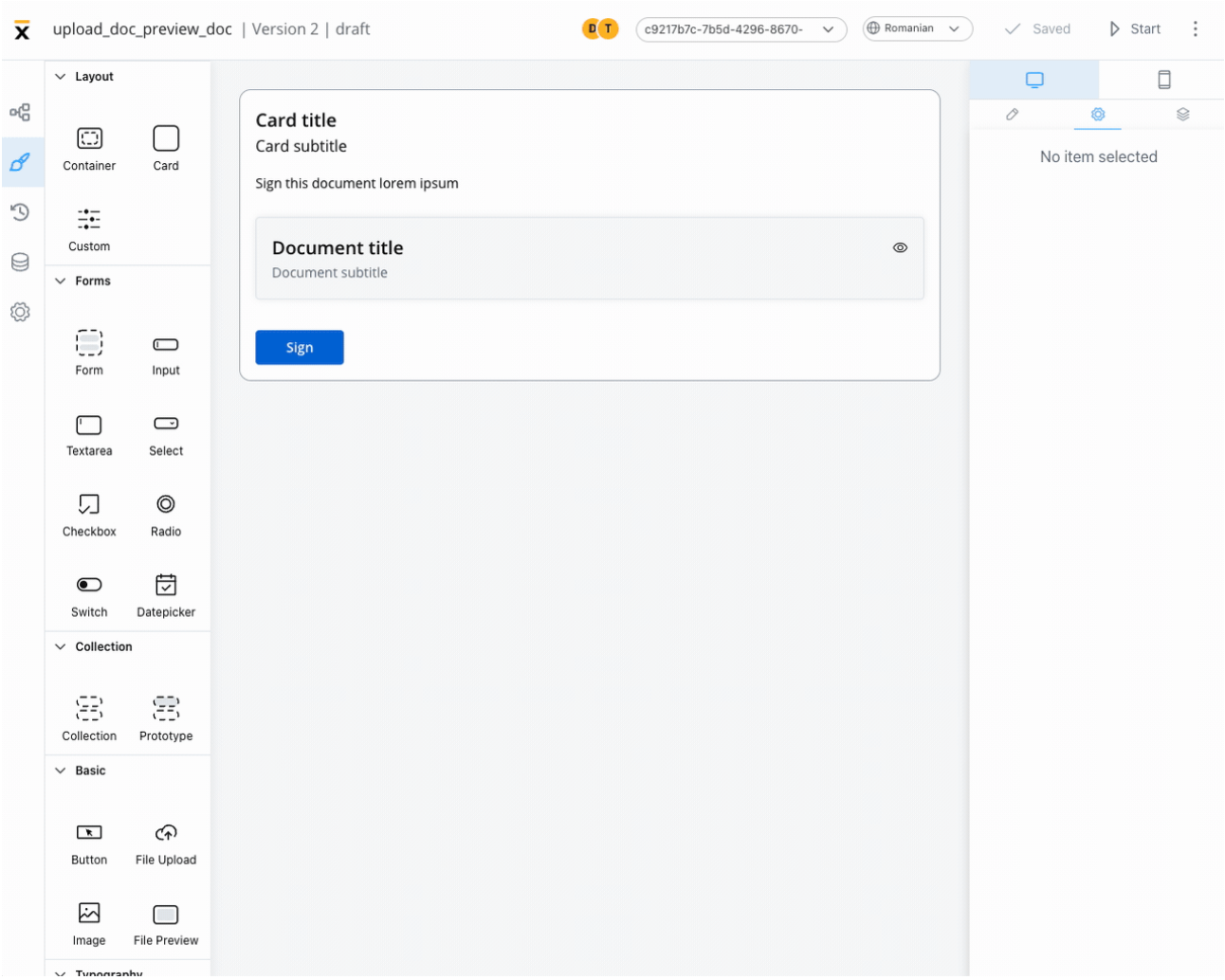
The file upload button can be styled using valid CSS properties (more details here

Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / File Preview

What is a File Preview UI element?

The File Preview UI element is a user interface component that enables users to preview the contents of files quickly and easily without fully opening them. It can save time and enhance productivity, providing a glimpse of what's inside a file without having to launch it entirely.

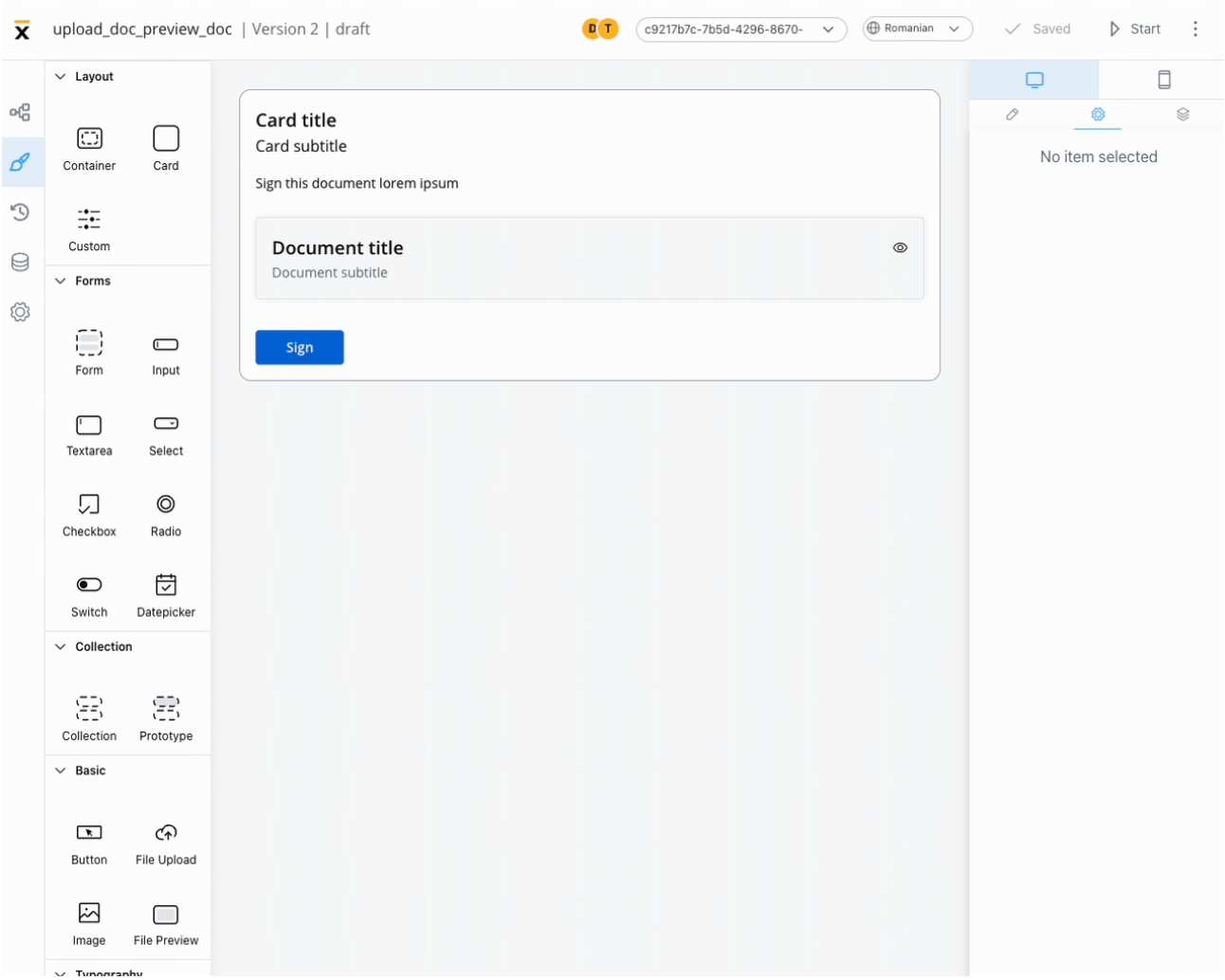


File Preview UI elements offer various benefits such as conveying information, improving the aesthetic appeal of an interface, providing visual cues and feedback or presenting complex data or concepts in a more accessible way.

Configuring a File Preview element

A File Preview element can be configured for both mobile and web applications.

File Preview properties (web)



The File Preview element settings consist of the following properties:

- **Title** - the title of the element (if it is downloaded or shared - the file name should be the title used in preview component)
- **Has subtitle** - the subtitle of the element
- **Display mode** - depending on the selected display method the following properties are available:
    - **Inline → Has accordion**:

    - `false` - display preview inline, without expand/collapse option
    - `true` - Default View: Collapsed - display preview inline, with expand/collapse option, by default collapsed
    - `true` - Default View: Expanded - display preview inline, with expand/collapse option, by default expanded
  - **Modal** → view icon is enabled

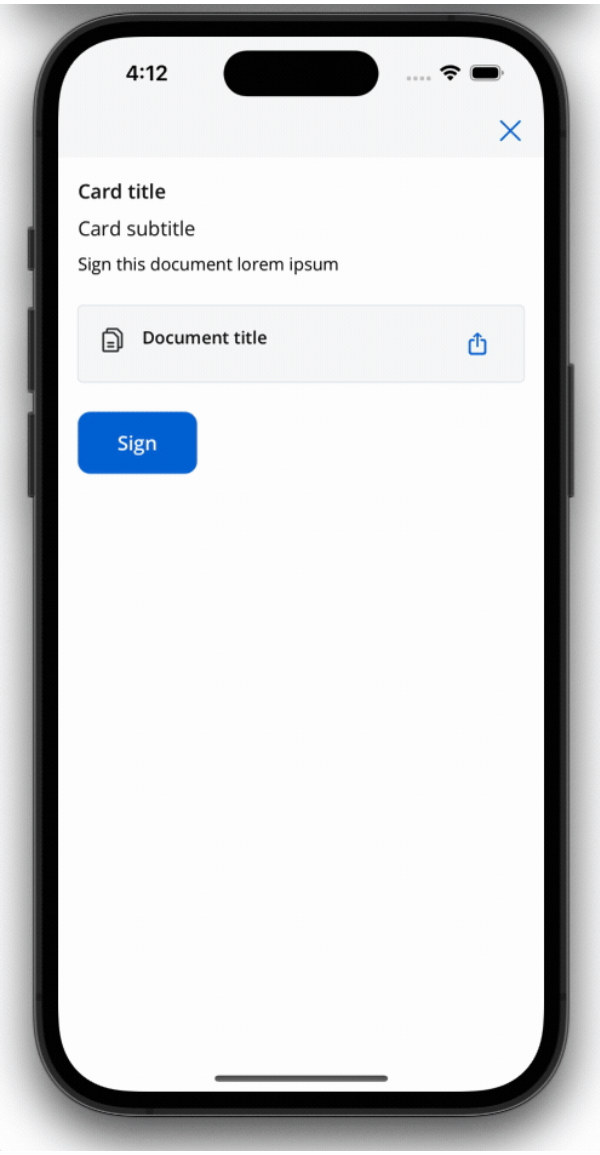
- **Source Type** -
  - **Process Data** - process key where the document is found (creates the binding between the element and process data)
  - **Static** - URL of the document

> **CAUTION**
>
> It's worth noting that the inline modal view can raise accessibility issues if the file preview's height exceeds the screen height.

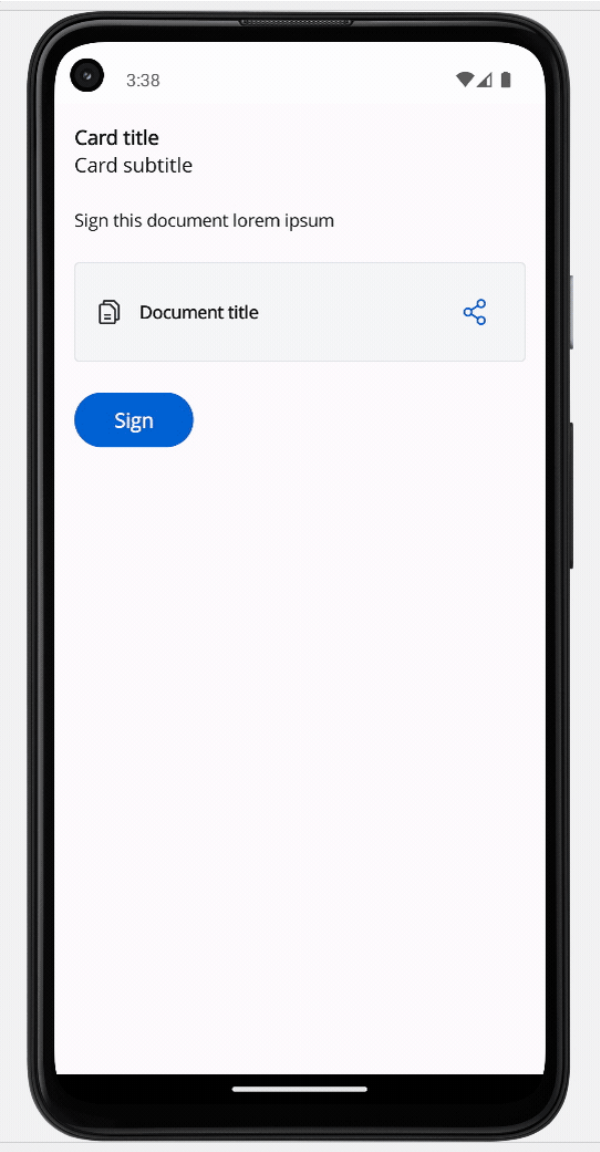
File Preview properties (mobile)

> **INFO**
>
> Both iOS and Android devices support the share button.

iOS


4:12
Card title
Card subtitle
Sign this document lorem ipsum
Document title
Sign

Android



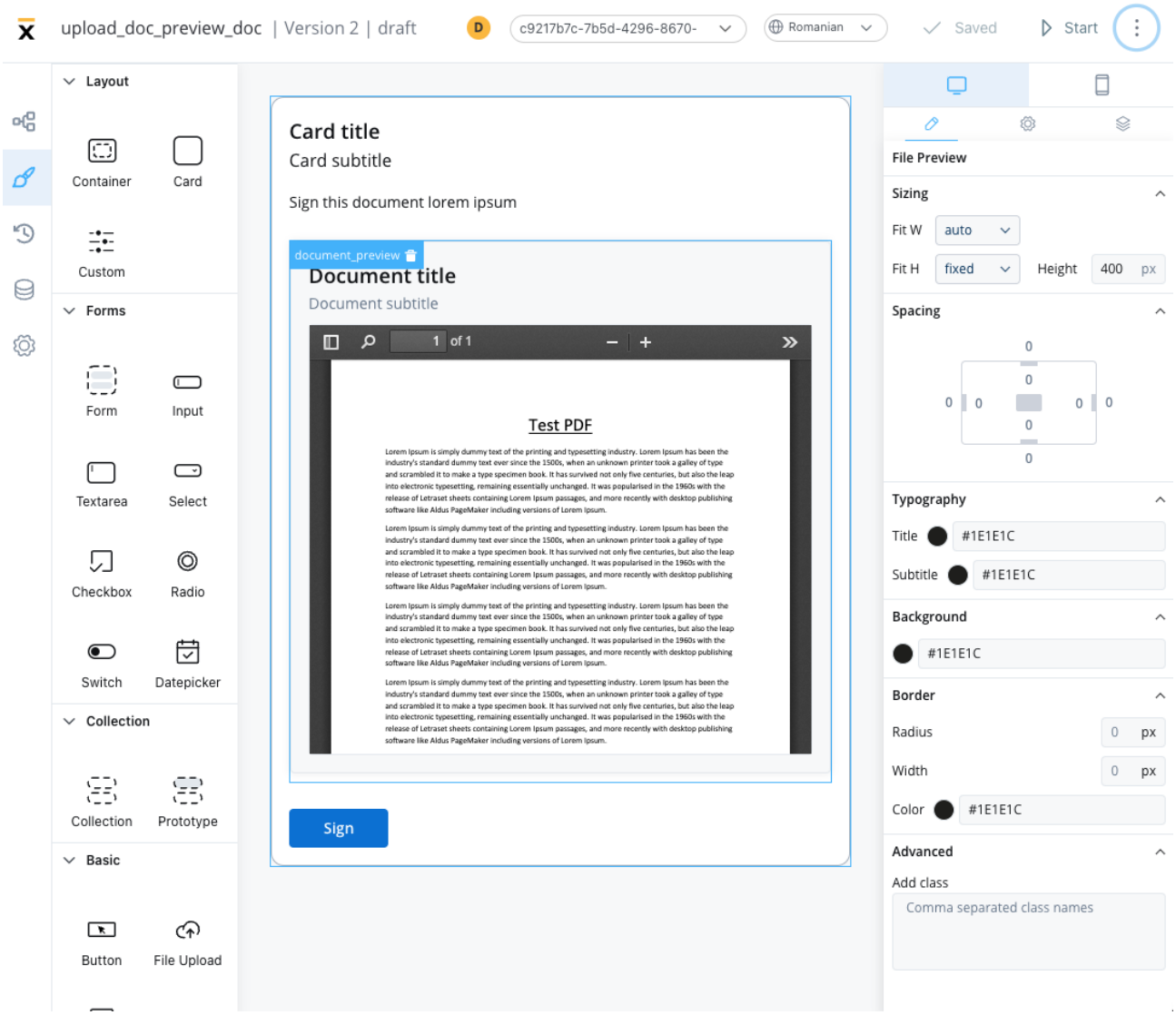
File preview styling

The File Preview styling property enables you to customize the appearance of the element by adding valid CSS properties, for more details, click here.

When drag and drop a File Preview element in UI Designer, it comes with the following default styling properties:

Sizing

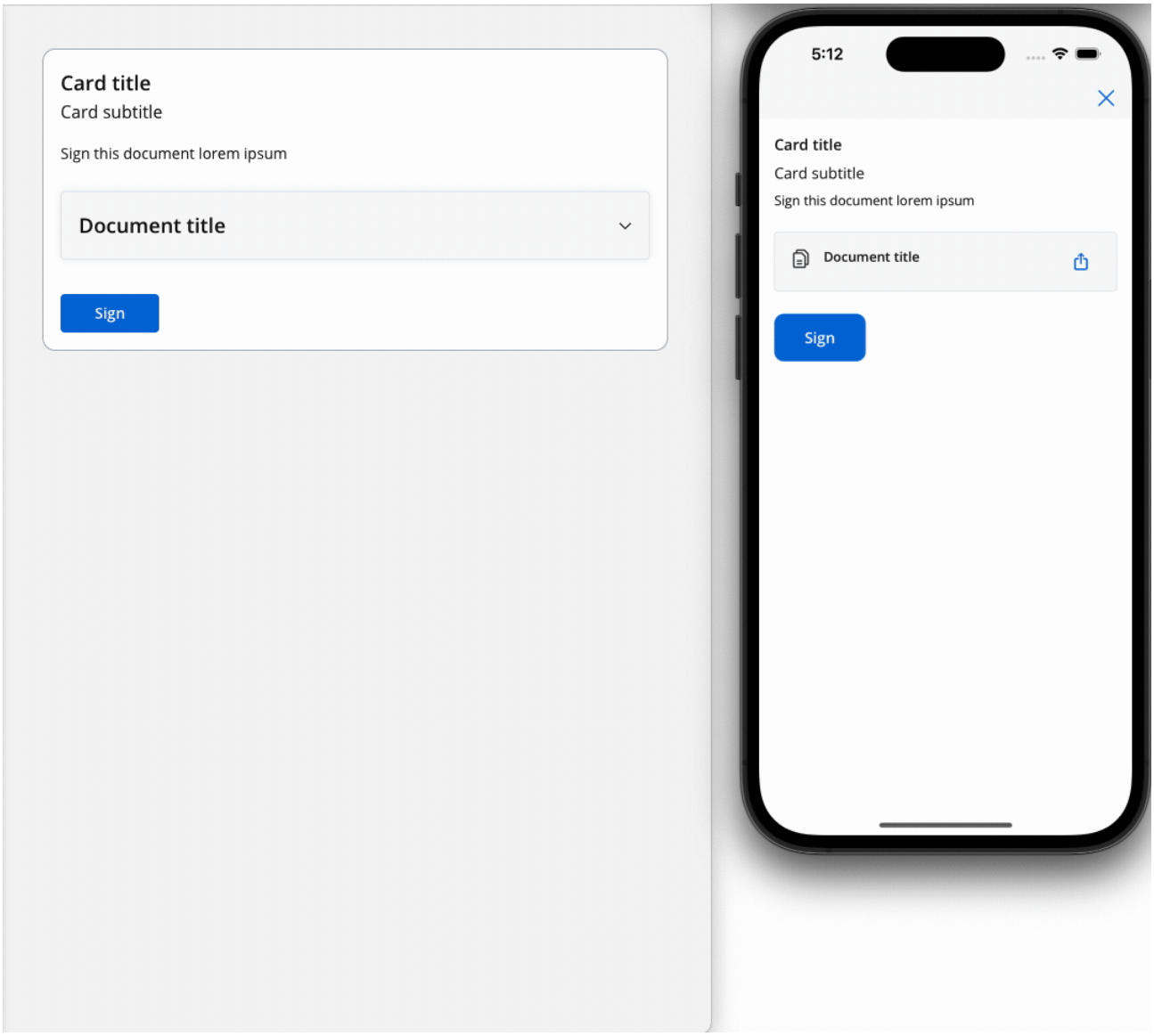
- **Fit W** - auto
- **Fit H** - fixed / Height - 400 px



File Preview example

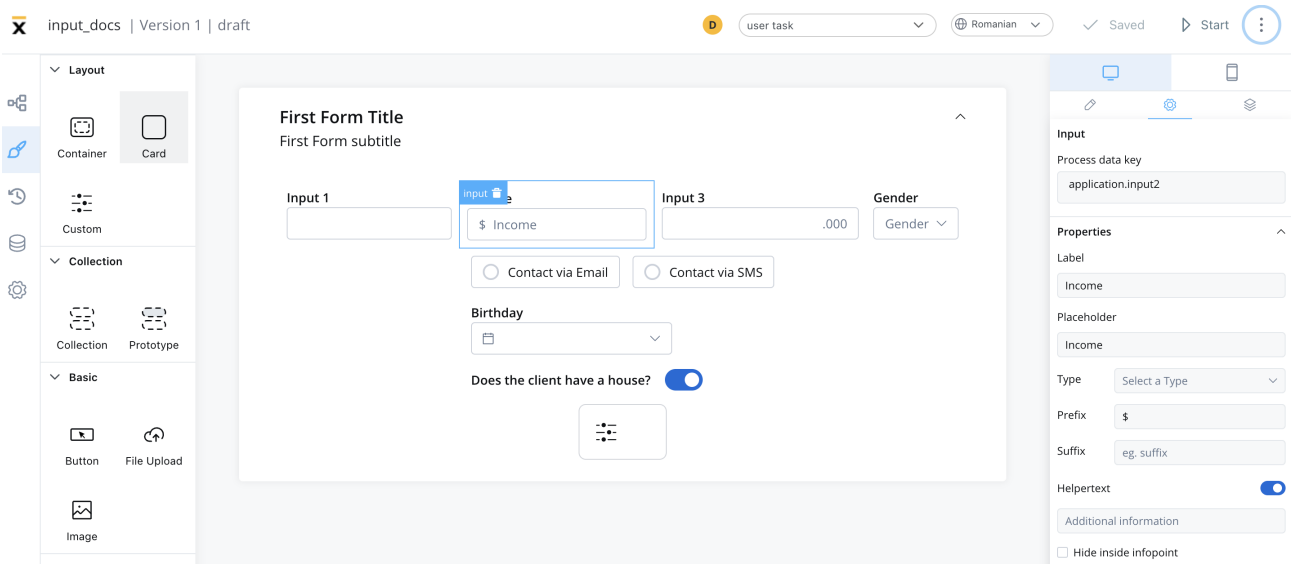
Below is an example of a File Preview UI element with the following properties:

- **Display mode** - Inline
- **Has accordion** - True
- **Default view** - Expanded
- **Source Type** - Static



Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Form elements / Input



An input field is a form element that enables users to input data with validations and can be hidden or disabled.

Configuring the input element

Input settings

The Input Field offers the following configuration options:

- **General**
- **Properties**
- **Datasource**

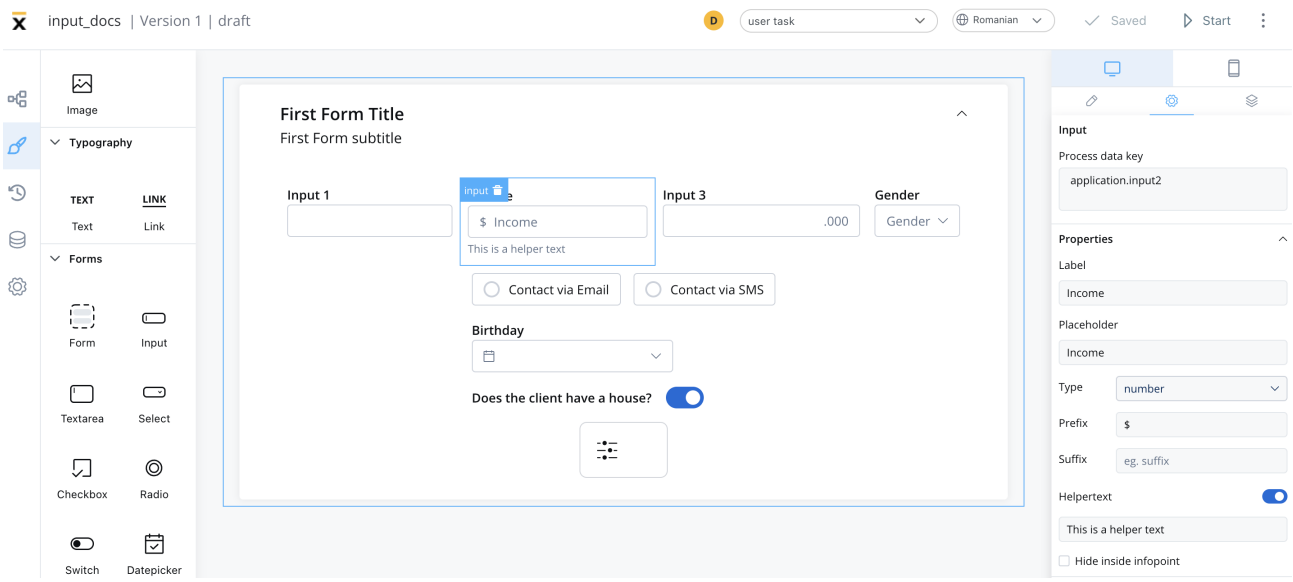
- **Validators**
- **Expressions**
- **UI actions**
- **Input styling**

General

- **Process data key** - creates the binding between form element and process data, so it can be later used in decisions, business rules or integrations

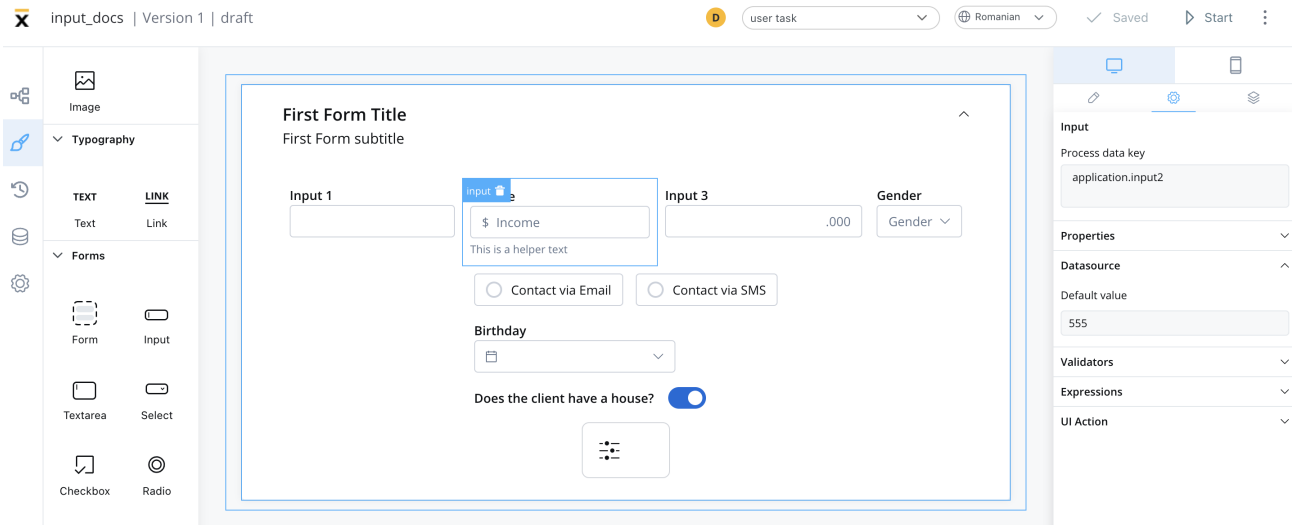
Properties

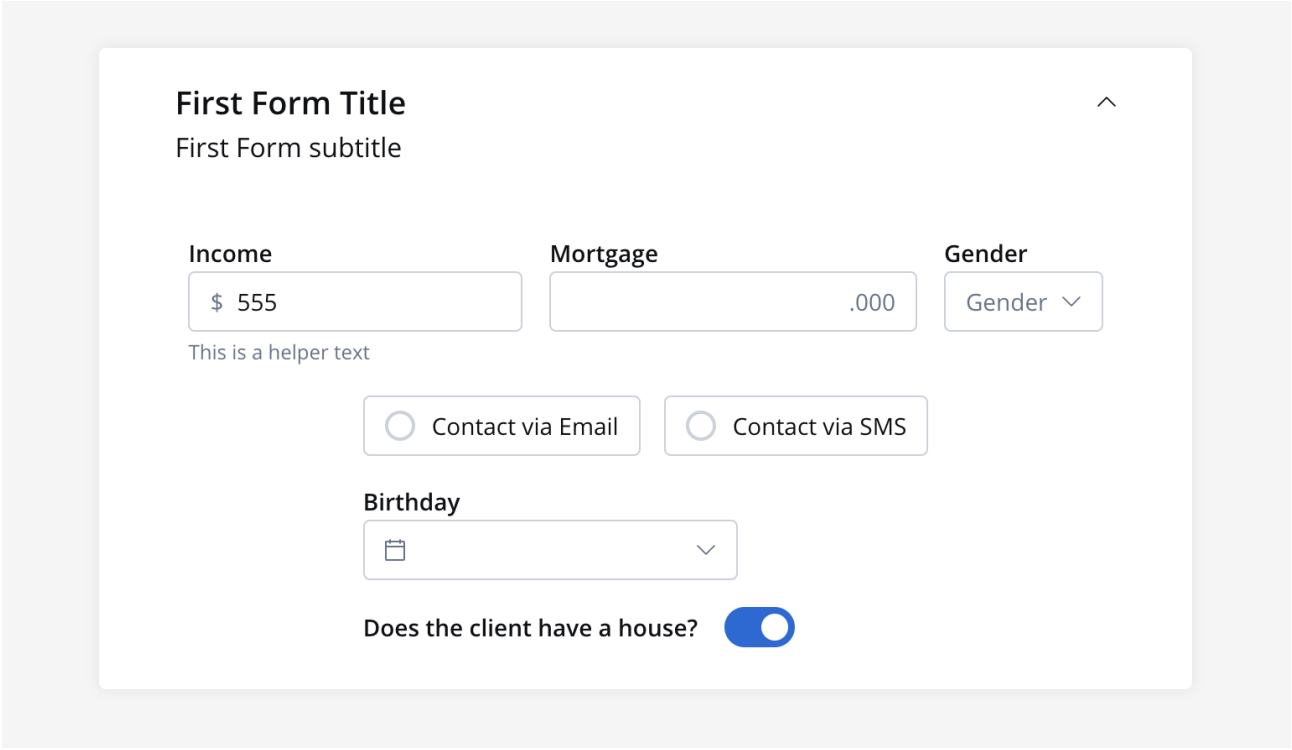
- **Label** - the label that appears on the input field
- **Placeholder** - the placeholder text that appears in the input field when it is empty
- **Type** - the type of data that the input field can accept, such as text, number, email, or password
- **Prefix** - a label that appears as a prefix to the input field
- **Suffix** - a label that appears as a suffix to the input field
- **Helpertext** - additional information about the input field (can be hidden inside an infopoint)



Datasource

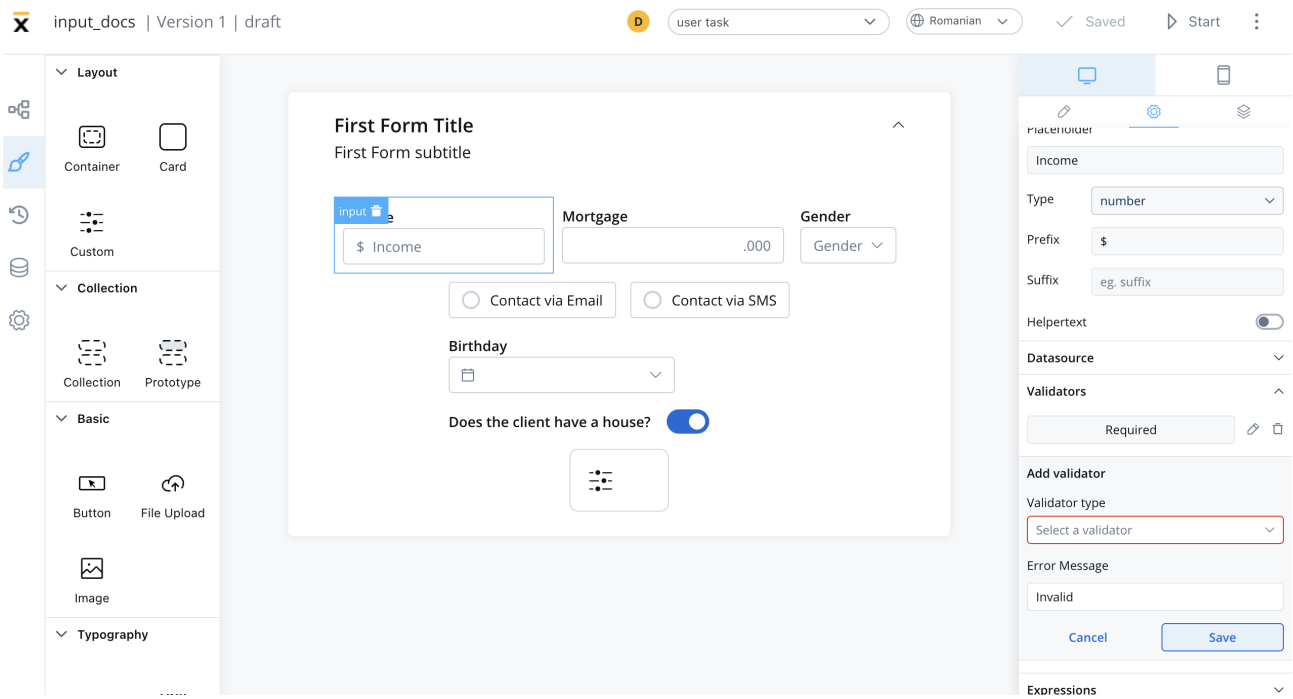
The default value for the element can be configured here, this will autofill the input field when you will run the process.





Validators

There are multiple validators can be added to an input (more details here).

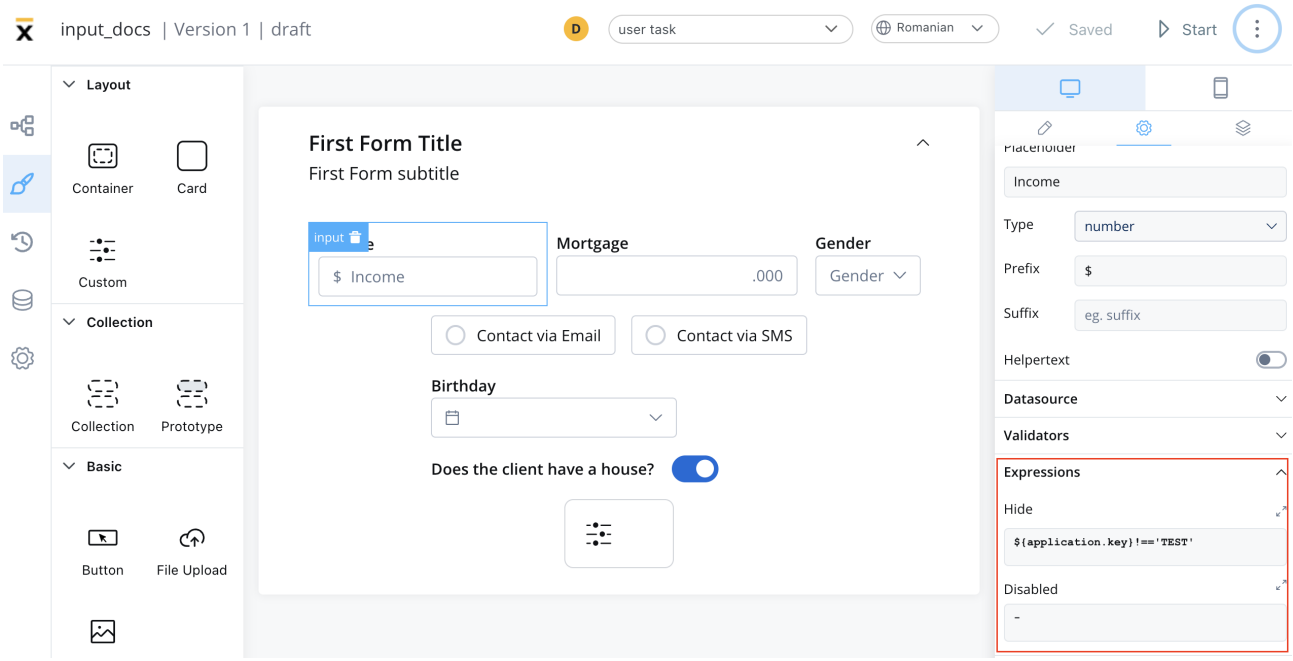


Expressions

The input field's behavior can be defined using JavaScript expressions for hiding or disabling the element. The following properties can be configured for expressions:

- **Hide** - JavaScript expression used to hide the Input Field when it returns a truthy value
- **Disabled** - JavaScript expression used to disable the Input Field when it returns a truthy value

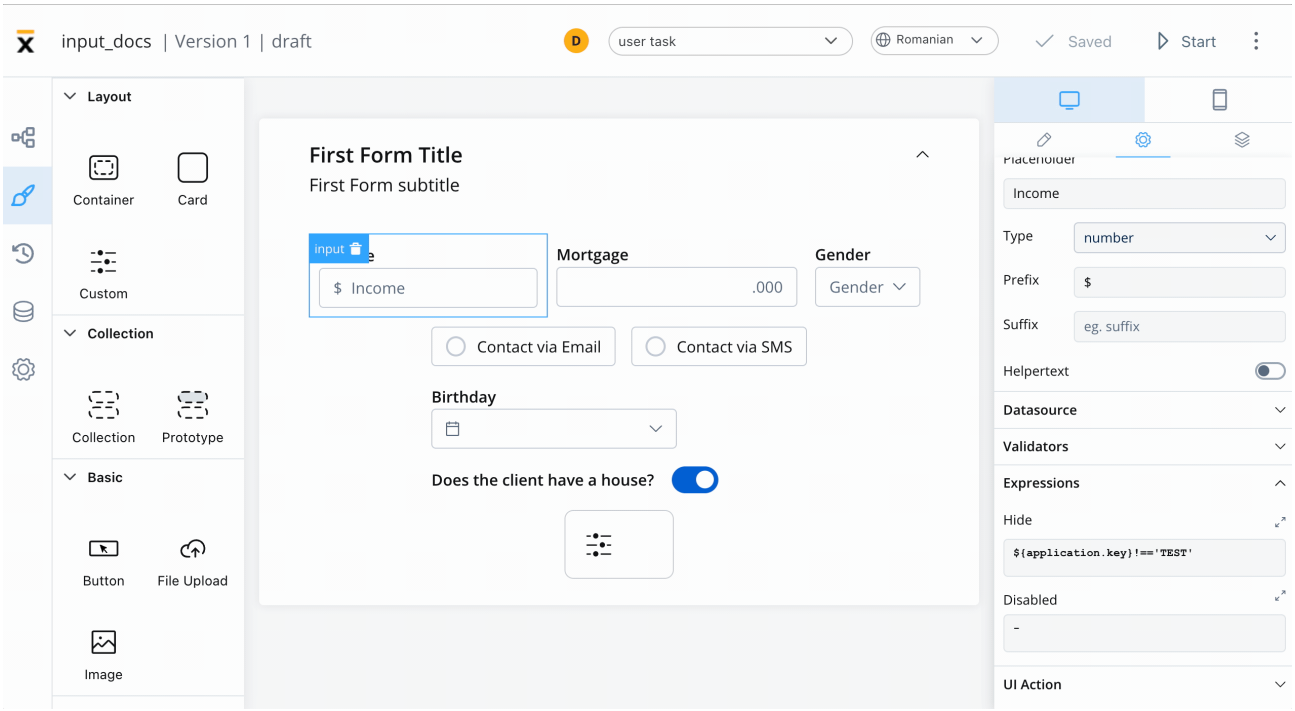
> **INFO**
>
> It's important to make sure that disabled fields have the same expression configured under the path expressions → hide.



UI actions

UI actions can be added to the Input Field to define its behavior and interactions.

- **Event** - possible value: `CHANGE`
- **Action Type** - select the action type



> **INFO**
>
> For more details on how to configure a UI action, click **here**.

Input styling

Icons

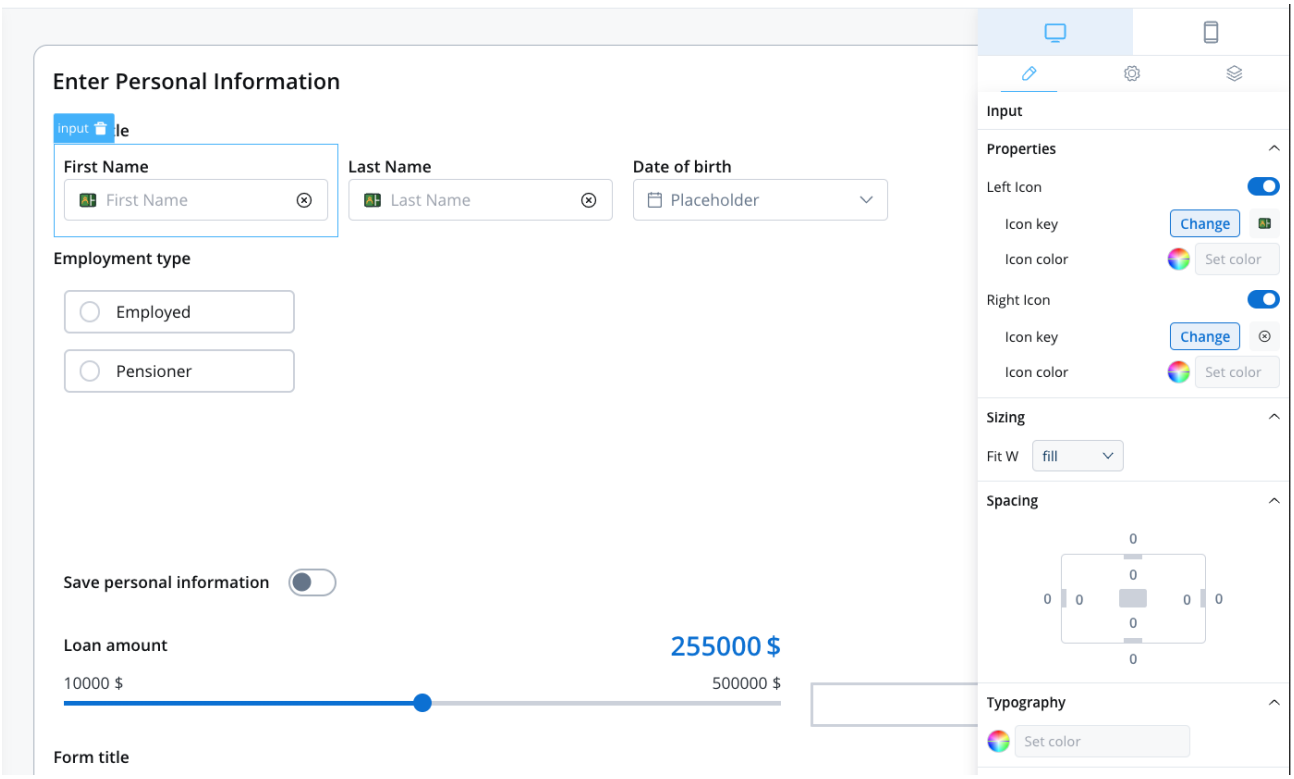
- **Icon Key** - the key associated in the Media library, select the icon from the **Media Library**
- **Icon Color** - select the color of the icon using the color picker

> **INFO**
>
> When setting the color, the entire icon is filled with that color, the SVG's fill. Avoid changing colors for multicolor icons.

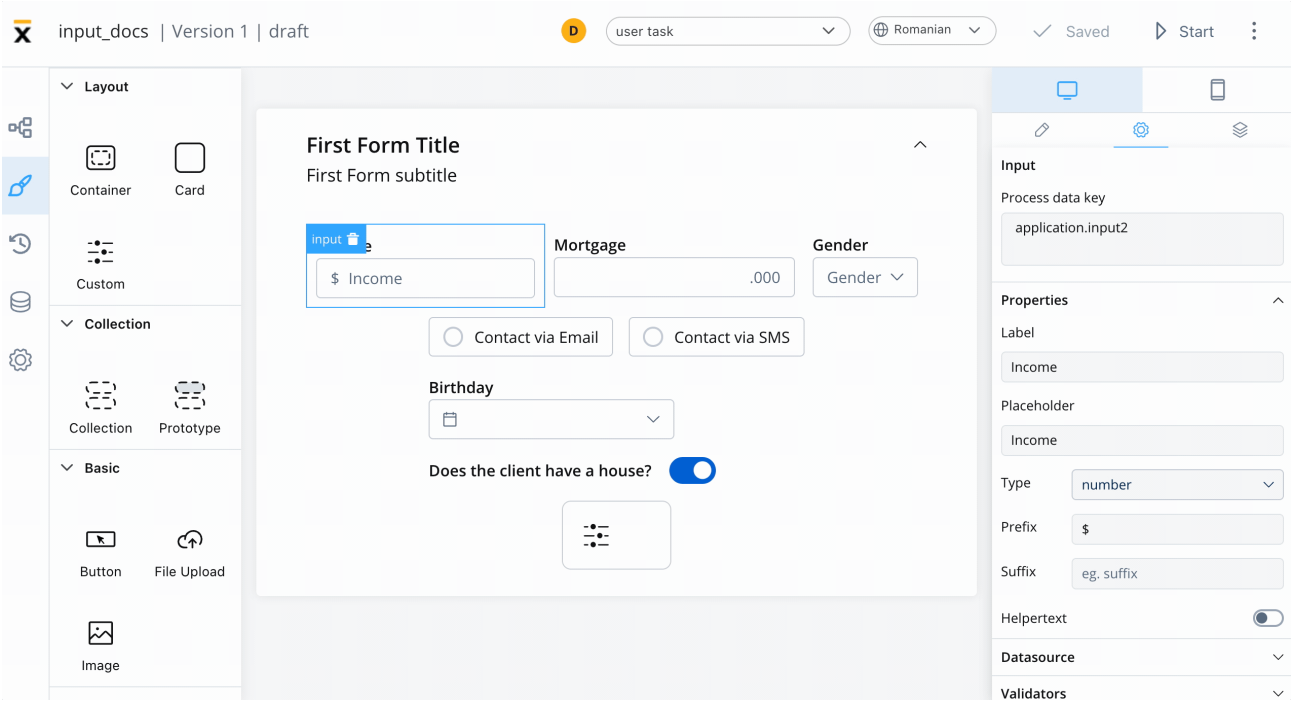
You have the option to enhance the Input element by incorporating two types of icons:

- **Left Icon**: You can include an icon on the left side of the Input element. This icon can serve as a visual cue or symbol associated with the input field's purpose or content.
- **Right Icon**: Same as left icon.

By utilizing these two types of icons, you can provide users with a more intuitive and visually appealing experience when interacting with the Input element.

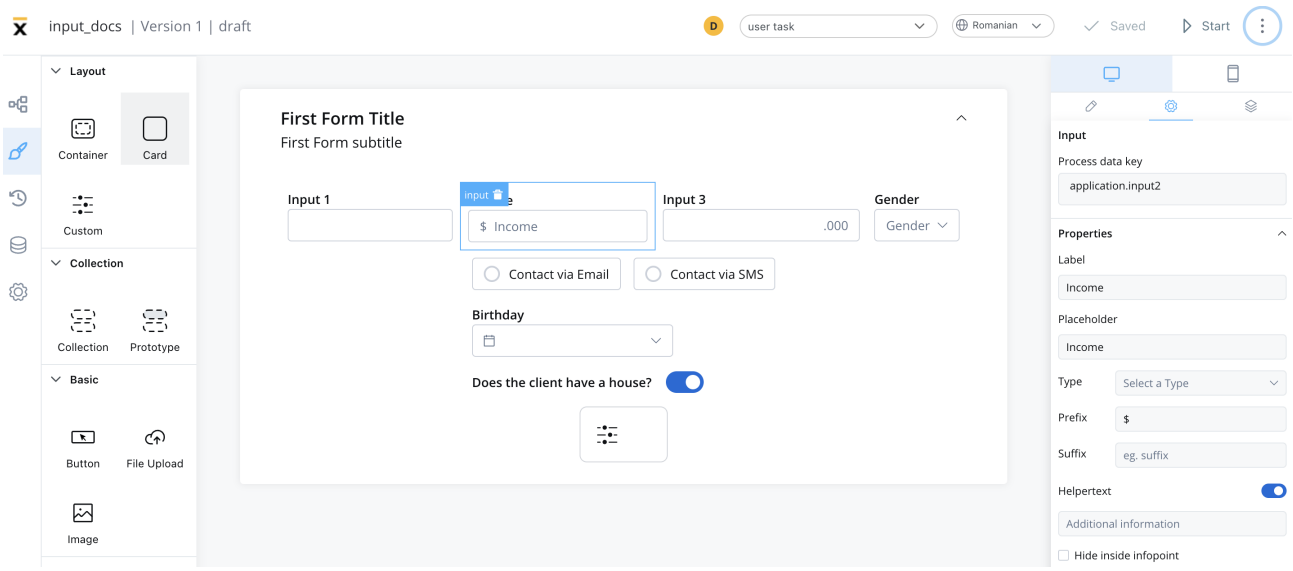


- The Input Field can be styled using valid CSS properties (more details here)



Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Form elements / Text area



A text area is a form element used to capture multi-line input from users in a conversational interface. The text area component is typically used for longer inputs such as descriptions, comments, or feedback, providing users with more space to type their responses.

It is an important tool for creating intuitive and effective conversational interfaces that can collect and process large amounts of user input.

Configuring the text area element

Text area settings

The text area offers the following configuration options:

- **General**
- **Properties**
- **Datasource**
- **Validators**

- **Expressions**
- **UI actions**
- **Text area styling**

General

- **Process data key** - creates the binding between form element and process data, so it can be later used in decisions, business rules or integrations

Properties

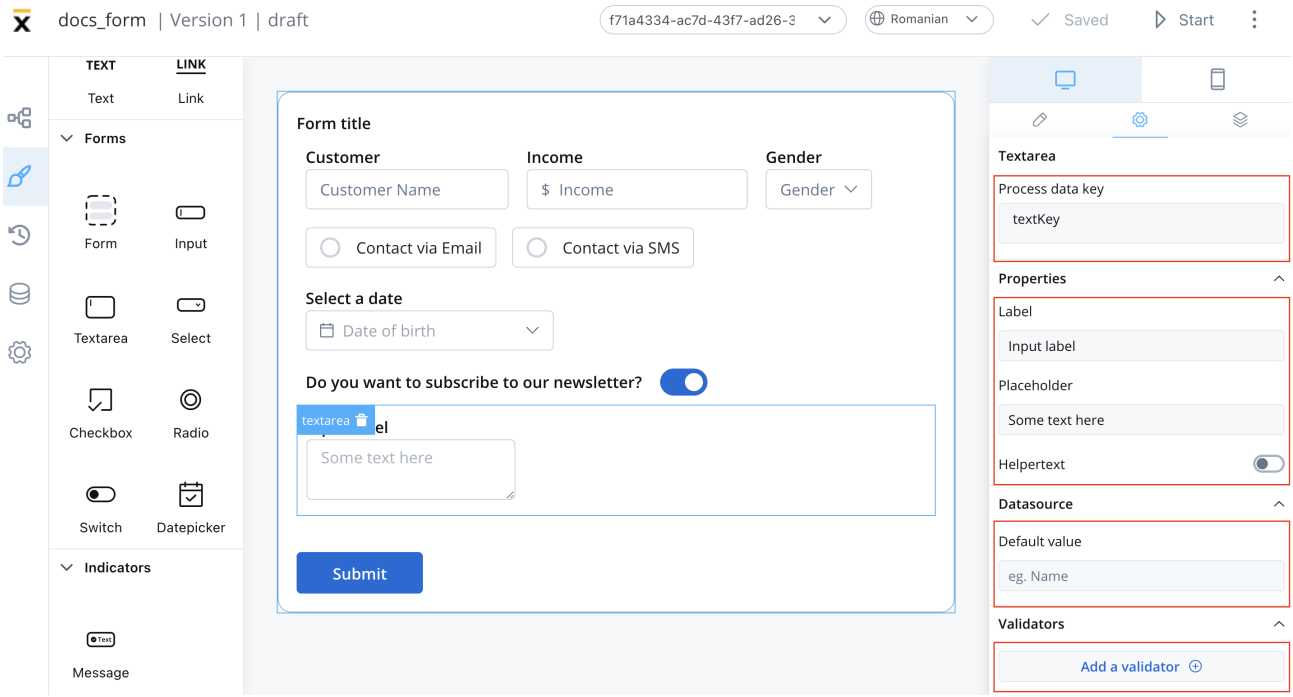
- **Label** - the label of the text area
- **Placeholder** - the placeholder text that appears in the text area
- **Helpertext** - additional information about the text area field (can be hidden inside an infopoint)

Datasource

The default value for the element can be configured here, this will autofill the text field when you will run the process.

Validators

There are multiple validators can be added to a text area element (more details here).

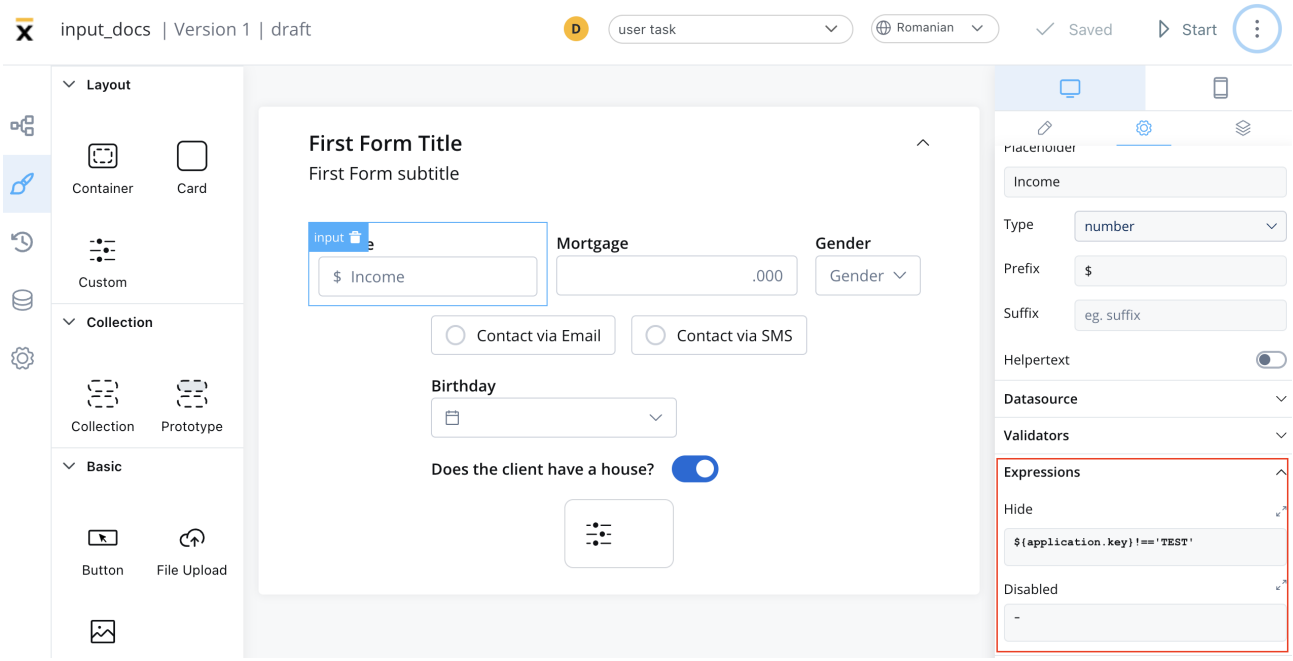


Expressions

The text area's behavior can be defined using JavaScript expressions for hiding or disabling the element. The following properties can be configured for expressions:

- **Hide** - JavaScript expression used to hide the text area when it returns a truthy value
- **Disabled** - JavaScript expression used to disable the text area when it returns a truthy value

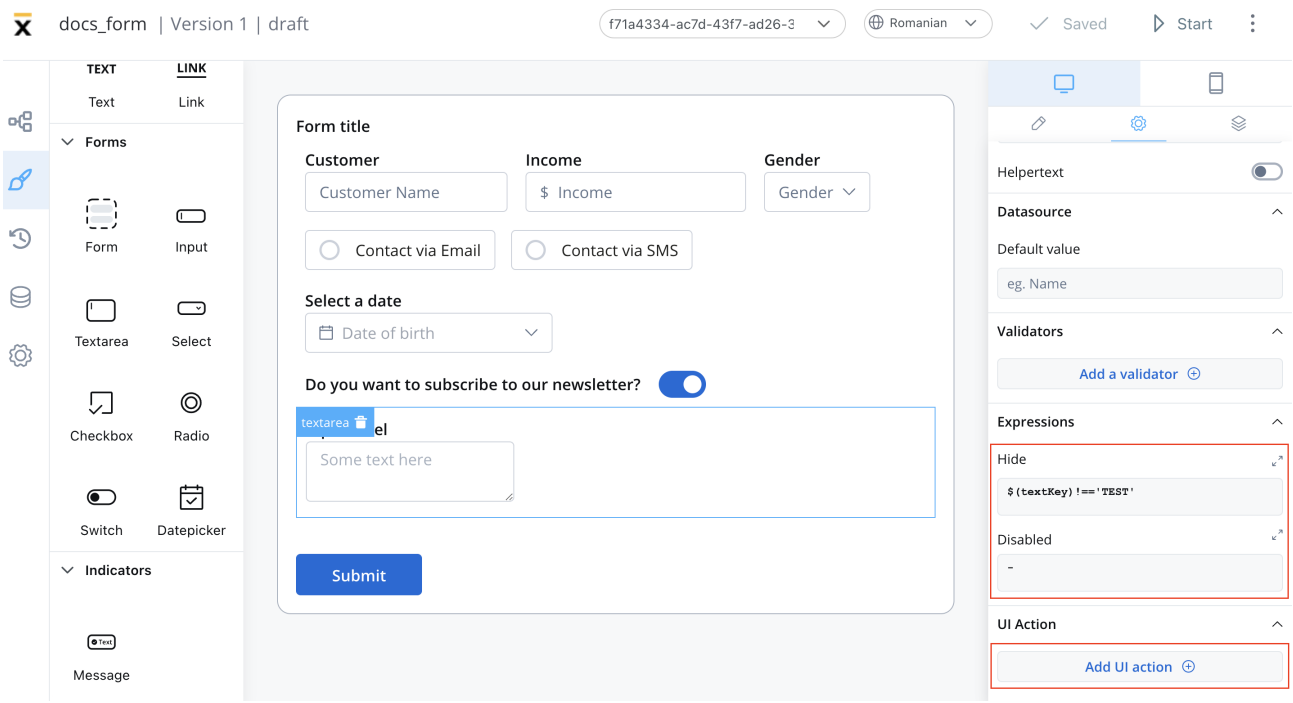
> **INFO**
>
> It's important to make sure that disabled fields have the same expression configured under the path expressions → hide.



UI actions

UI actions can be added to the text area field to define its behavior and interactions.

- **Event** - possible value: `CHANGE`
- **Action Type** - select the action type



> **INFO**
>
> For more details on how to configure a UI action, click **here**.

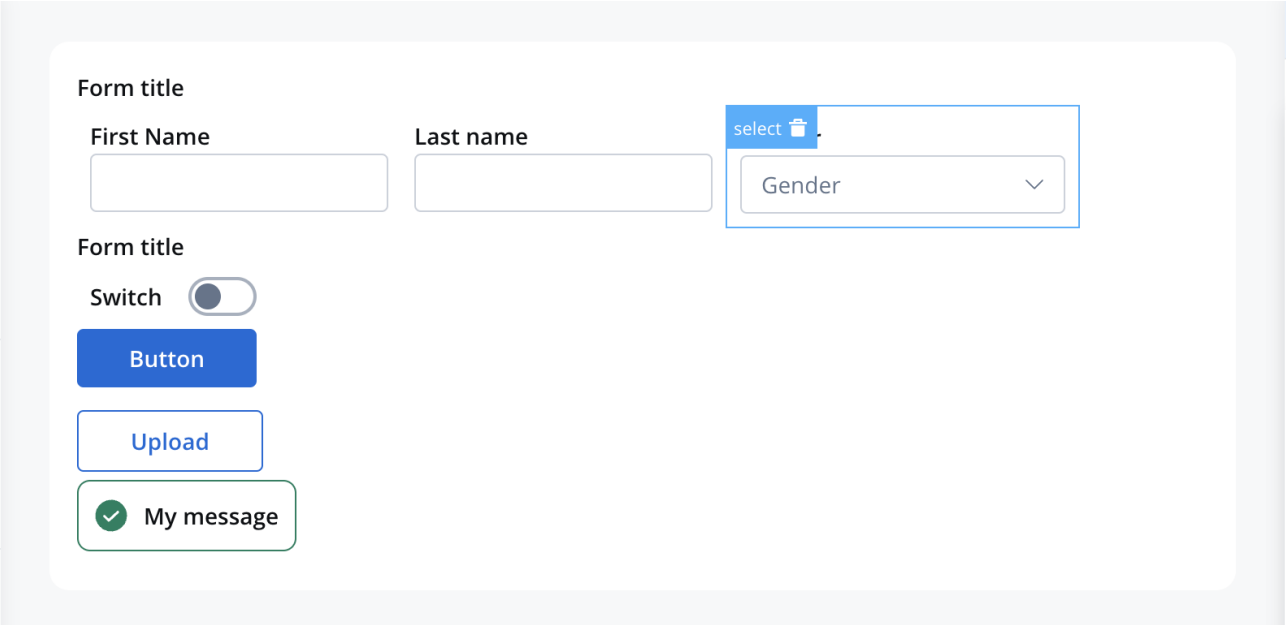
Text area styling

The ability to style the text area element using CSS properties is relevant because it allows you to customize the appearance of the text area to match the overall design of the website or application.

» UI Designer styling

Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Form elements / Select



The Select form field is an element that enables users to make a choice from a list of predefined options. It consists of multiple values, each of which is defined by a label that is displayed in the dropdown menu, and a code that is saved.

> **INFO**
>
> For instance, you could have a label of "Female" with the value "F" and "Male" with the value "M". This means that when a user selects "Female" in the process instance, the value "F" will be stored for the "Select" key.

Configuring the Select element

Select Settings

These allow you to customize the settings for the Select Field:

- **General**
- **Properties**
- **Datasource**
- **Validators**
- **Expressions**
- **UI actions**
- **Select styling**

General

- **Process data key** - creates the binding between form element and process data so it can be later used in decisions, business rules or integrations

Properties

- **Label** - the label of the select
- **Placeholder** - placeholder when the field has no value
- **Empty message** - text displayed for custom type when no results are found
- **Search for options** - displays a search to filter options
- **Helpertext** - additional information about the select field (can be hidden inside an infopoint)

Datasource

- **Default value** - autofill the select with this value. Going back to the example with Woman label with F value and Man with M to have a default value of Woman we need to configure here F
- **Source Type** - it can be Static, Enumeration, or Process Data
- **Add option** - label - value pairs can be defined here

Validators

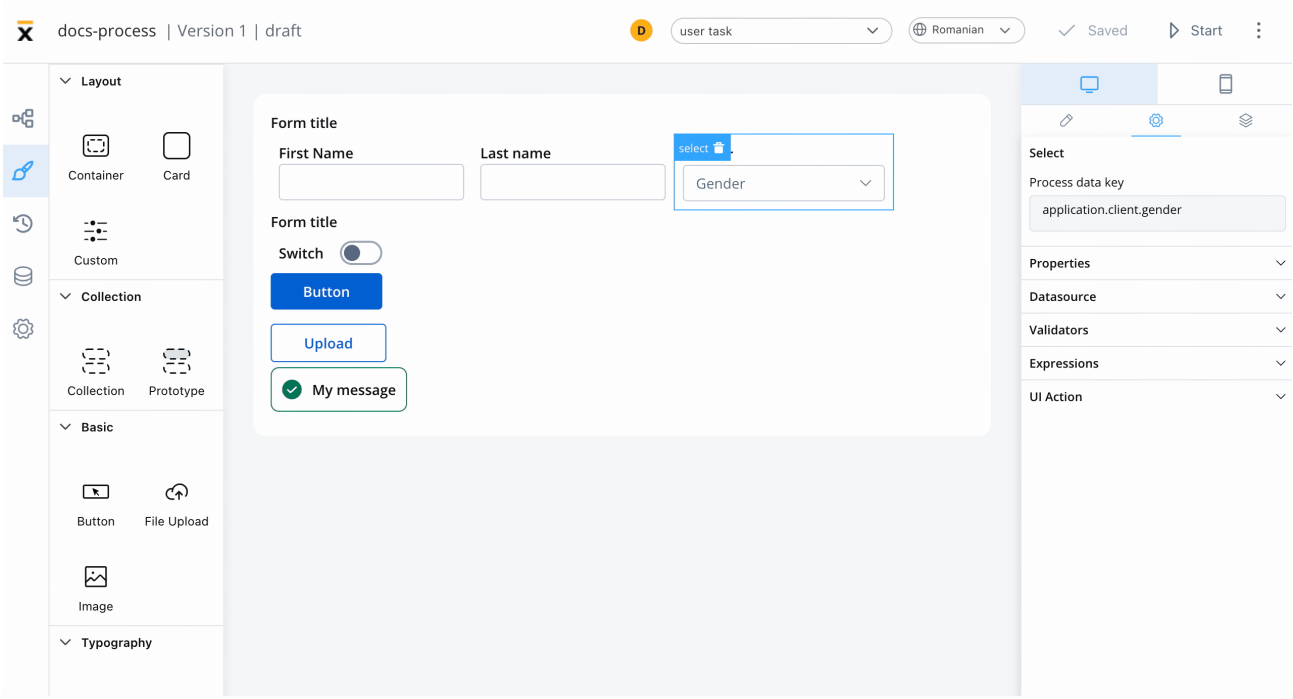
There are multiple validators can be added to a select (more details here).

Expressions

The select field's behavior can be defined using JavaScript expressions for hiding or disabling the element. The following properties can be configured for expressions:

- **Hide** - JavaScript expression used to hide the Select Field when it returns a truthy value
- **Disabled** - JavaScript expression used to disable the Select Field when it returns a truthy value

> **INFO**
>
> It's important to make sure that disabled fields have the same expression configured under the path expressions → hide.



UI actions

UI actions can be added to the select element to define its behavior and interactions.

- **Event** - possible value: `CHANGE`
- **Action Type** - select the action type

> **INFO**
>
> For more details on how to configure a UI action, click **here**.

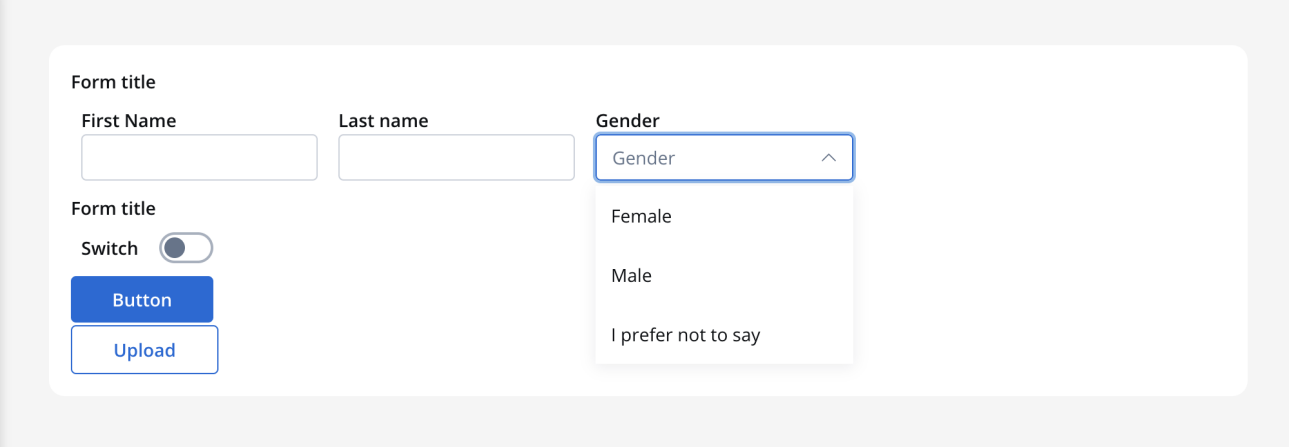
Select styling

Styling the Select field using CSS properties allows you to customize the appearance of the dropdown list and make it more visually appealing and

consistent with the overall design of the website or application.

» UI Designer styling

For example, a FORM element with a layout configuration including direction of Horizontal and some inputs, and a select element will look like this:



Icons

When customizing the appearance of a Select UI element that includes an icon, you can utilize the following properties:

- **Icon Key** - the key associated in the Media library, select the icon from the **Media Library**
- **Icon Color** - select the color of the icon using the color picker

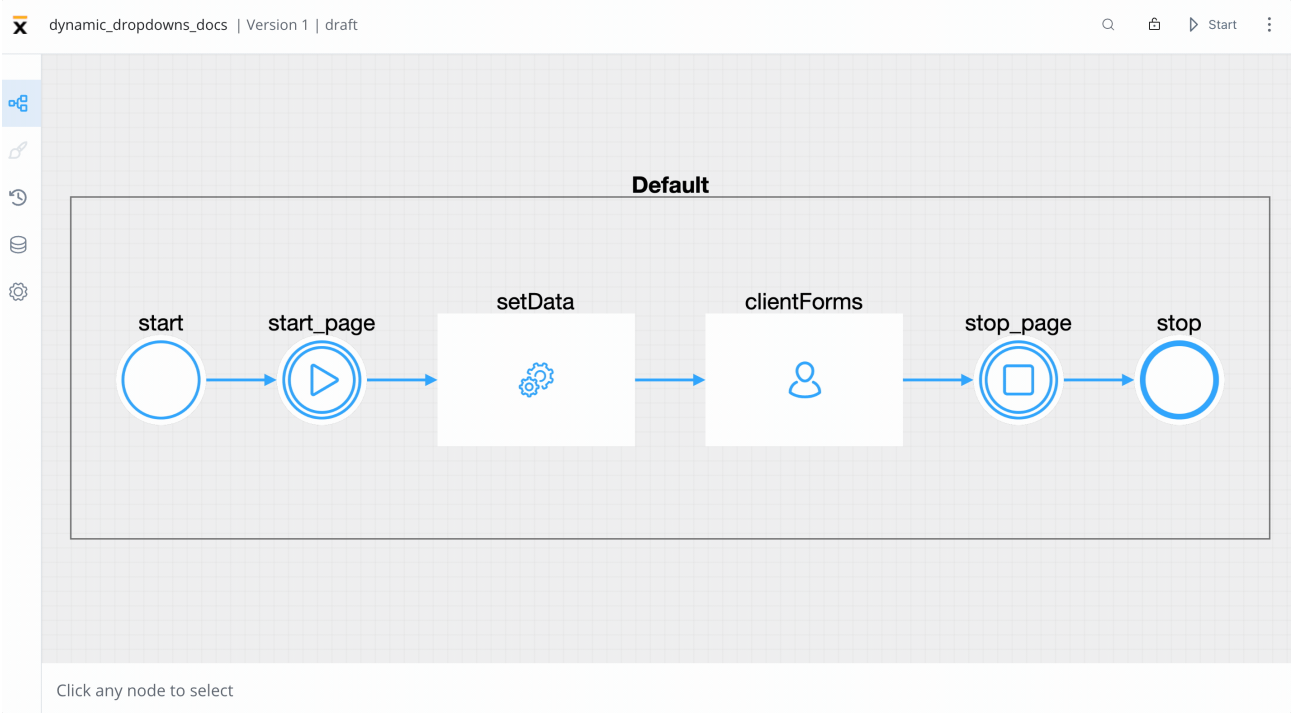
> **INFO**
>
> When setting the color, the entire icon is filled with that color, the SVG's fill. Avoid changing colors for multicolor icons.

Example - Dynamic dropdowns

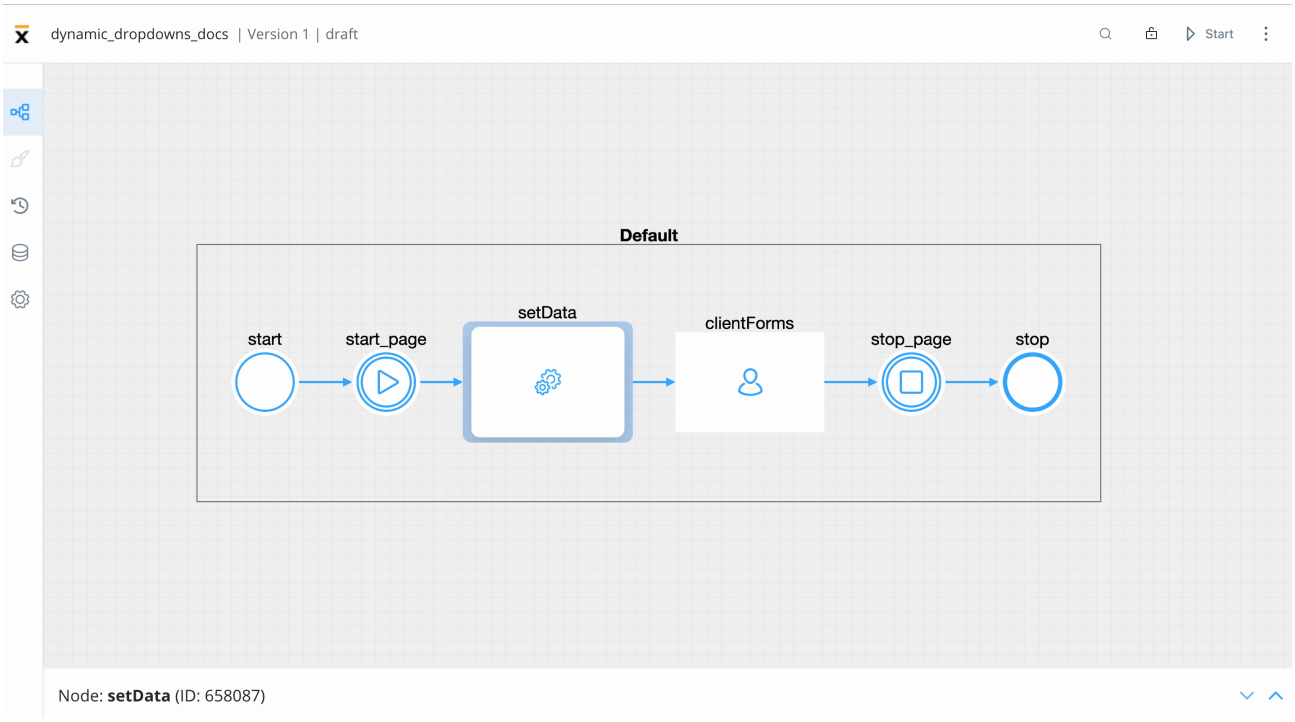
As mentioned previously, you can create dropdowns including static data, enumerations, or **process data**. Let's create an example using **process data** to create a process that contains **dynamic dropdowns**.

To create this kind of process, we need the following elements:

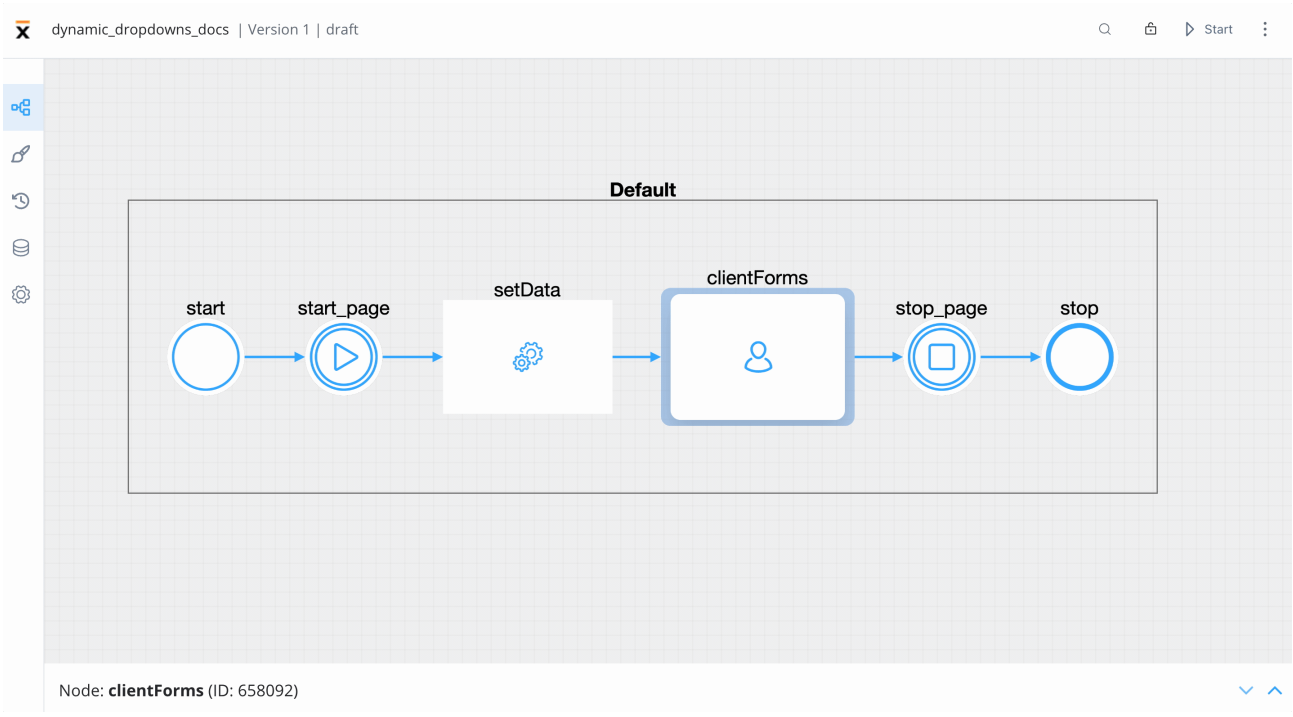
- a **start** node and an **end** node
- a **start milestone** UI element to it and an **end milestone** node



- a **task node** (this will be used to set which data will be displayed on the dropdowns)



- a **user task node** (here we have the client forms and here we add the SELECT elements)



Creating the process

Follow the next steps to create the process from scratch:

1. Open **FLOWX Designer** and from the **Processes** tab select **Definitions**.
2. Click on the breadcrumbs (top-right corner) then click **New process** (the Process Designer will now open).
3. Now add all the **necessary nodes** (as mentioned above).

Configuring the nodes

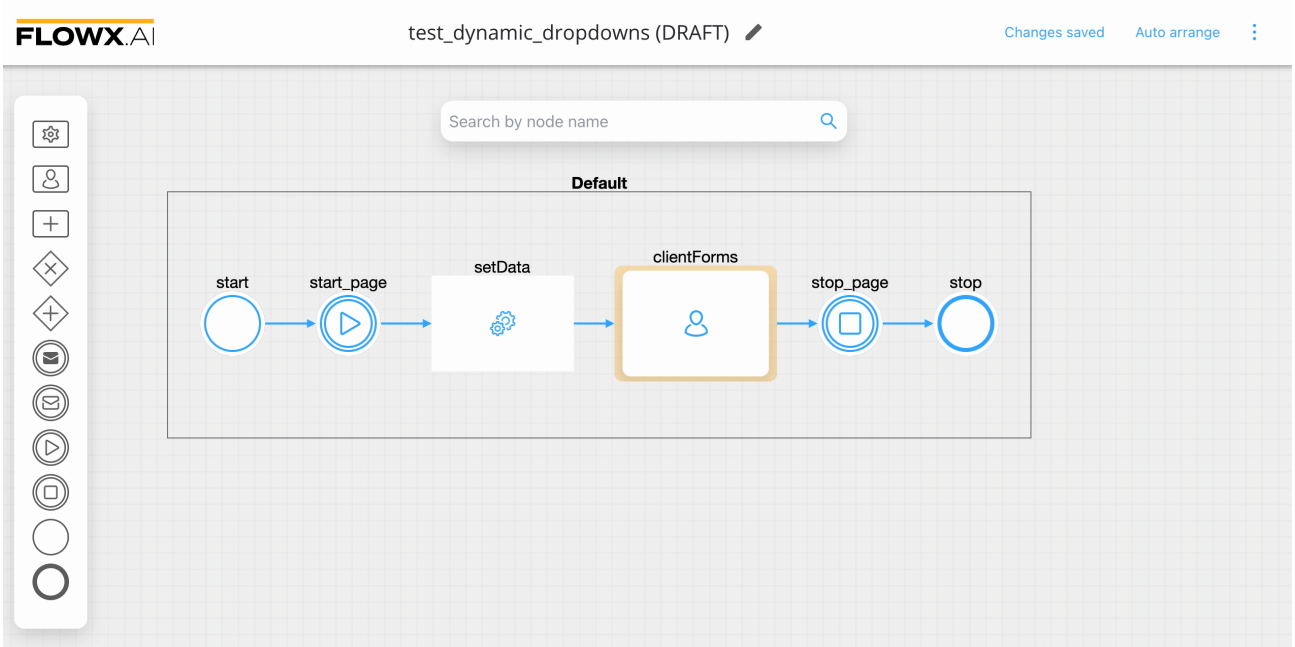
1. On the **start milestone** node, add a **page** UI element.

2. On the **task node**, add a new **Action** (this will set the data for the dropdowns) with the following properties:

    - Action type - **Business Rule**
    - **Automatic**
    - **Mandatory**
    - **Language** (we used an **MVEL** script to create a list of objects)

3. On the **user task node**, add a new **Action** (submit action, this will validate the forms and save the date) with the following properties:

- **Action type** - Save Data
- **Manual**
- **Mandatory**
- **Data to send** (the key where the data will be sent) - `application`



Below you can find the MVEL script used in the above example:

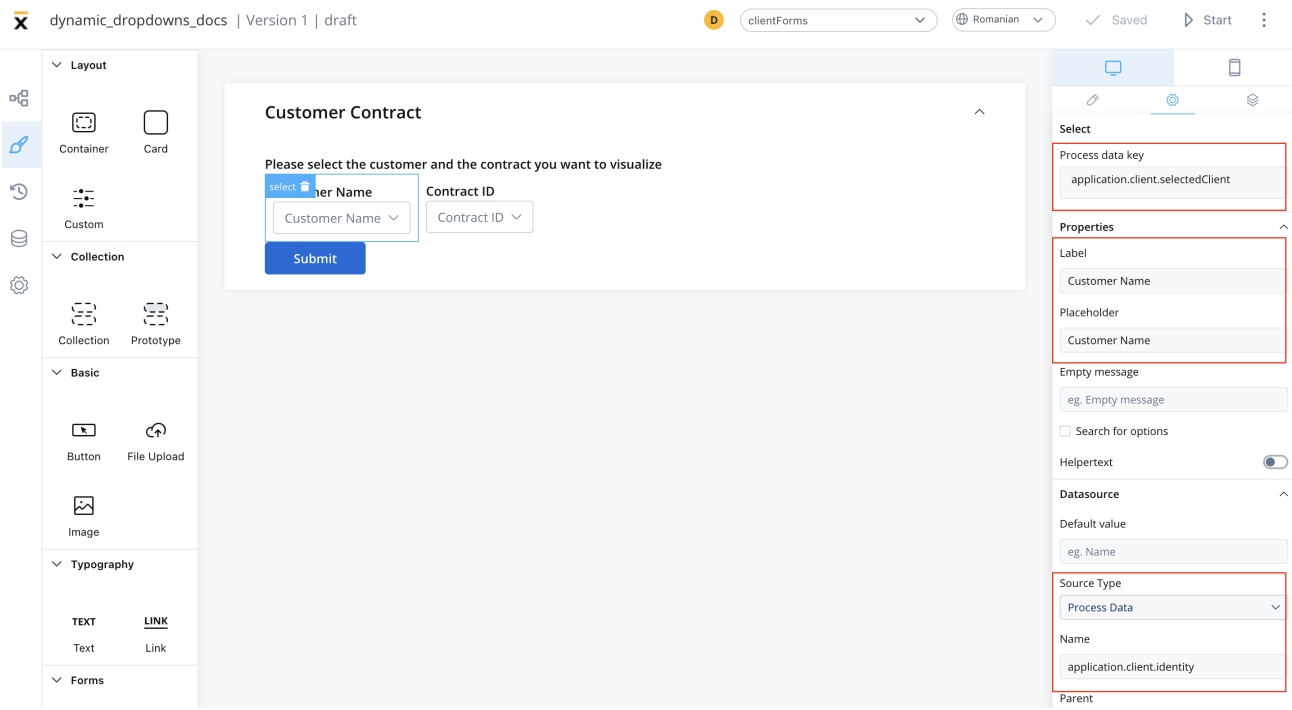
```
output.put("application",
{
    "client": {
        "identity": [
        {
            "value": "001",
            "label": "Eddard Stark"
        },
        {
            "value": "002",
            "label": "Sansa Stark"
        },
        {
            "value": "003",
            "label": "Catelyn Stark"
        }
    ]},
```

```
    "contracts": {
        "001": [
            {
                "value": "c001",
                "label": "Eddard Contract 1"
            },
            {
                "value": "c007",
                "label": "Eddard Contract 2"
            }
        ],
        "003": [
            {
                "value": "c002",
                "label": "Catelyn Contract 1",
            },
            {
                "value": "c003",
                "label": "Catelyn Contract 2 ",
            },
            {
                "value": "c004",
                "label": "Catelyn Contract 3"
            }
        ],
        "002": [
            {
                "value": "c005",
                "label": "Sansa Contract 1",
            }
        ]
    }
});
```

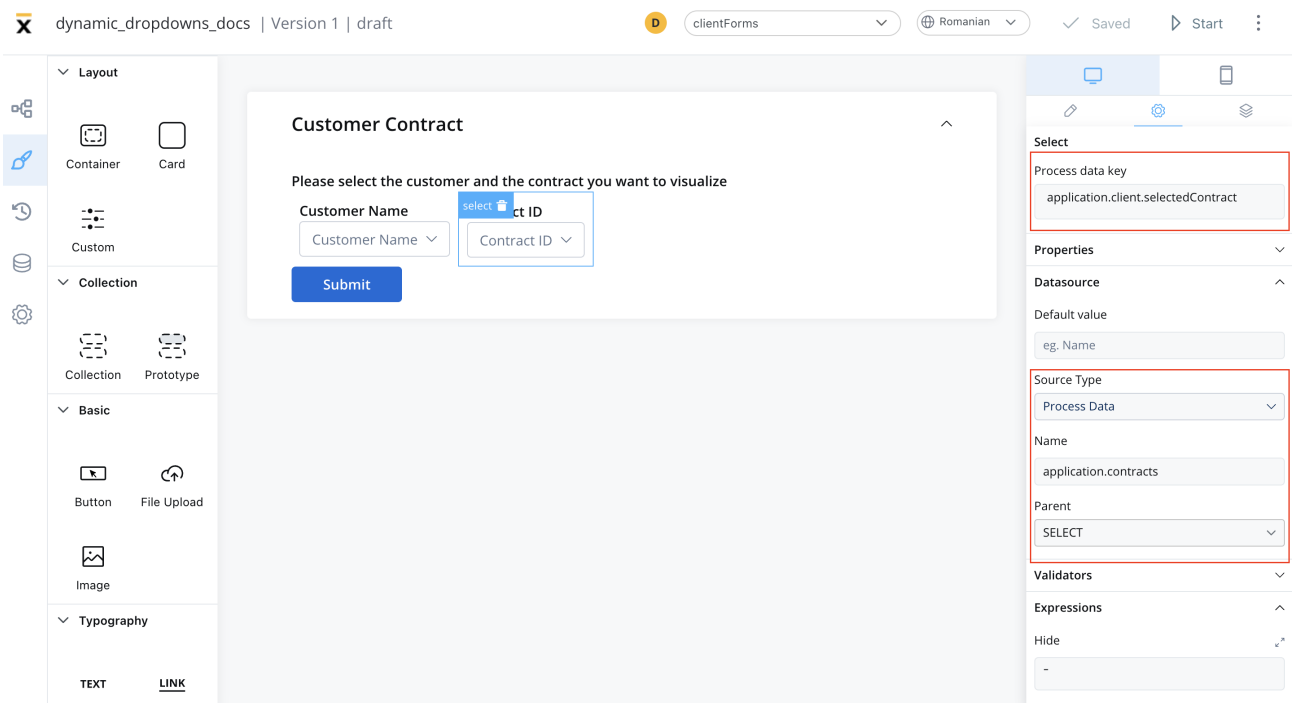
Configuring the UI

Follow the next steps to configure the UI needed:

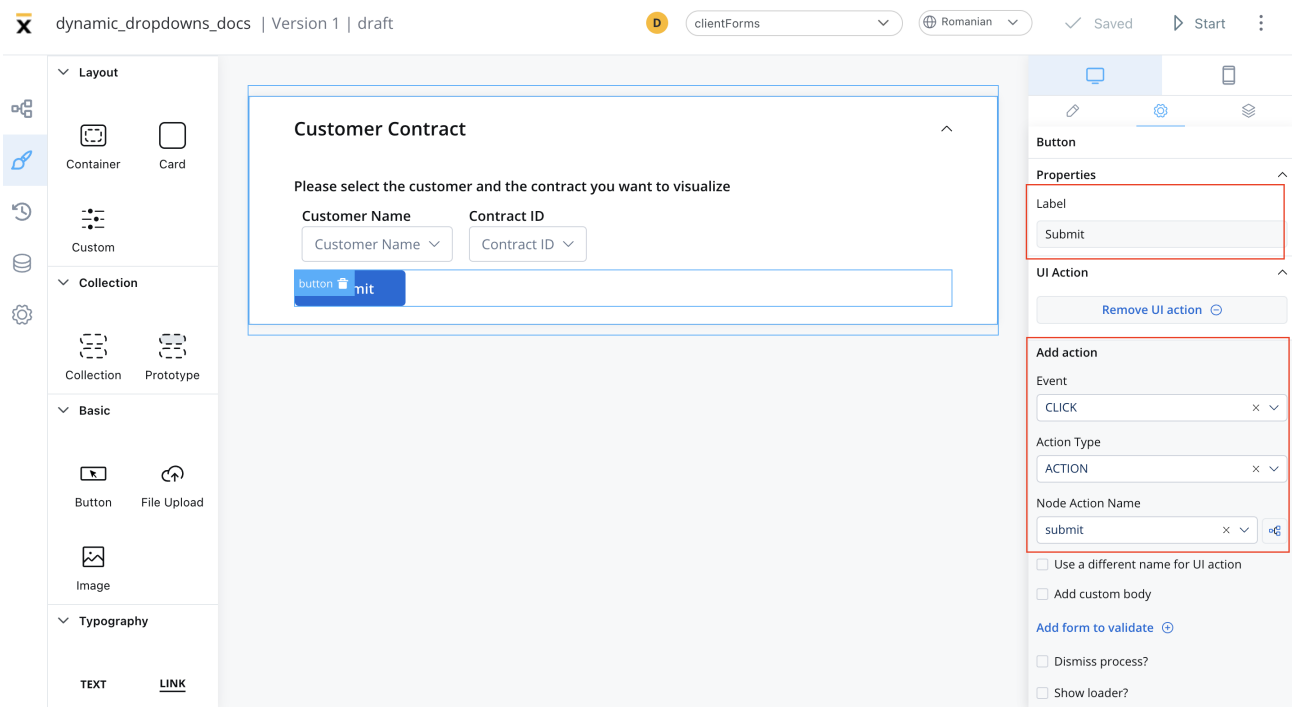
1. Select the **user task node** and click the **brush icon** to open **UI Designer**
2. Add a **card** element as a **root component** (this will group the other elements inside it) with the following properties:
   - **Message** - `{"application": ${application}}`
   - **Title** - *Customer Contract*
3. Inside the **card**, add a **form element**.
4. Inside the **form** add two **select elements**, first will represent, for example, the *Customer Name* and the second the *Contract ID.*
5. For first select element (Customer Name) set the following properties:
   - **Process data key** - `application.client.selectedClient`
   - **Label** - Customer Name
   - **Placeholder** - Customer Name
   - **Source type** - Process Data (to extract the data added in the **task node**)
   - **Name** - `application.client.identity`



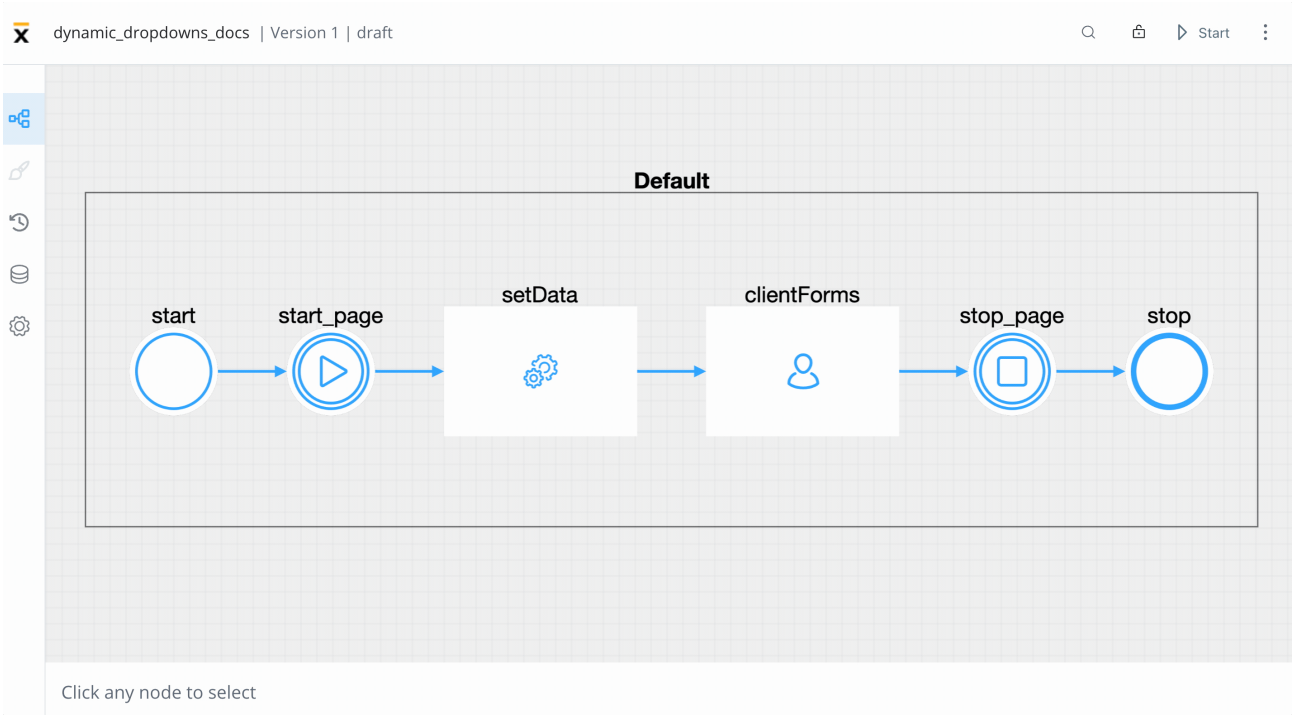
6. For the second select element (Contract ID) set the following properties:
    - **Process data key** - `application.client.selectedContract`
    - **Label** - Contract ID
    - **Placeholder** - Contract ID
    - **Source Type** - Process Data
    - **Name** - `application.contracts`
    - **Parent** - `SELECT` (choose from the dropdown list)



7. Add a button under the form that contains the select elements with the following properties:
    - **Label** - Submit
    - **Add UI action** - add the submit action attached earlier to the user task node

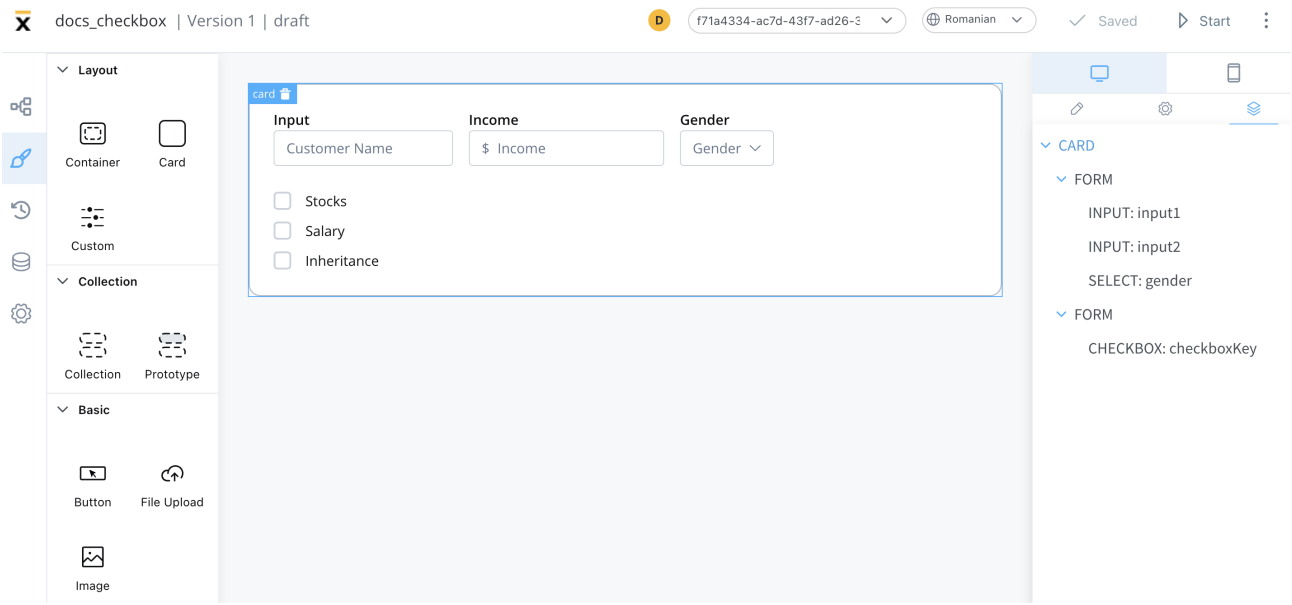


8. Test and run the process by clicking **Start process**.



Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Form elements / Checkbox



A checkbox form field is an interactive element in a web form that provides users with multiple selectable options. It allows users to choose one or more options from a pre-determined set by simply checking the corresponding checkboxes.

This type of form field can be used to gather information such as interests, preferences, or approvals, and it provides a simple and intuitive way for users to interact with a form.

Configuring the checkbox element

Checkbox settings

The available configuration options for this form element are:

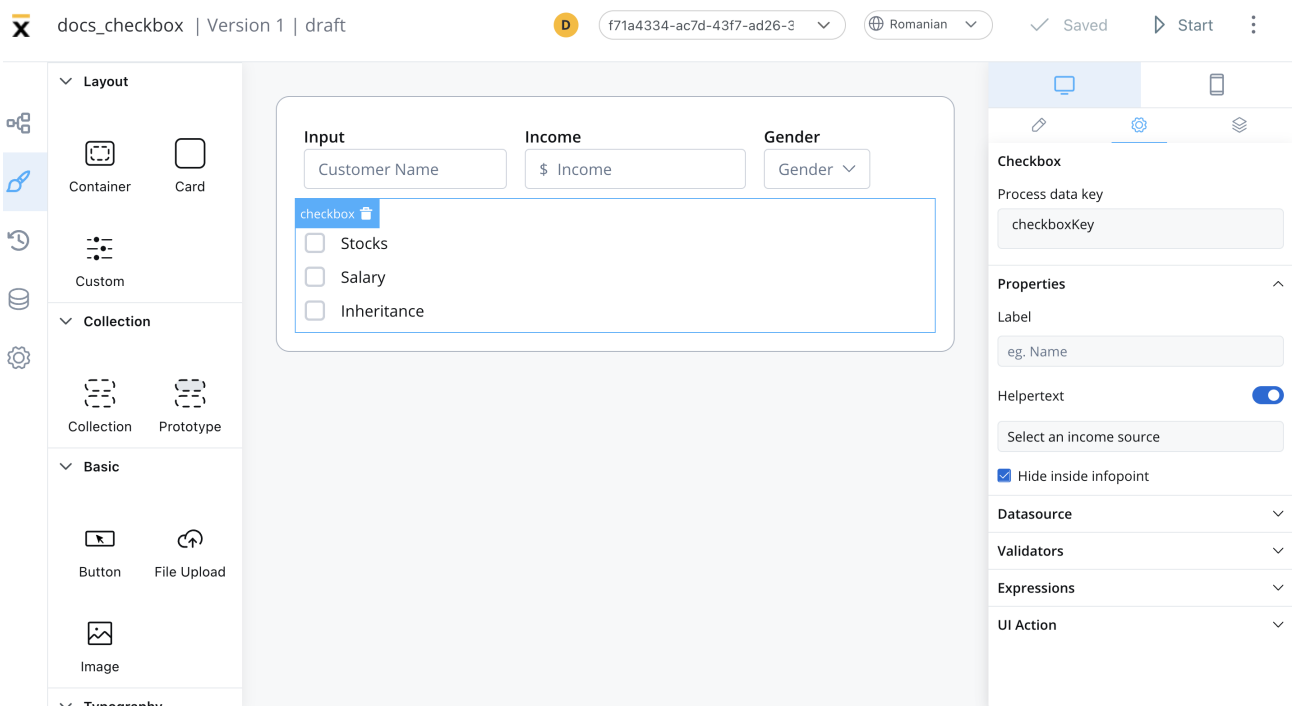
- **General**
- **Properties**
- **Datasource**
- **Validators**
- **Expressions**
- **UI actions**
- **Checkbox styling**

General

- **Process data key** - creates the binding between form element and process data, so it can be later used in decisions, business rules or integrations

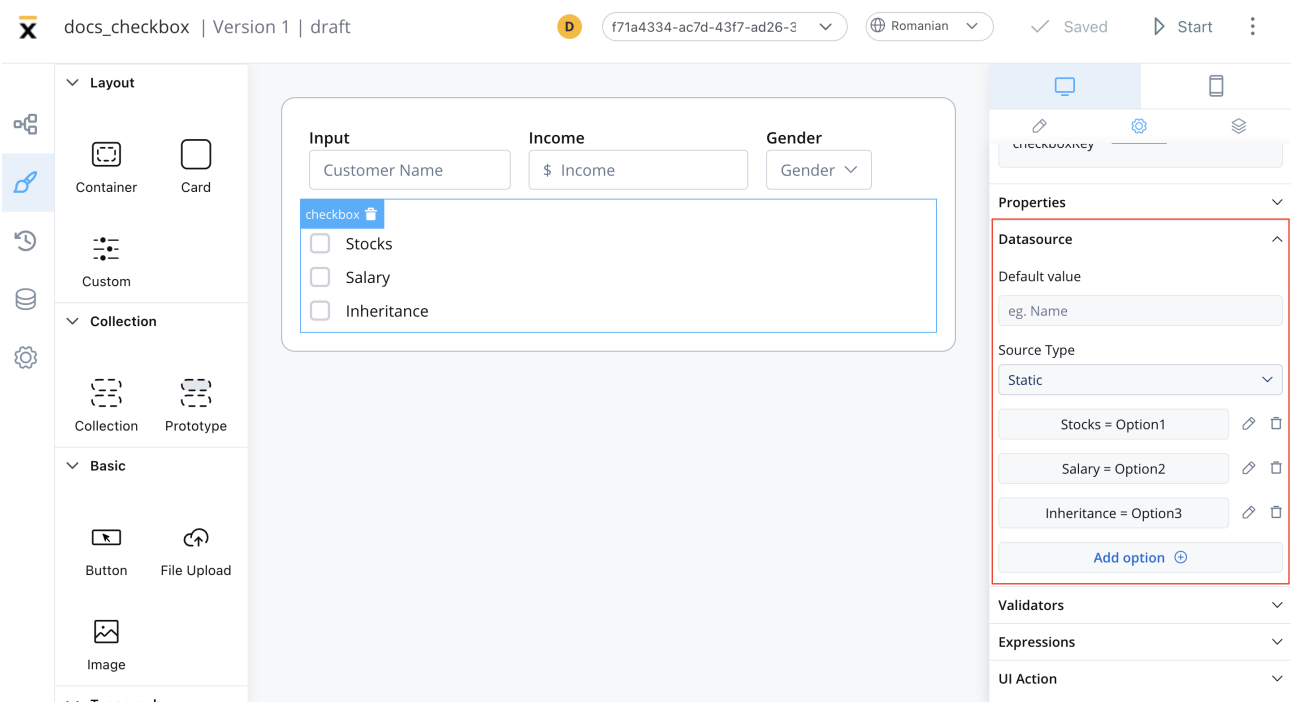
Properties

- **Label** - the label that appears on the checkbox
- **Helpertext** - additional information about the checkbox (can be hidden inside an infopoint)



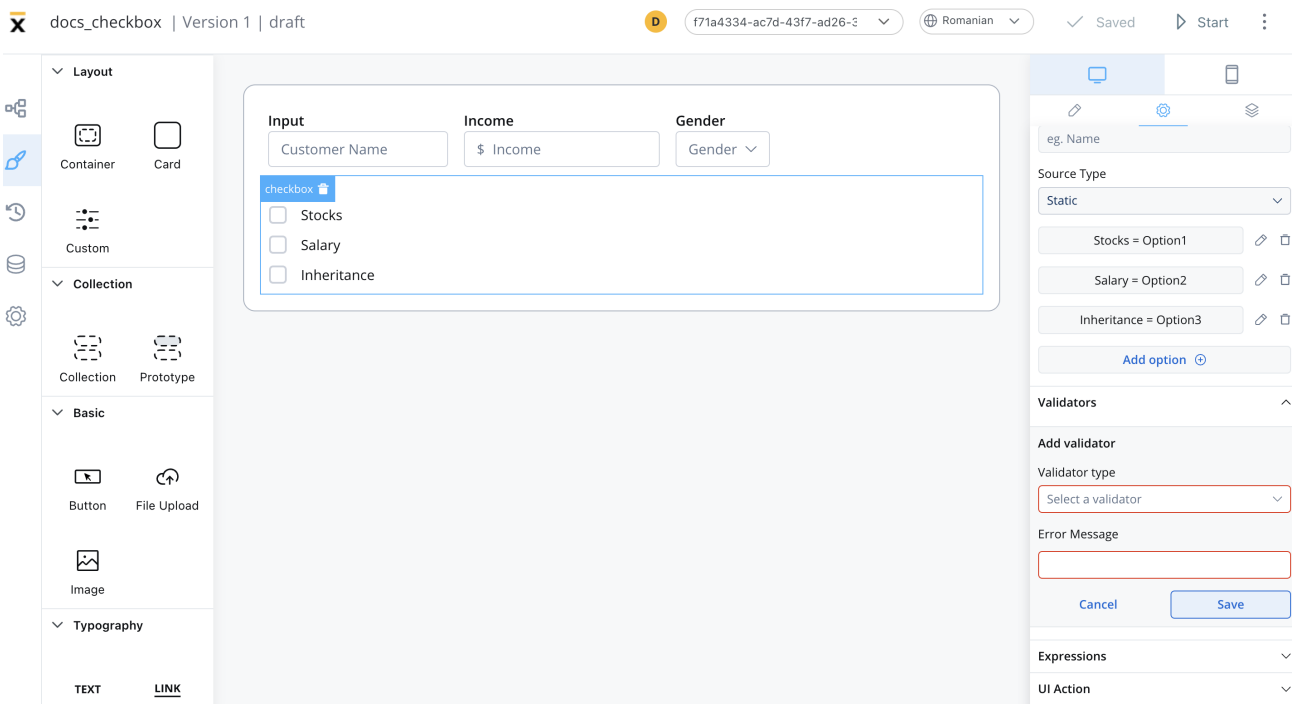
Datasource

- **Default Value** - the default value of the checkbox
- **Source Type** - it can be Static, Enumeration, or Process Data
- **Add option** - label - value pairs can be defined here



Validators

The following validators can be added to a checkbox: `required` and `custom` (more details here).



Expressions

The checkbox behavior can be defined using JavaScript expressions for hiding or disabling the element. The following properties can be configured for expressions:

- **Hide** - JavaScript expression used to hide the checkbox when it returns a truthy value
- **Disabled** - JavaScript expression used to disable the checkbox when it returns a truthy value

> **INFO**
>
> It's important to make sure that disabled fields have the same expression configured under the path expressions → hide.

UI actions

UI actions can be added to the checkbox element to define its behavior and interactions.

- **Event** - possible value: `CHANGE`
- **Action Type** - select the action type

> **INFO**
>
> For more details on how to configure a UI action, click **here**.

Checkbox styling

The type of the checkbox can be selected by using the **styling** tab in **UI Designer**, possible values:

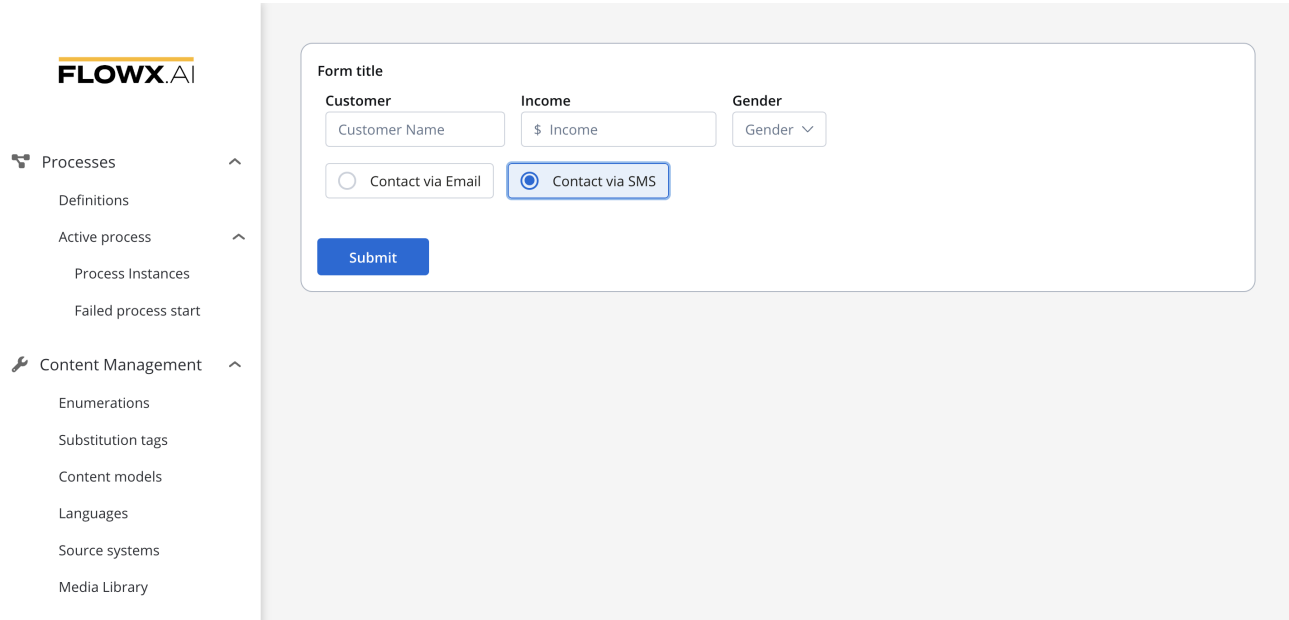
- clear
- bordered

> **INFO**
>
> For more valid CSS properties, click here.

A clear checkbox element with three options added, and a column layout will look like as it follows:

Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Form elements / Radio



Radio buttons are normally presented in radio groups (a collection of radio buttons describing a set of related options). Only one radio button in a group can be selected at the same time.

Configuring the radio field element

Radio settings

The available configuration options for this form element are:

- **General**
- **Properties**
- **Datasource**
- **Validators**
- **Expressions**
- **UI actions**
- **Radio styling**

General

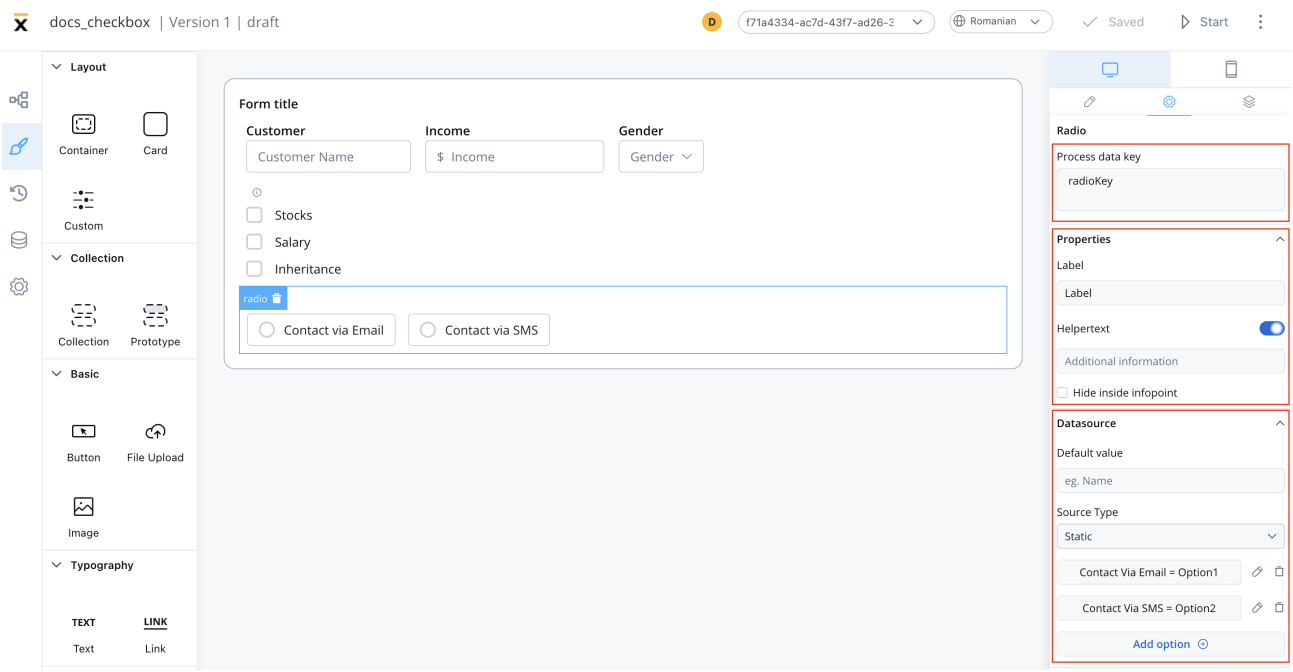
- **Process data key** - creates the binding between form element and process data so it can be later used in decisions, business rules or integrations

Properties

- **Label** - the label that appears on the radio
- **Helpertext** - additional information about the radio (can be hidden inside an infopoint)

Datasource

- **Default Value** - the default values of the radio element
- **Source Type** - it can be Static, Enumeration, or Process Data
- **Add option** - label - value pairs can be defined here



Validators

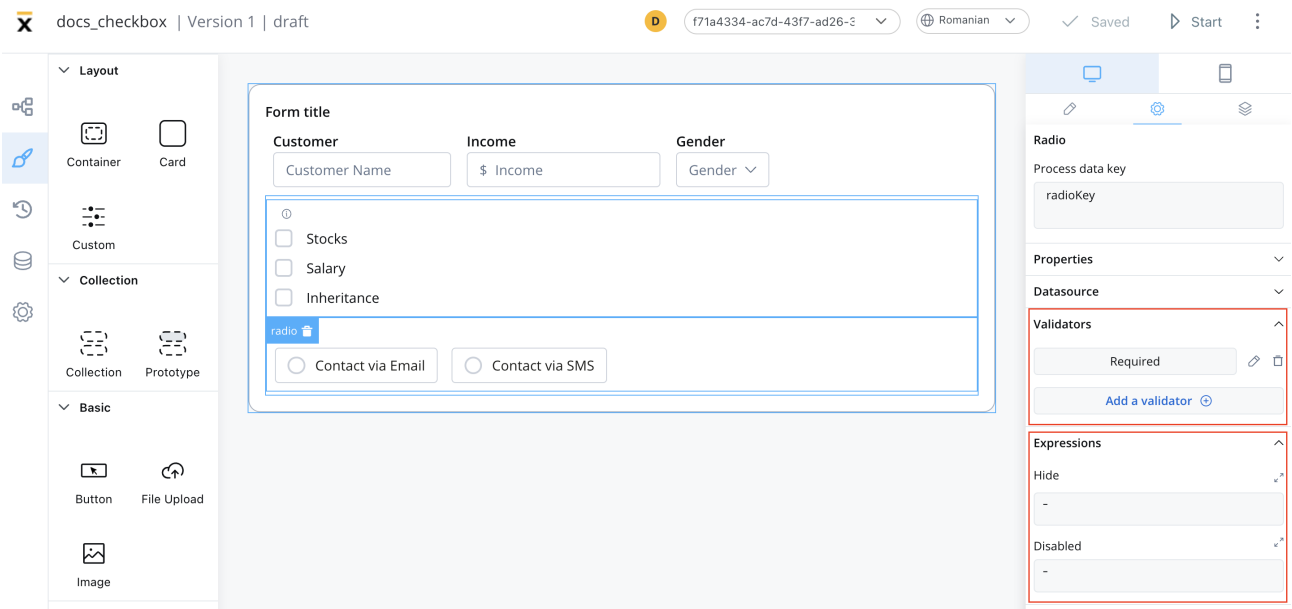
The following validators can be added to a radio: `required` and `custom` (more details here)

Expressions

The radio's element behavior can be defined using JavaScript expressions for hiding or disabling the element. The following properties can be configured for expressions:

- **Hide** - JavaScript expression used to hide the Radio element when it returns a truthy value
- **Disabled** - JavaScript expression used to disable the Radio element when it returns a truthy value

> **INFO**
>
> It's important to make sure that disabled fields have the same expression configured under the path expressions → hide.



UI actions

UI actions can be added to the radio element to define its behavior and interactions.

- **Event** - possible value: `CHANGE`
- **Action Type** - select the action type

> **INFO**
>
> For more details on how to configure a UI action, click **here**.

Radio styling

The type of the radio can be selected by using the **styling** tab in **UI Designer**, possible values:

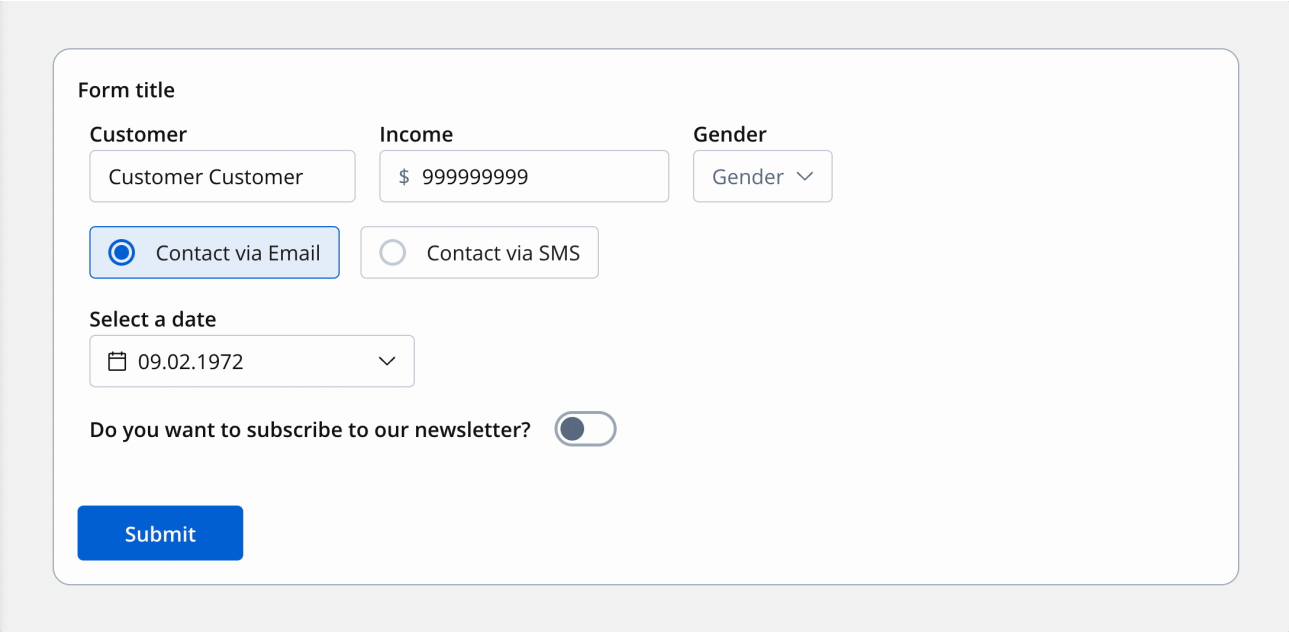
- clear
- bordered

> **INFO**
>
> For more valid CSS properties, click here.

A Radio element with two options added, and with a layout configuration set to horizontal will look like as it follows:

Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Form elements / Switch



A switch, a toggle switch, is another form element that can be utilized to create an intuitive user interface. The switch allows users to select a response by toggling it between two states. Based on the selection made by the user, the corresponding Boolean value of either true or false will be recorded and stored in the process instance values for future reference.

Configuring the switch element

Switch settings

The available configuration options for this form element are:

- General
- Properties
- Datasource
- Validators
- Expressions
- UI actions
- Switch styling

General

- **Process data key** - creates the binding between form element and process data so it can be later used in decisions, business rules or integrations

Properties

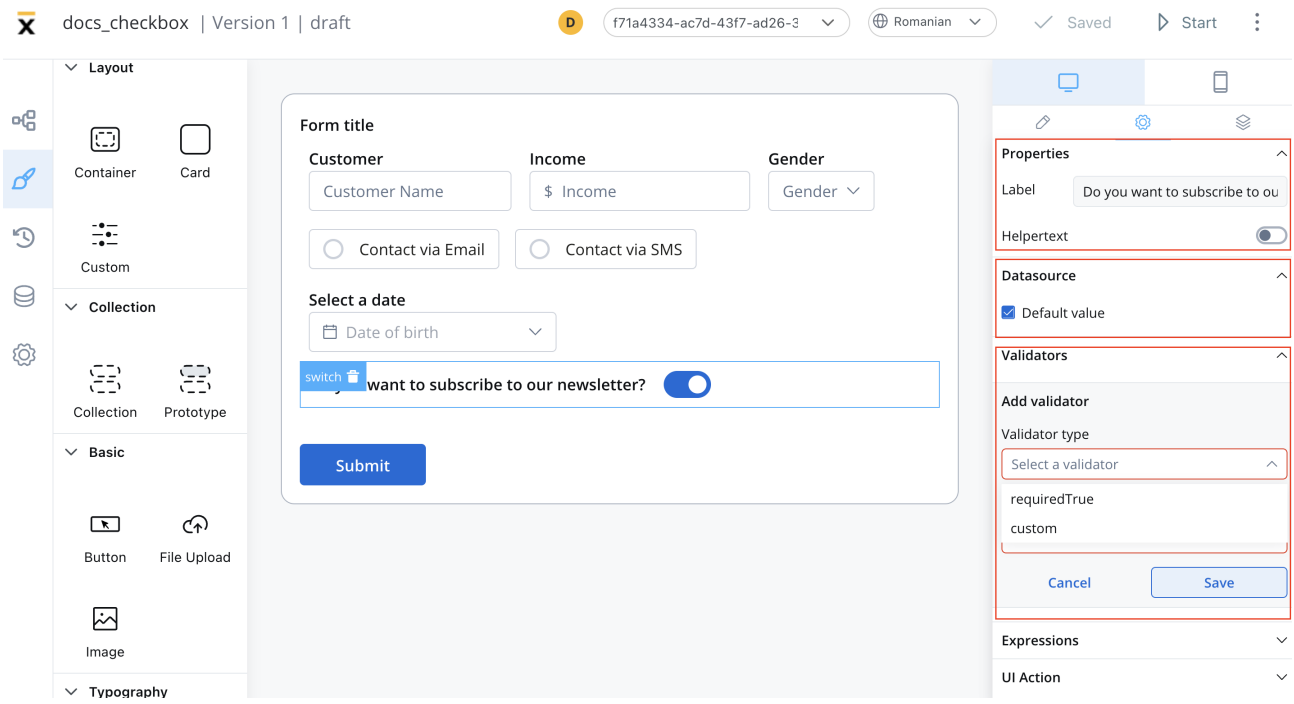
- **Label** - the label of the switch
- **Helpertext** - additional information about the switch element (can be hidden inside an infopoint)

Datasource

- **Default Value** - the default value of the switch form field (it can be switched on or switched off)

Validators

The following validators can be added to a switch element: `requiredTrue` and `custom` (more details here).



Expressions

- **Hide** - JavaScript expression used to hide the Switch element when it returns a truthy value
- **Disabled** - JavaScript expression used to disable the Switch element when it returns a truthy value

UI actions

UI actions can be added to the switch element to define its behavior and interactions.

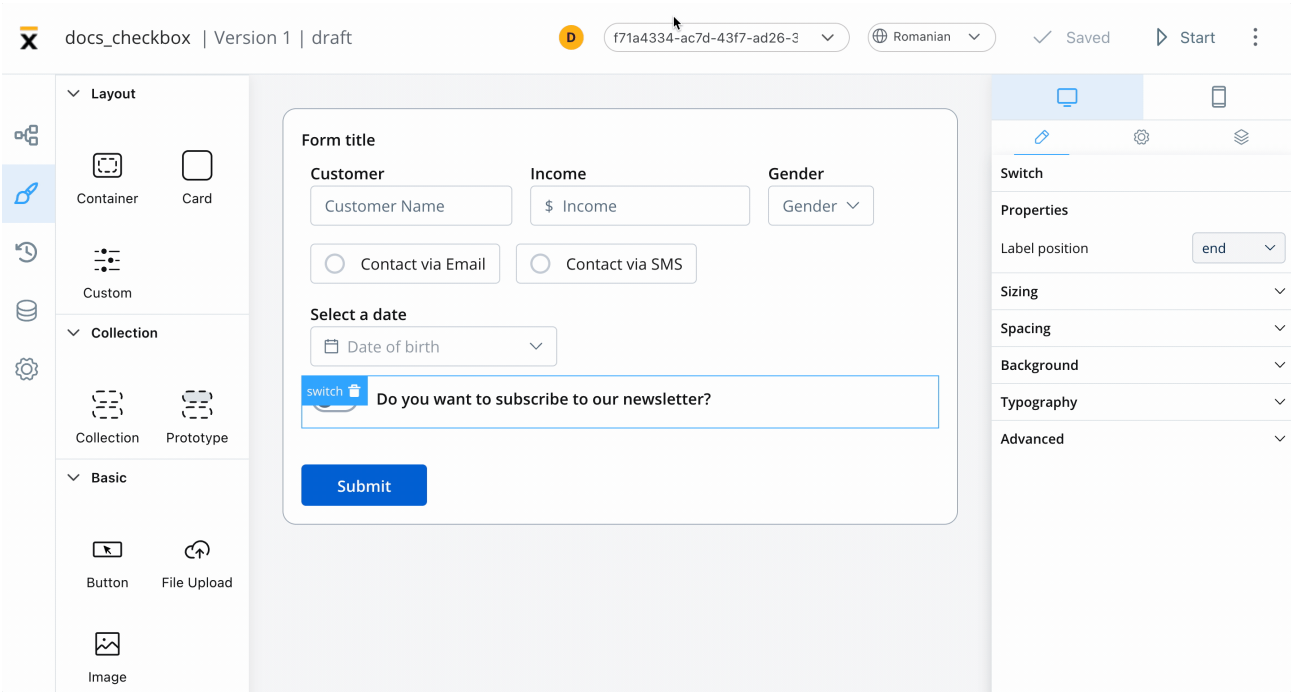
- Event - possible value: CHANGE
- Action Type - select the action type

> **INFO**

> For more details on how to configure a UI action, click here.

Switch styling

The label of the switch can be positioned either as `start` or `end`.

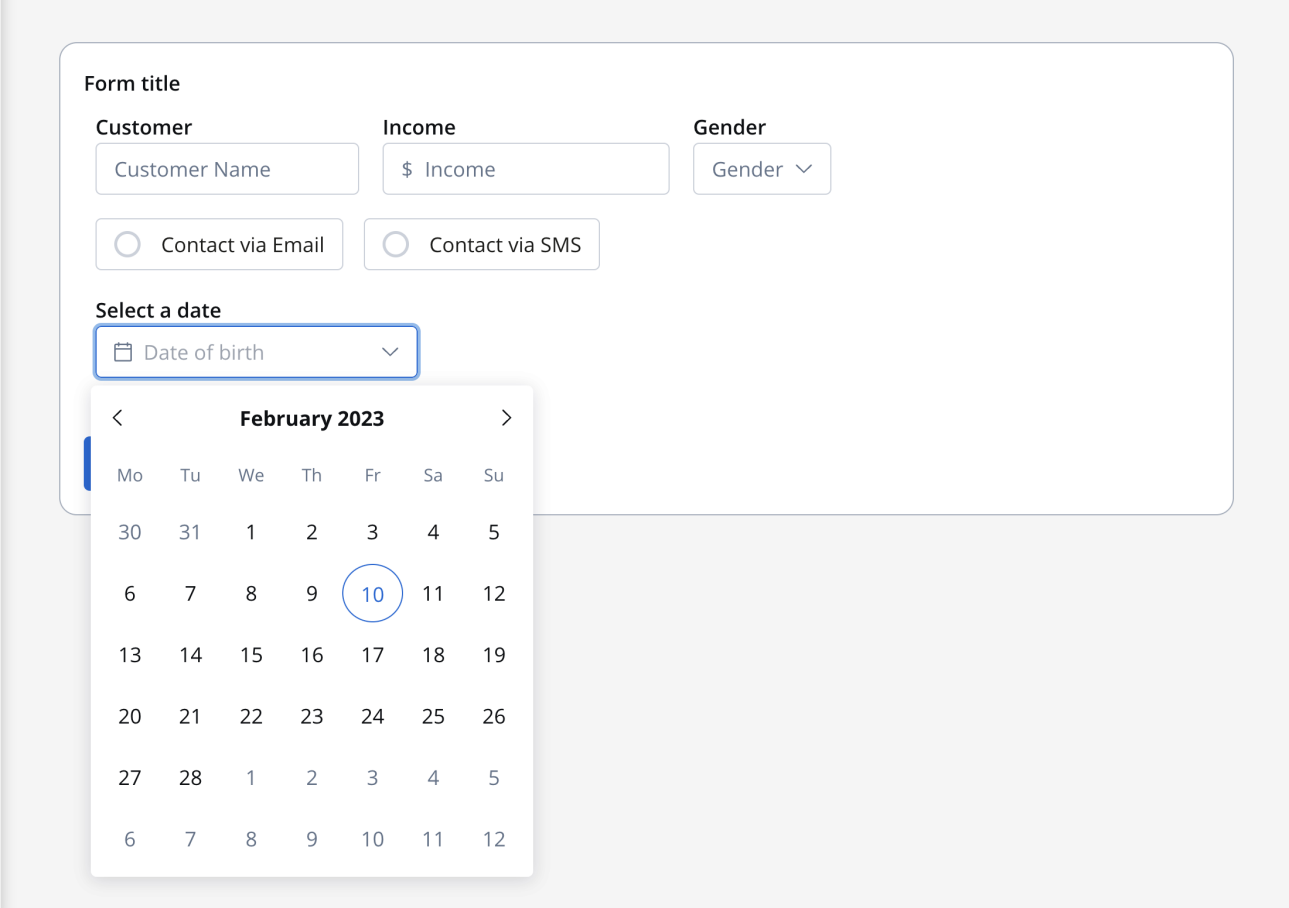


> **INFO**
>
> For more valid CSS properties, click here.

Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Form elements /

Datepicker



The datepicker (Calendar Picker) is a lightweight component that allows end users to enter or select a date value.

> **INFO**
>
> The default datepicker value is `DD.MM.YYYY`.

Configuring the datepicker element

Datepicker settings

The available configuration options for this form element are:

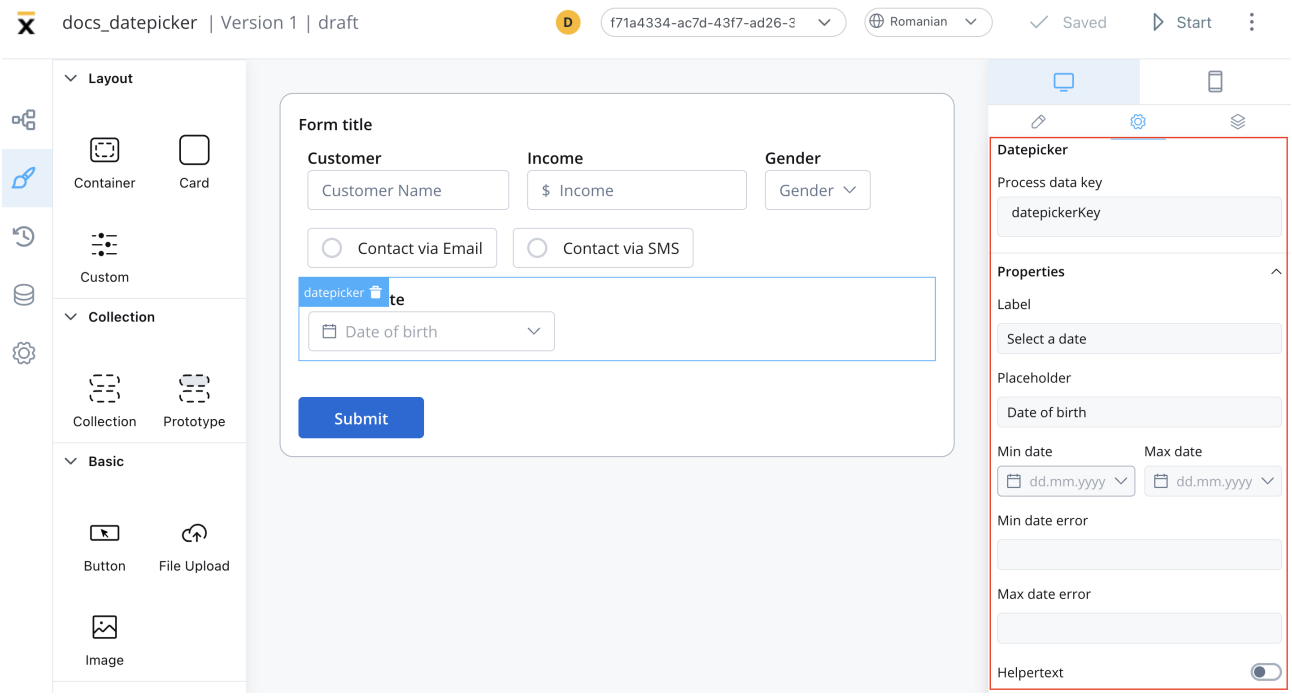
- General
- Properties
- Datasource
- Validators
- Expressions
- UI actions
- Datepicker styling

General

- **Process data key** - creates the binding between form element and process data so it can be later used in decisions, business rules or integrations

Properties

- **Label** - the label of the datepicker
- **Placeholder** - placeholder when the field has no value
- **Min Date** - set the minimum valid date selectable in the datepicker
- **Max Date** - set the maximum valid date selectable in the datepicker
- **Min Date, Max Date error** - when a date is introduced by typing, define the error message to be displayed
- **Helpertext** - additional information about the input field (can be hidden inside an infopoint)

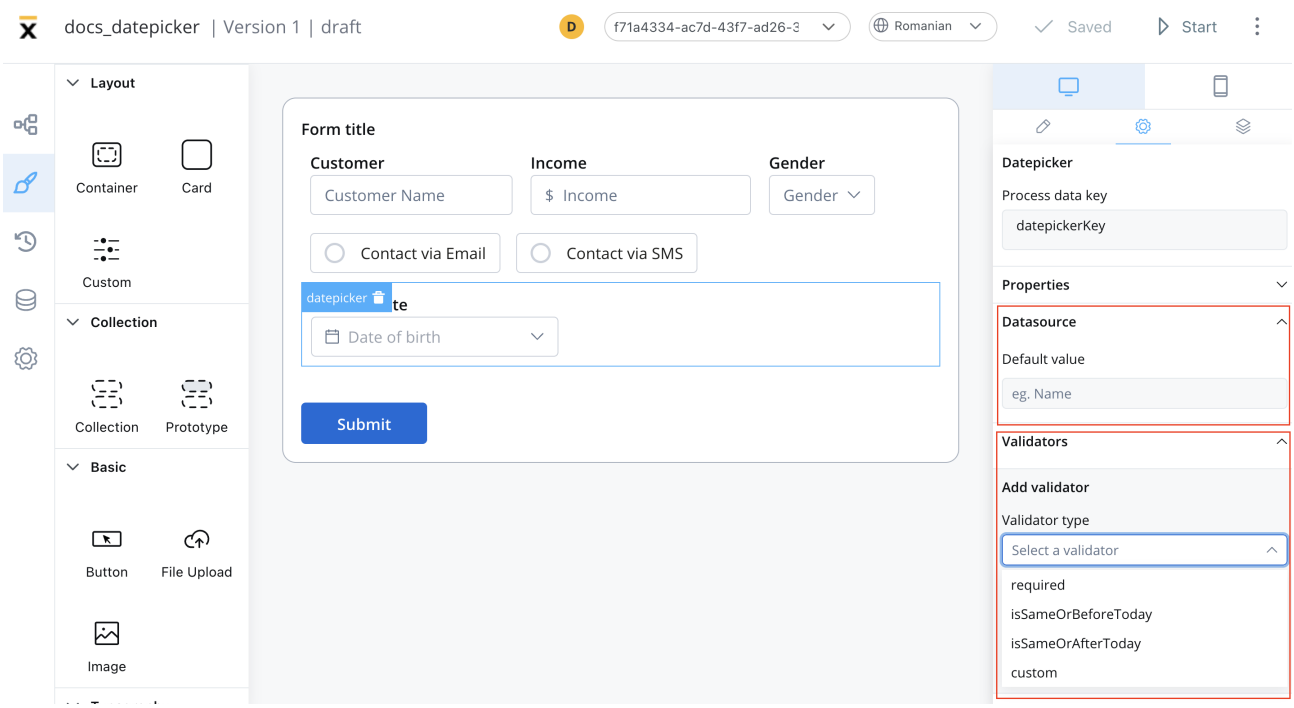


Datasource

- **Default Value** - the default values of the datepicker element, this will autofill the datepicker when you will run the process

Validators

The following validators can be added to a datepicker: `required`, `custom`, `isSameOrBeforeToday` or `isSameOrAfterToday` (more details here).



Expressions

The datepicker behavior can be defined using JavaScript expressions for hiding or disabling the element. The following properties can be configured for expressions:

- **Hide** - JavaScript expression used to hide the datepicker when it returns a truthy value
- **Disabled** - JavaScript expression used to disable the datepicker when it returns a truthy value

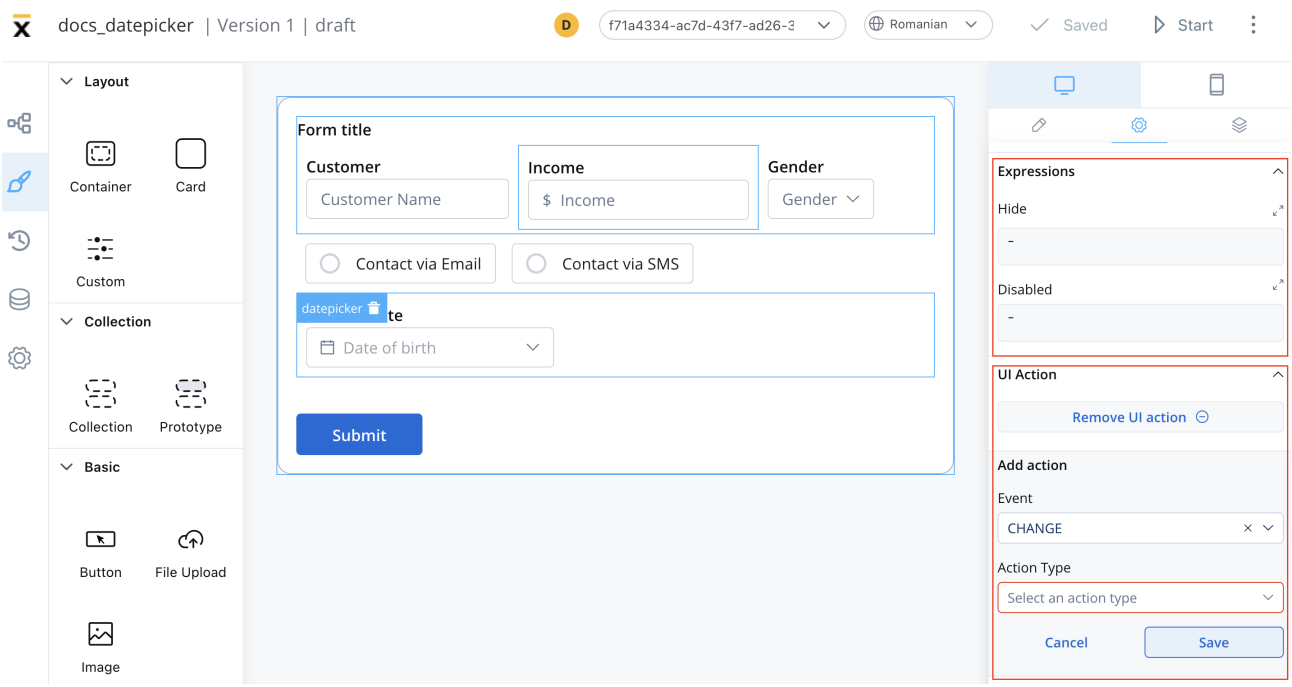
> **INFO**
>
> It's important to make sure that disabled fields have the same expression configured under the path expressions → hide.

UI actions

UI actions can be added to the datepicker element to define its behavior and interactions.

- **Event** - possible value: `CHANGE`
- **Action Type** - select the action type

> **INFO**
>
> For more details on how to configure a UI action, click **here**.



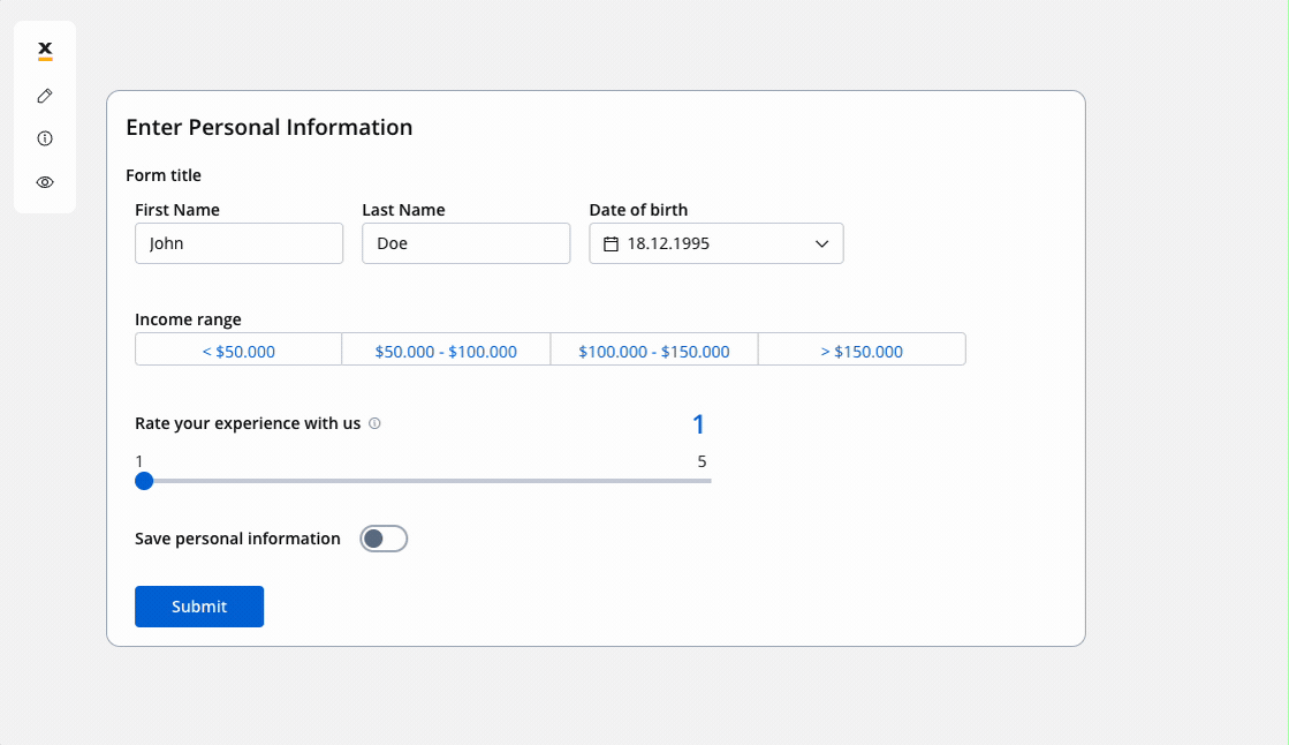
Datepicker styling

The styling of a datepicker element can be customized in various ways using CSS properties like typography color, border-radius/width, or advanced CSS params. This allows you to create a datepicker that fits seamlessly with the overall design of the application you are developing.

> **INFO**
>
> For more valid CSS properties, click here.

Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Form elements / Slider



It allows users to pick only one option from a group of options, and you can choose to have between 2 and 5 options in the group. The segmented button is

easy to use, and can help make your application easier for people to use.

Configuring the slider element

Slider settings

The available configuration options for this form element are:

- General
- Properties
- Datasource
- Validators
- Expressions
- UI actions
- Slider styling

General

- **Process data key** - creates the binding between form element and process data so it can be later used in decisions, business rules or integrations

Properties

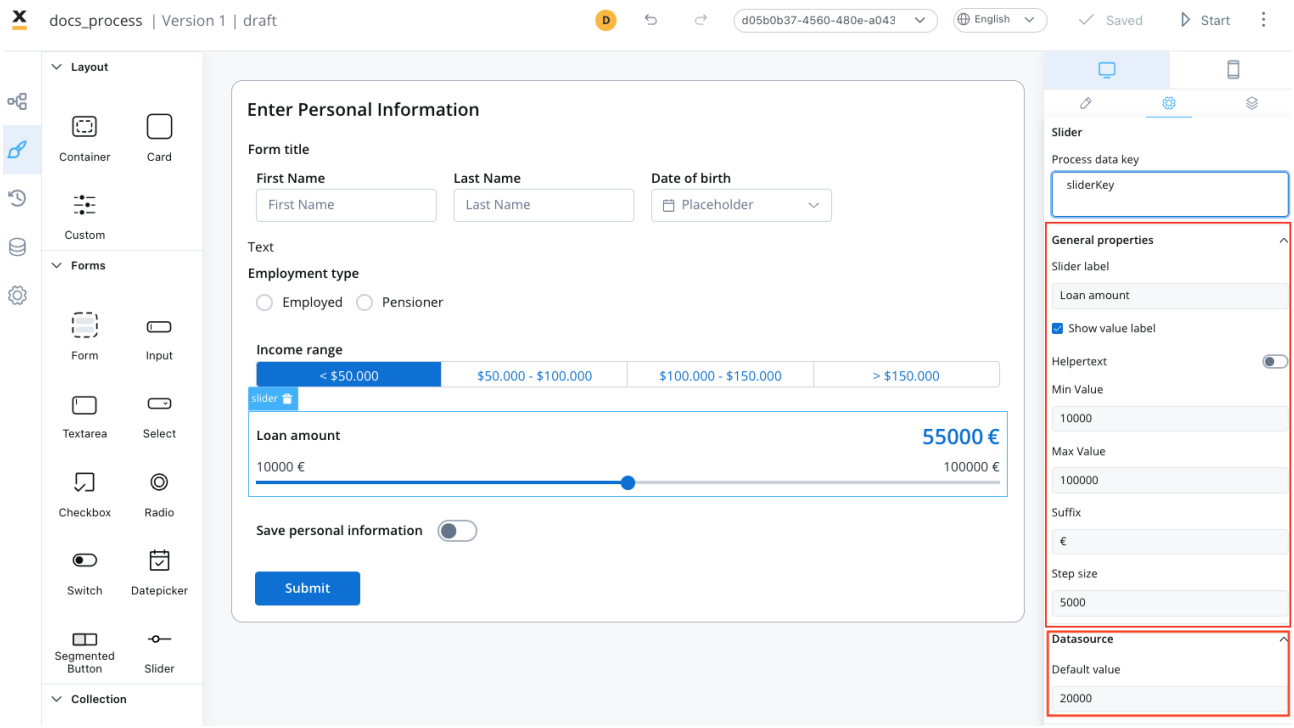
- **Label** - the label of the slider
- **Show value label** - a toggle option that determines whether the current selected value of the slider is displayed as a label alongside the slider handle
- **Helpertext** - an optional field that provides additional information or guidance related to the usage or function of the slider, it can be hidden within an

infopoint, which users can expand or access for further detail

- **Min Value** - the minimum value or starting point of the slider's range, it defines the lowest selectable value on the slider
- **Max Value** - the maximum value or end point of the slider's range, it sets the highest selectable value on the slider
- **Suffix** - an optional text or symbol that can be added after the displayed value on the slider handle or value label, it is commonly used to provide context or units of measurement
- **Step size** - the increment or granularity by which the slider value changes when moved, it defines the specific intervals or steps at which the slider can be adjusted, allowing users to make more precise or discrete value selections

Datasource

- **Default Value** - the default value of the slider (static value - integer) the initial value set on the slider when it is first displayed or loaded, it represents a static value (integer), that serves as the starting point or pre-selected value for the slider, users can choose to keep the default value or adjust it as desired



Validators

The following validators can be added to a slider: `required` and `custom` (more details here).

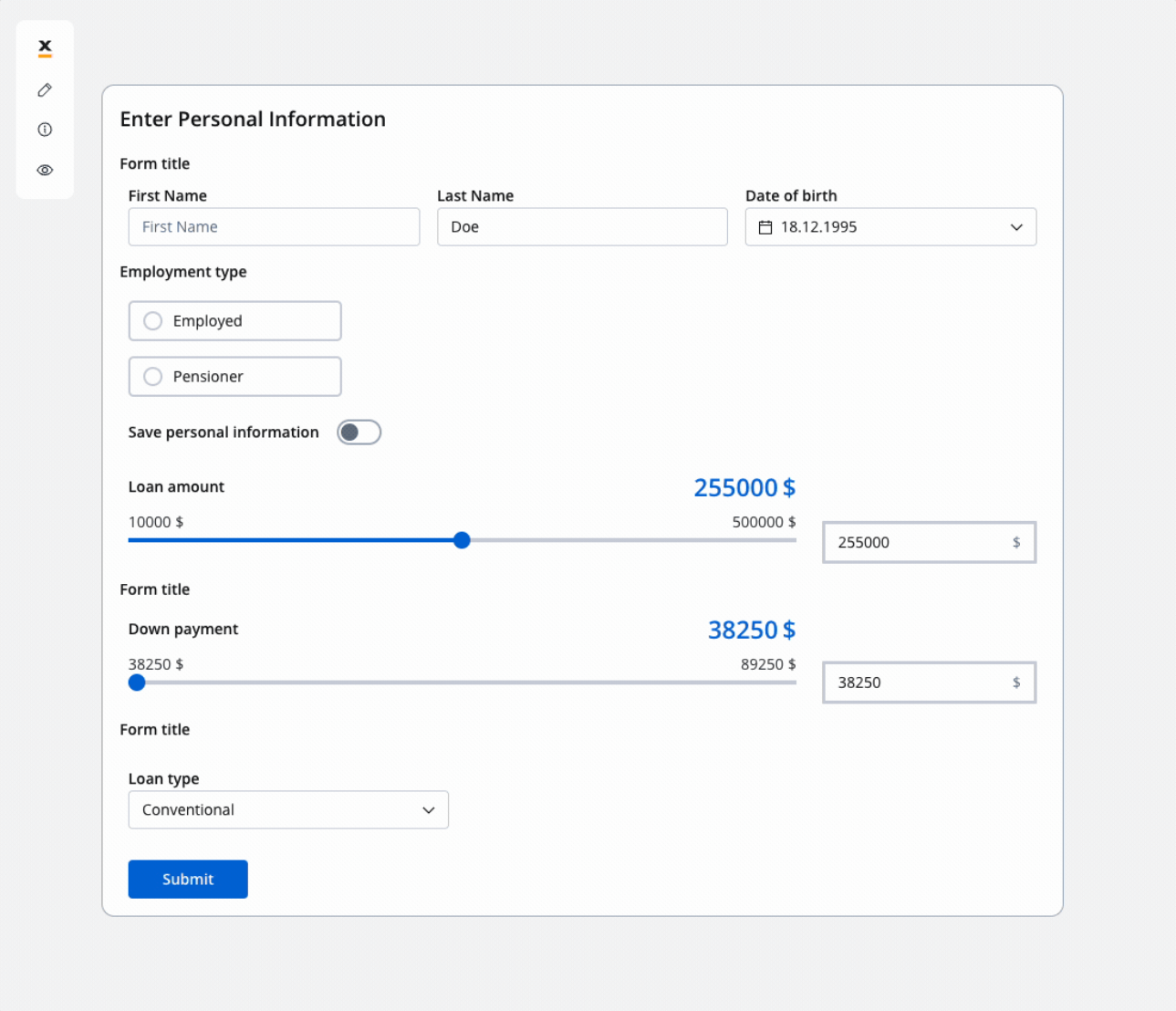
UI actions

UI actions can be added to the slider element to define its behavior and interactions.

- **Event** - possible value: CHANGE
- **Action Type** - select the action type, ❗ for more details on how to configure a UI action, click **here**

Multiple sliders

You can also use multiple sliders UI elements that are interdependent, as you can see in the following example:



INFO

You can improve the configuration of the slider using computed values as in the example above. These values provide a more flexible and powerful approach for handling complex use cases. You can find an example by referring to the following documentation:

> **Dynamic & computed values**

Slider styling

To create a slider with specific styling, sizing, typography, and color settings, you can use the following configuration:

- Sizing
- Typography
- Background

Sizing

- set the width of the button - fill/fixed/auto

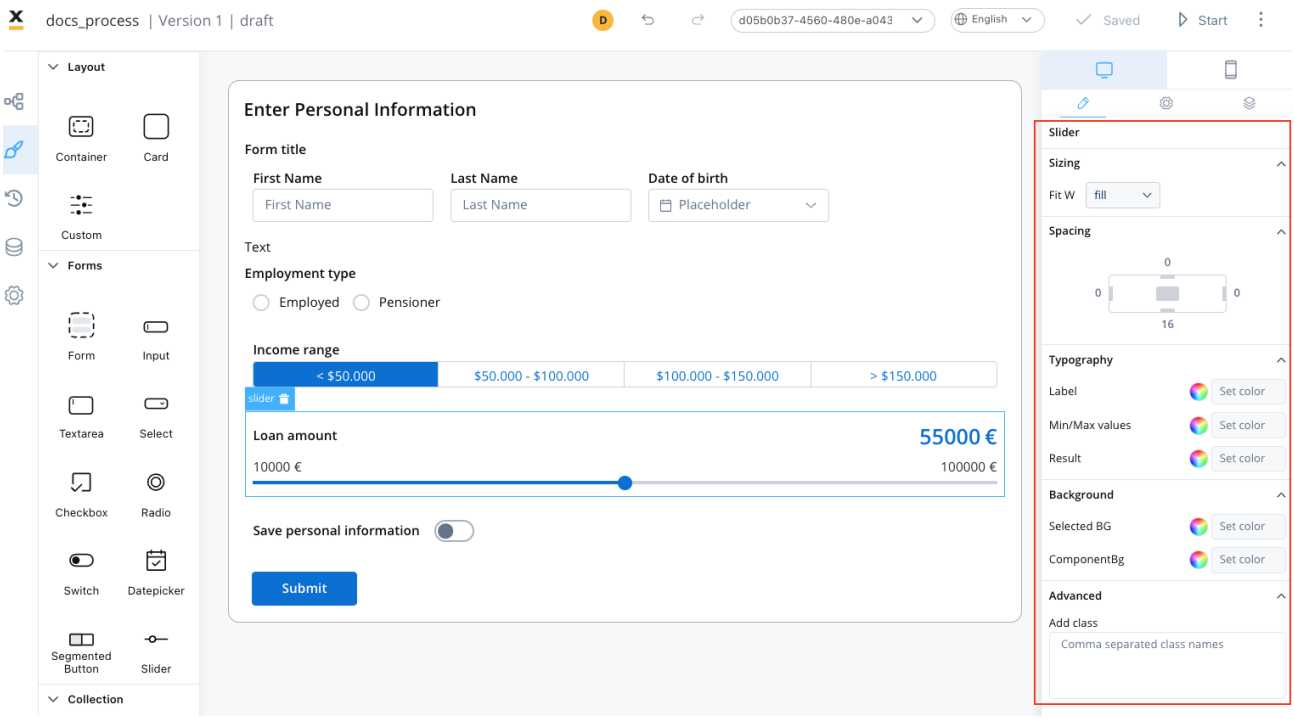
Typography

Choose an appropriate font family, size, and weight for the button label text.

- **Label Color** - set the color of the button label text
- **Min/Max values** - set the color of
- **Result** - set the color of the

Background

- **Selected BG** - set the background color of the button.
- **ComponentBg** - set the background color of the button.

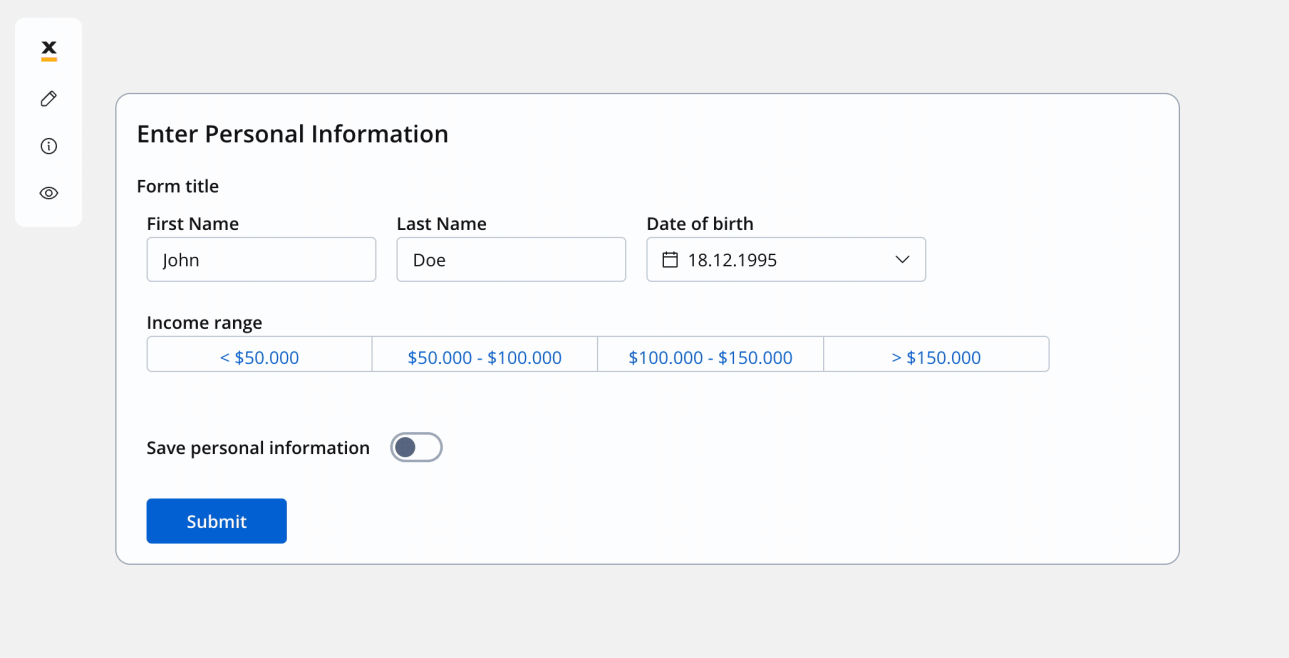


INFO

For more valid CSS properties, click **here**.

Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Form elements / Segmented button



It allows users to pick only one option from a group of options, and you can choose to have between 2 and 5 options in the group. The segmented button is easy to use, and can help make your application easier for people to use.

Configuring the segmented button

Segmented button settings

The available configuration options for this form element are:

- **General**
- **Properties**
- **Datasource**
- **Validators**
- **Expressions**

- **UI actions**
- **Segmented button styling**

General

- **Process data key** - creates the binding between form element and process data so it can be later used in decisions, business rules or integrations

Properties

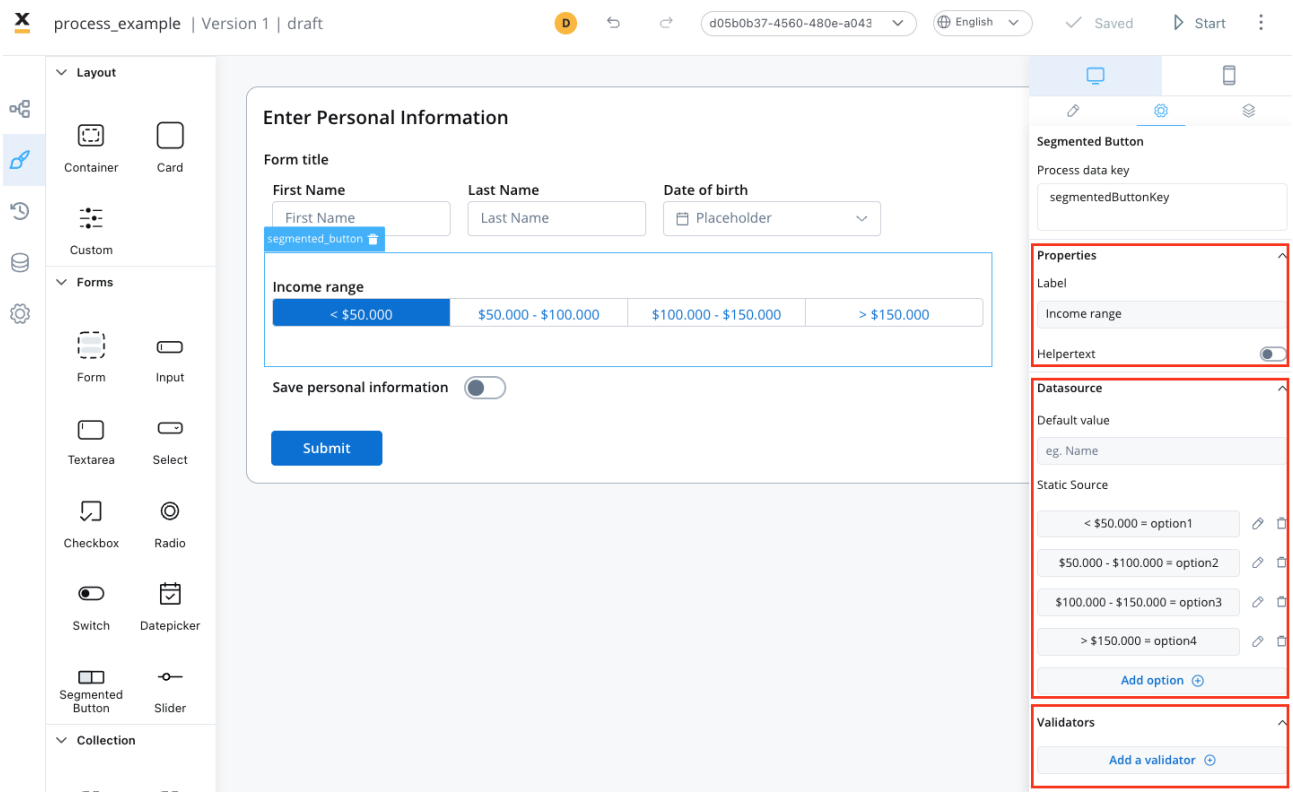
- **Label** - the label of the segmented button
- **Helpertext** - additional information about the segmented button (can be hidden inside an infopoint)

Datasource

- **Default Value** - the default value of the segmented button (it can be selected from one of the static source values)
- **Source Type** - it is by default Static
- **Add option** - value/label pairs can be defined here

Validators

The following validators can be added to a segmented button: `required` and `custom` (more details here).



UI actions

UI actions can be added to the segmented button element to define its behavior and interactions.

- **Event** - possible value: CHANGE
- **Action Type** - select the action type

> **INFO**
>
> For more details on how to configure a UI action, click **here**.

Segmented button styling

To create a segmented button with specific styling, sizing, typography, and color settings, you can use the following configuration:

- Sizing
- Typography
- Background

Sizing

- set the width of the button - fill/fixed/auto

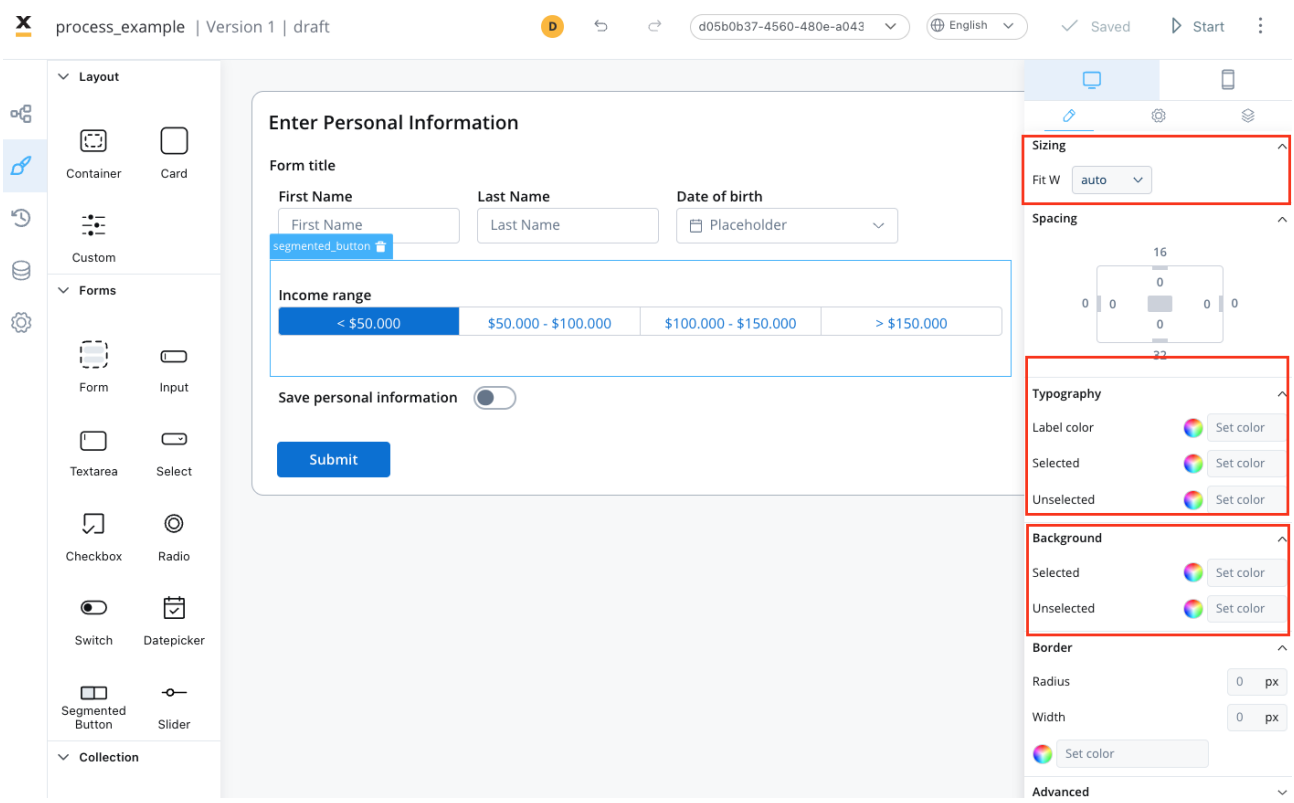
Typography

Choose an appropriate font family, size, and weight for the button label text.

- **Label Color** - set the color of the button label text
- **Selected State** - set the color of the label text when the button is selected
- **Unselected State** - set the color of the label text when the button is not selected

Background

- **Selected state** - set the background color of the button
- **Unselected state** - set the background color of the button

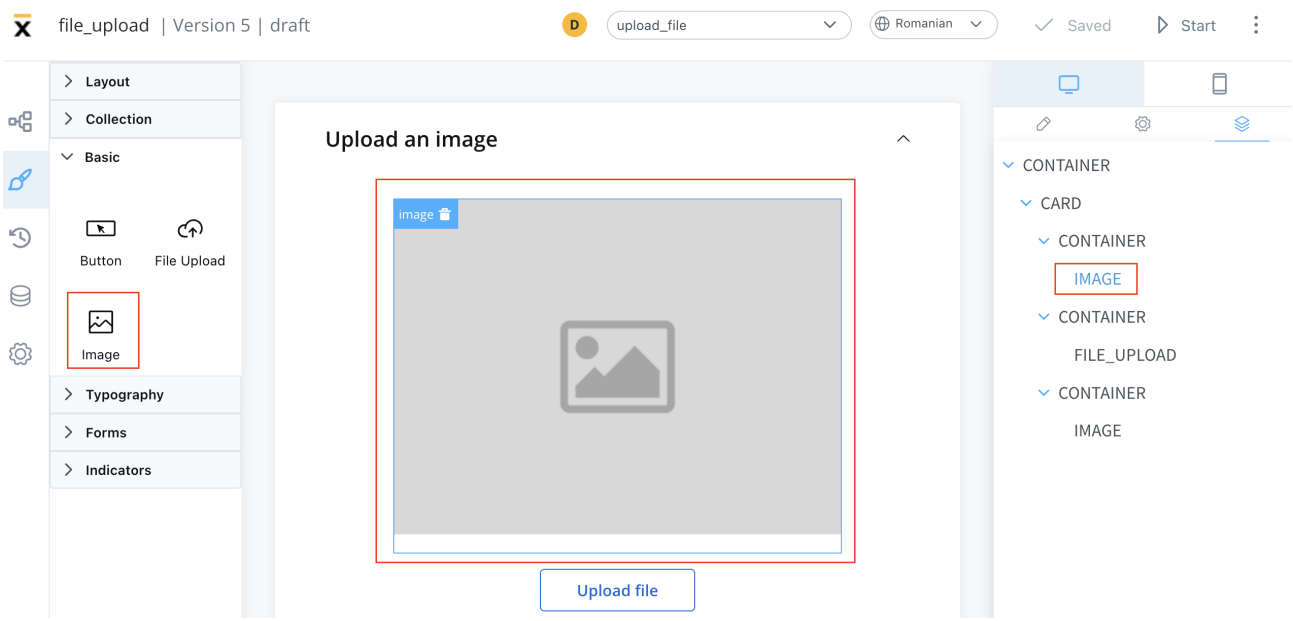


> **INFO**
>
> For more valid CSS properties, click **here**.

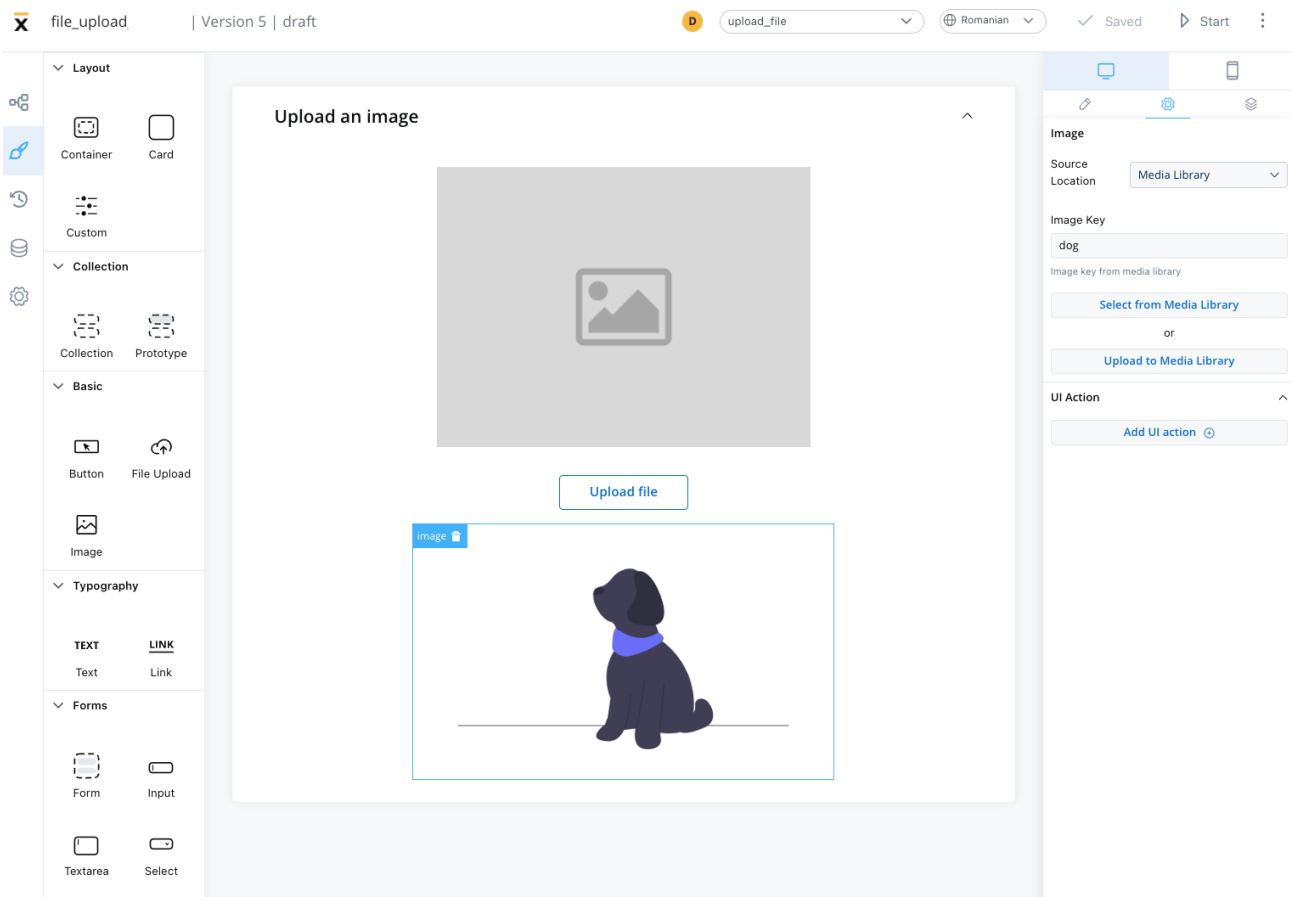
Was this page helpful?

BUILDING BLOCKS / UI Designer / UI component types / Image

Image UI elements are graphical components of a user interface that display a static or dynamic visual representation of an object, concept, or content.



These elements can be added to your interface using the UI Designer tool, and they are often used to convey information, enhance the aesthetic appeal of an interface, provide visual cues and feedback, support branding and marketing efforts, or present complex data or concepts in a more intuitive and accessible way.



Configuring an image

Configuring an image in the UI Designer involves specifying various settings and properties. Here are the key aspects of configuring an image:

Image settings

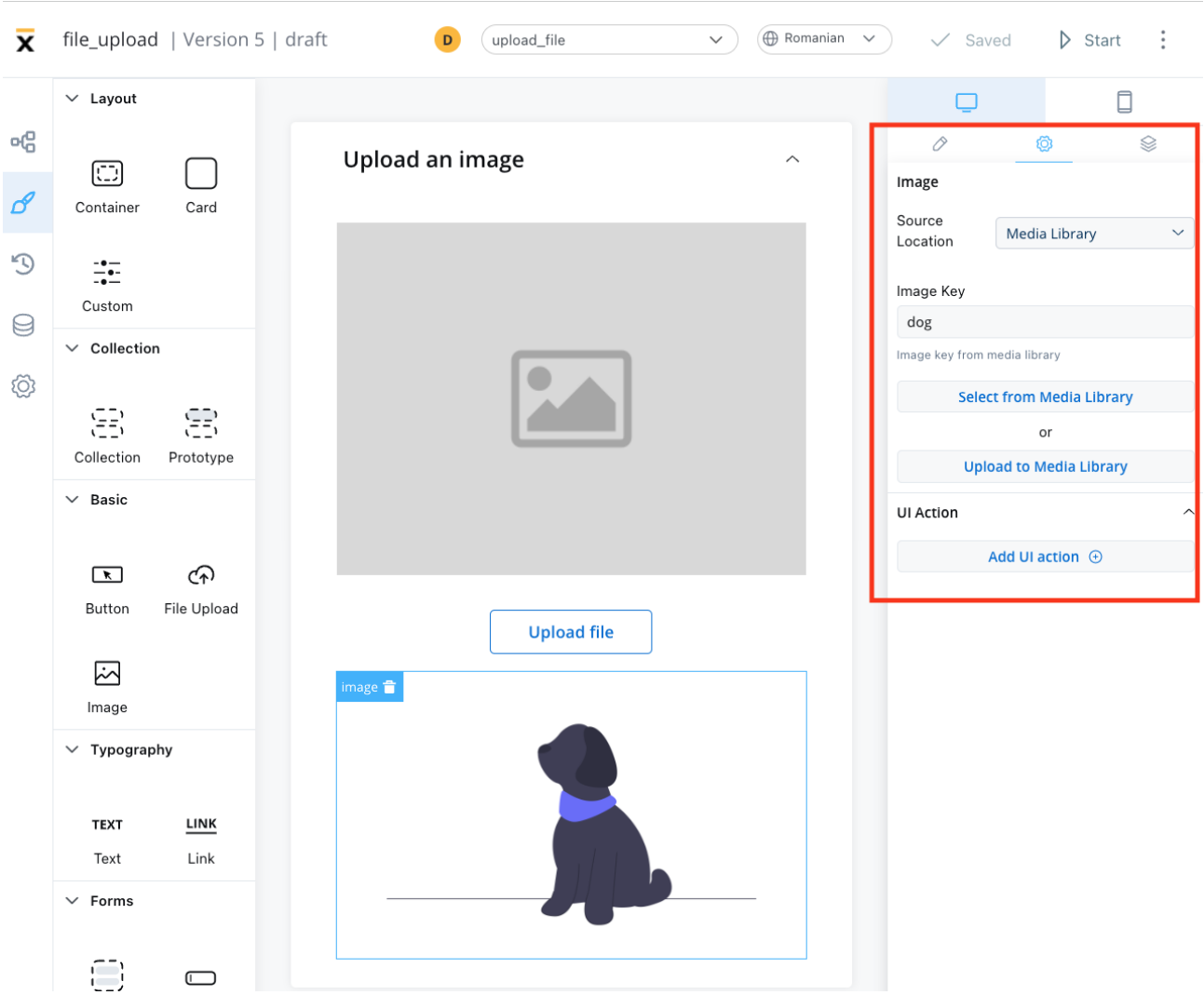
The image settings consist of the following properties:

- **Source location** - the location from where the image is loaded:
    - **Media Library**

- Process Data
- External

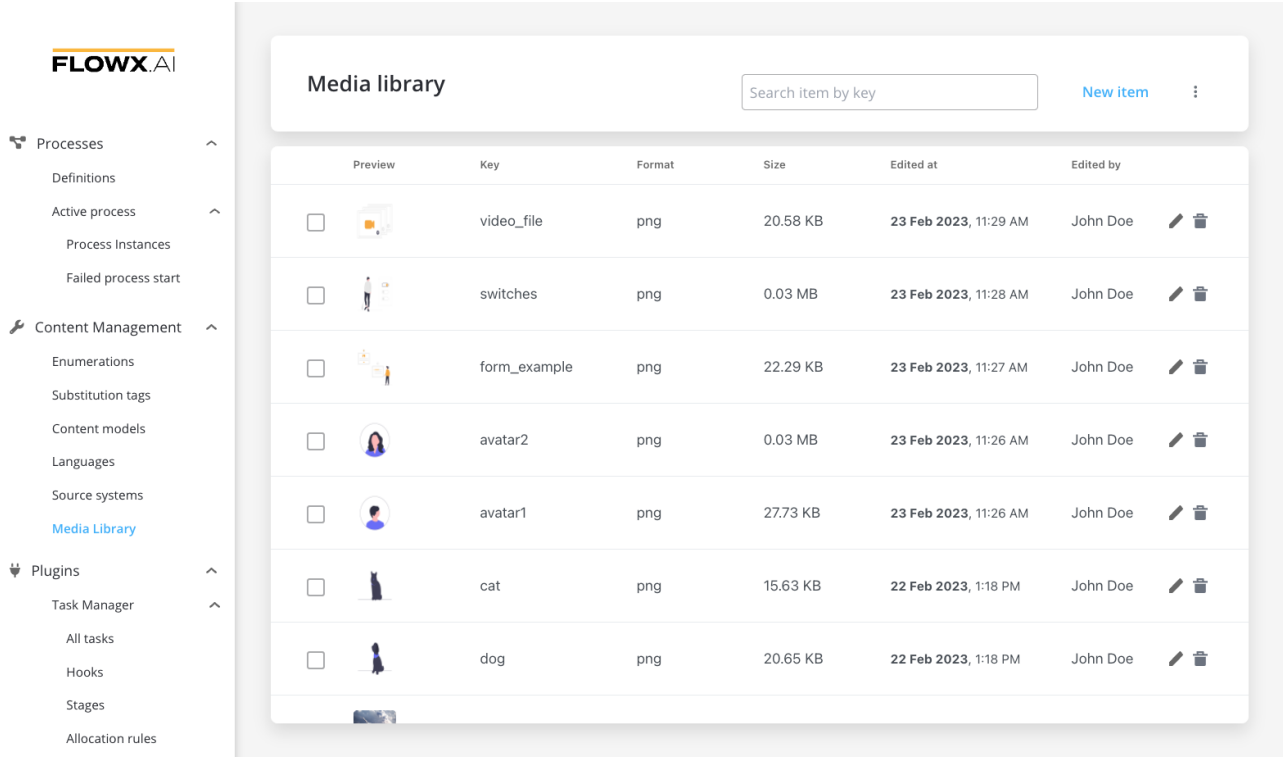
Depending on which **Source location** is selected, different configurations are available:

Media library

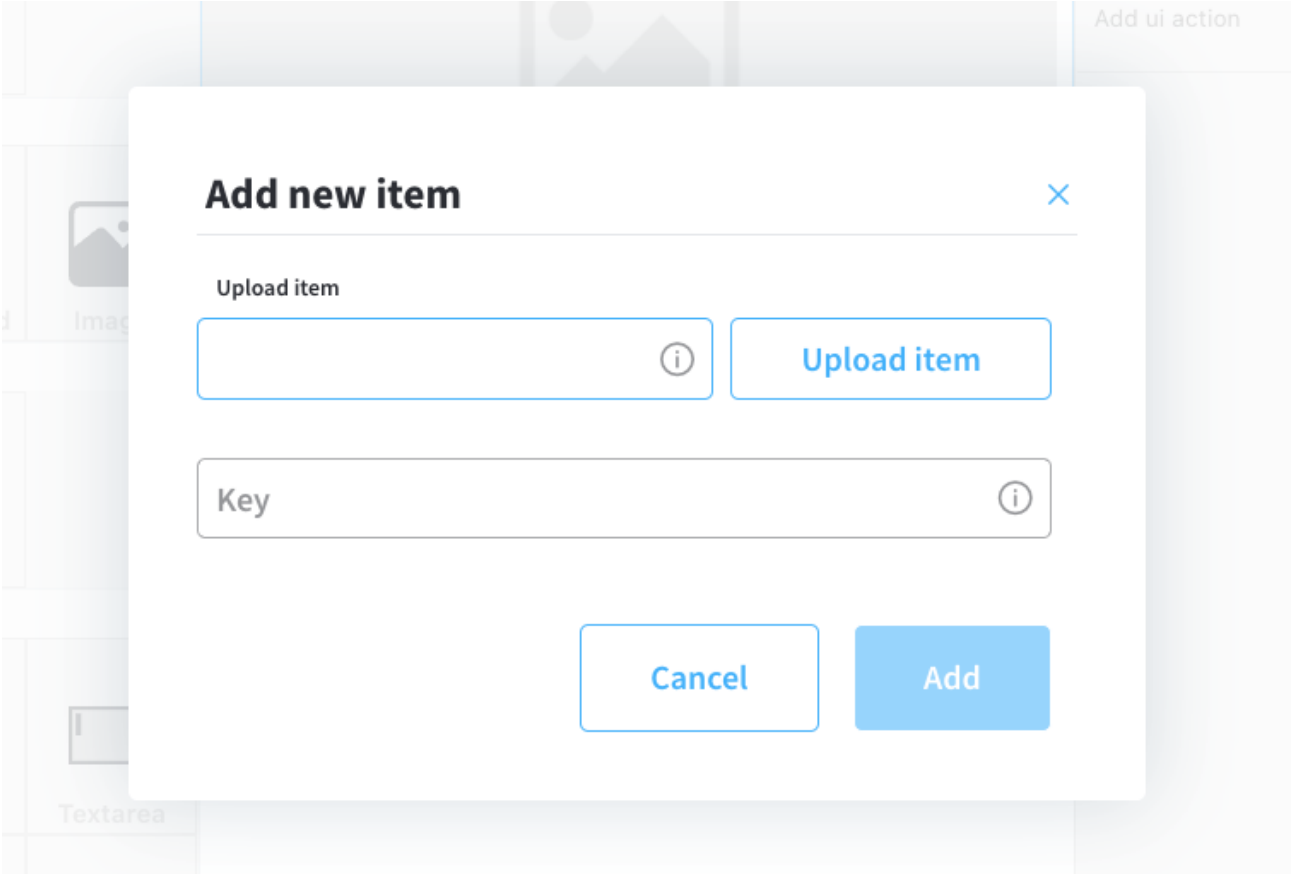


- **Image key** - the key of the image from the media library

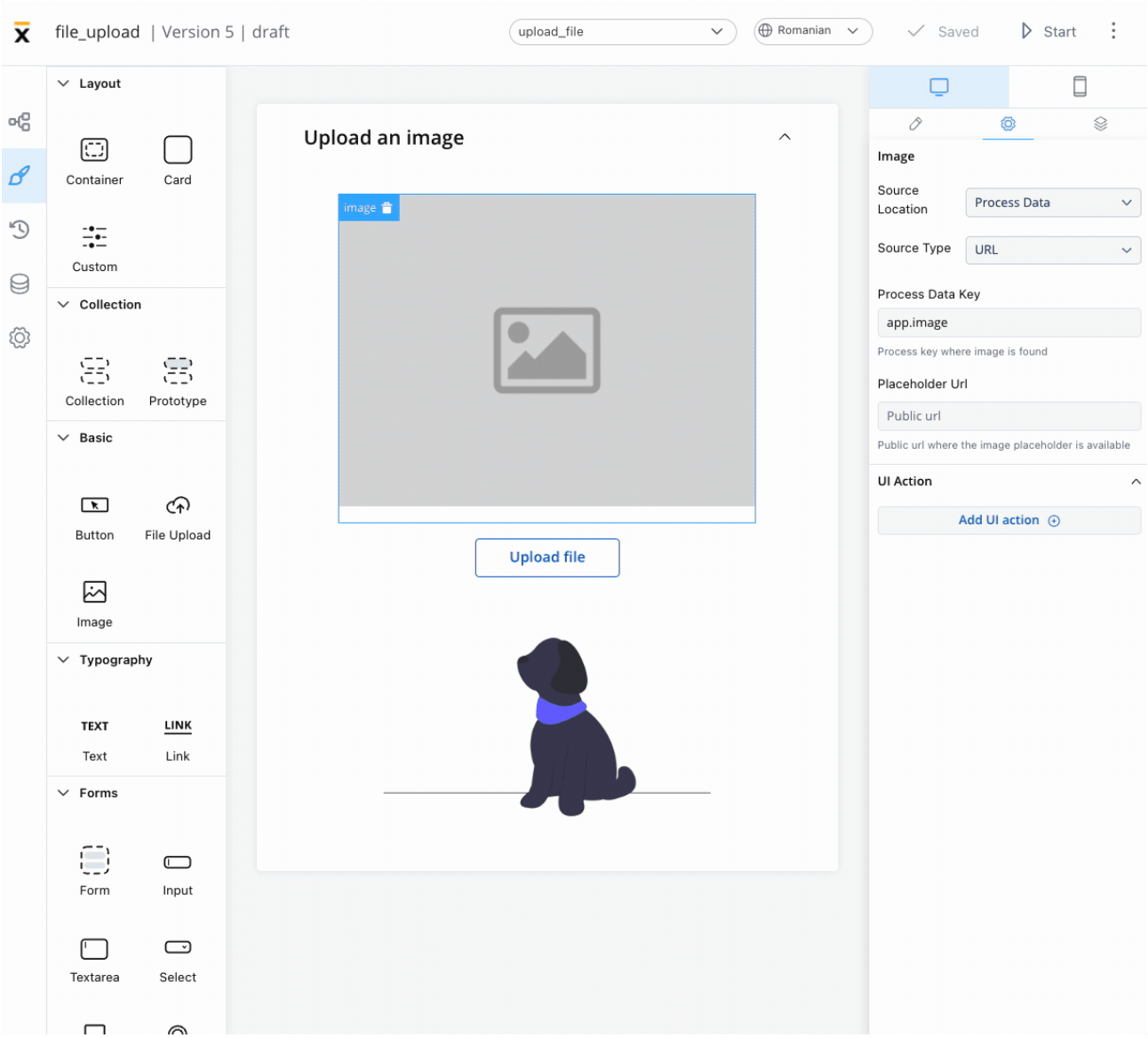
- **Select from media library** - search for an item by key and select it from the media library



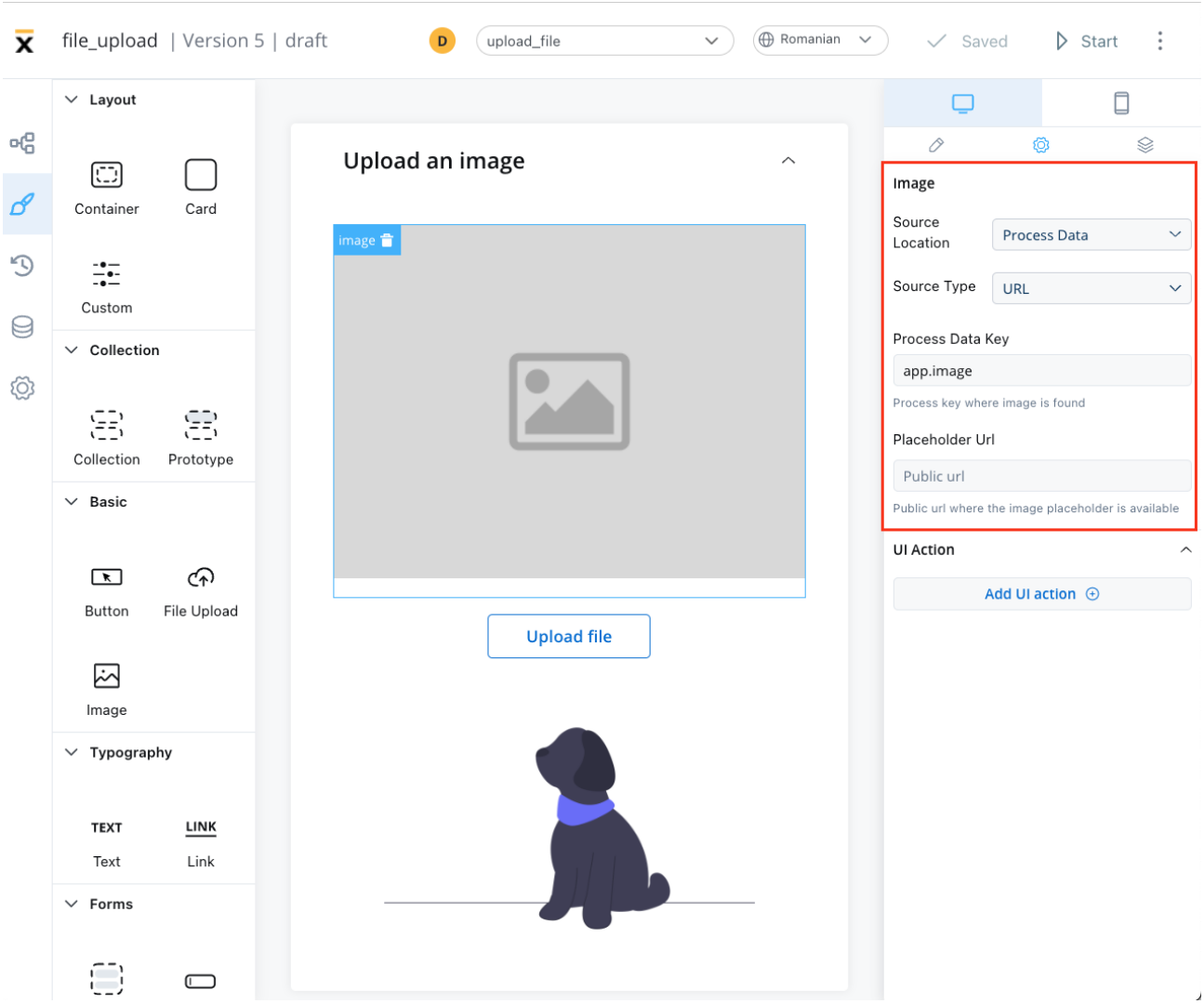
- **Upload to media library** - add a new item (upload an image on the spot)
  - **upload item** - supported formats: PNG, JPG, GIF, SVG, WebP;
    ! (maximum size - 1 MB)
  - **key** - the key must be unique and cannot be changed afterwards



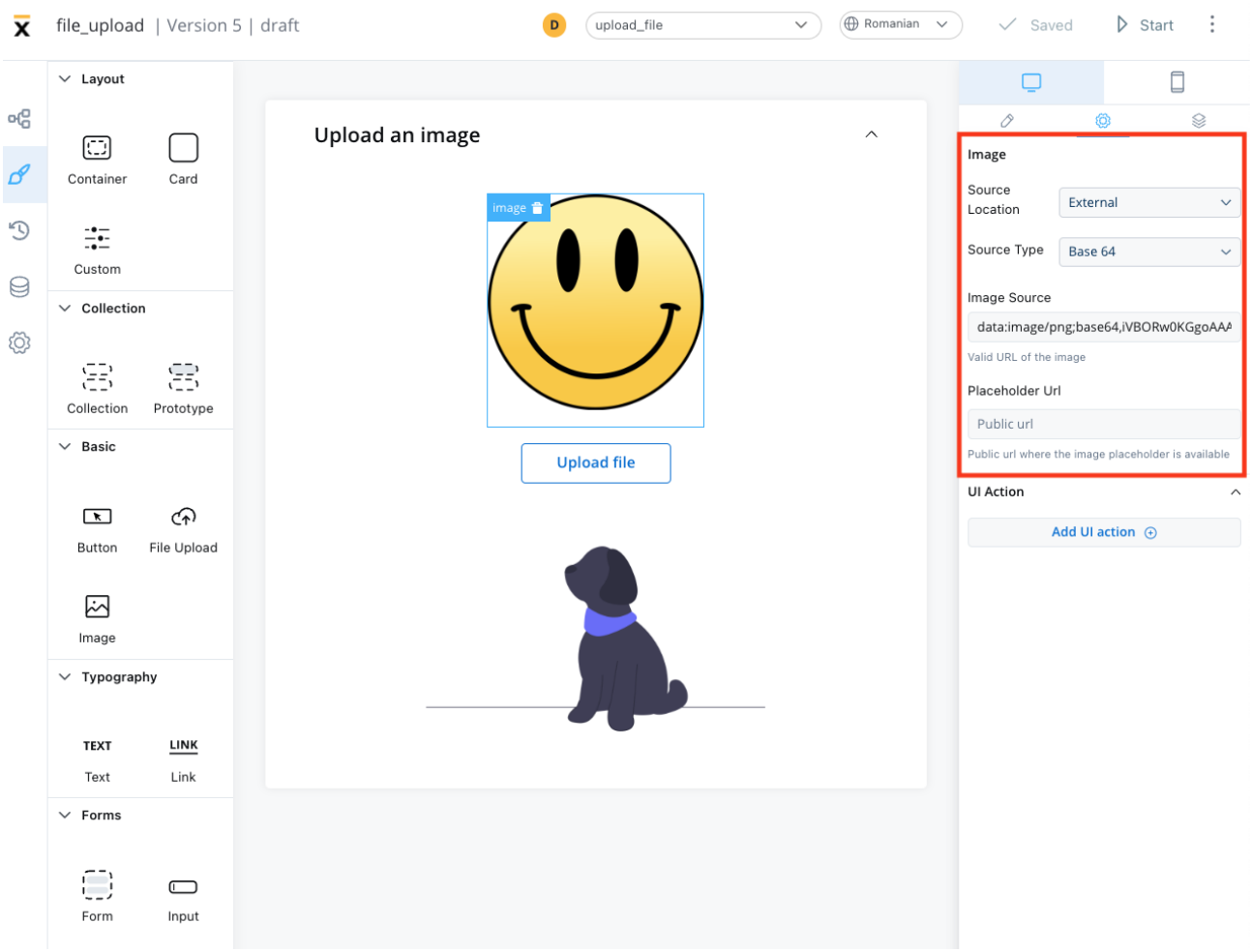
Process Data



- Identify the **Source Type**. It can be either a **URL** or a **Base 64 string**.
- Locate the data using the **Process Data Key**.
- If using a URL, provide a **Placeholder URL** for public access. This is the URL where the image placeholder is available.


file_upload | Version 5 | draft
upload_file
Romanian
Saved
Start
Layout
Container
Card
Custom
Collection
Collection
Prototype
Basic
Button
File Upload
Image
Typography
TEXT
Text
LINK
Link
Forms
Upload an image
image
Upload file
Image
Source Location
Process Data
Source Type
URL
Process Data Key
app.image
Process key where image is found
Placeholder Url
Public url
Public url where the image placeholder is available
UI Action
Add UI action

External



- **Source Type**: it can be either a **URL** or a **Base 64 string**
- **Image source**: the valid URL of the image.
- **Placeholder URL**: the public URL where the image placeholder is available

UI actions

The UI actions property allows you to add a UI Action, which must be configured on the same node. For more details on UI Actions, refer to the documentation here.

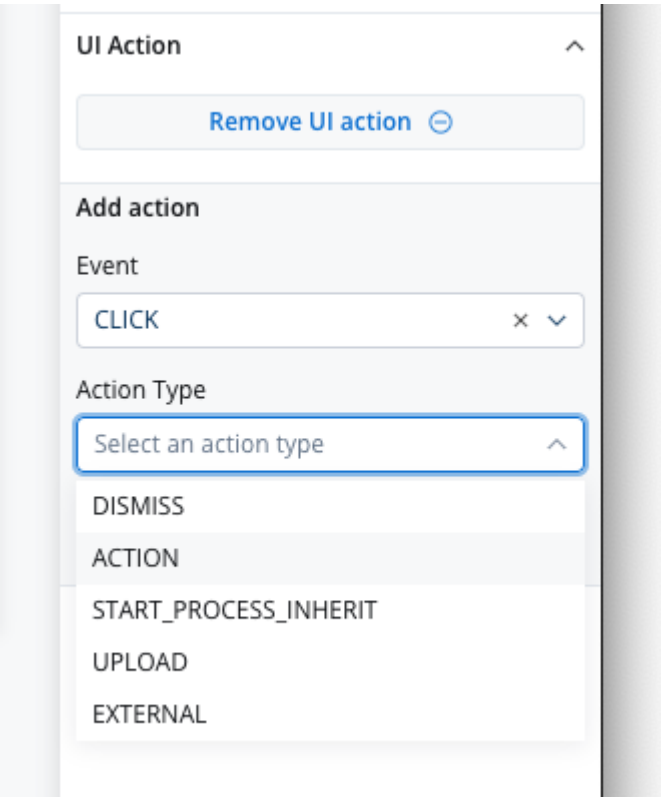
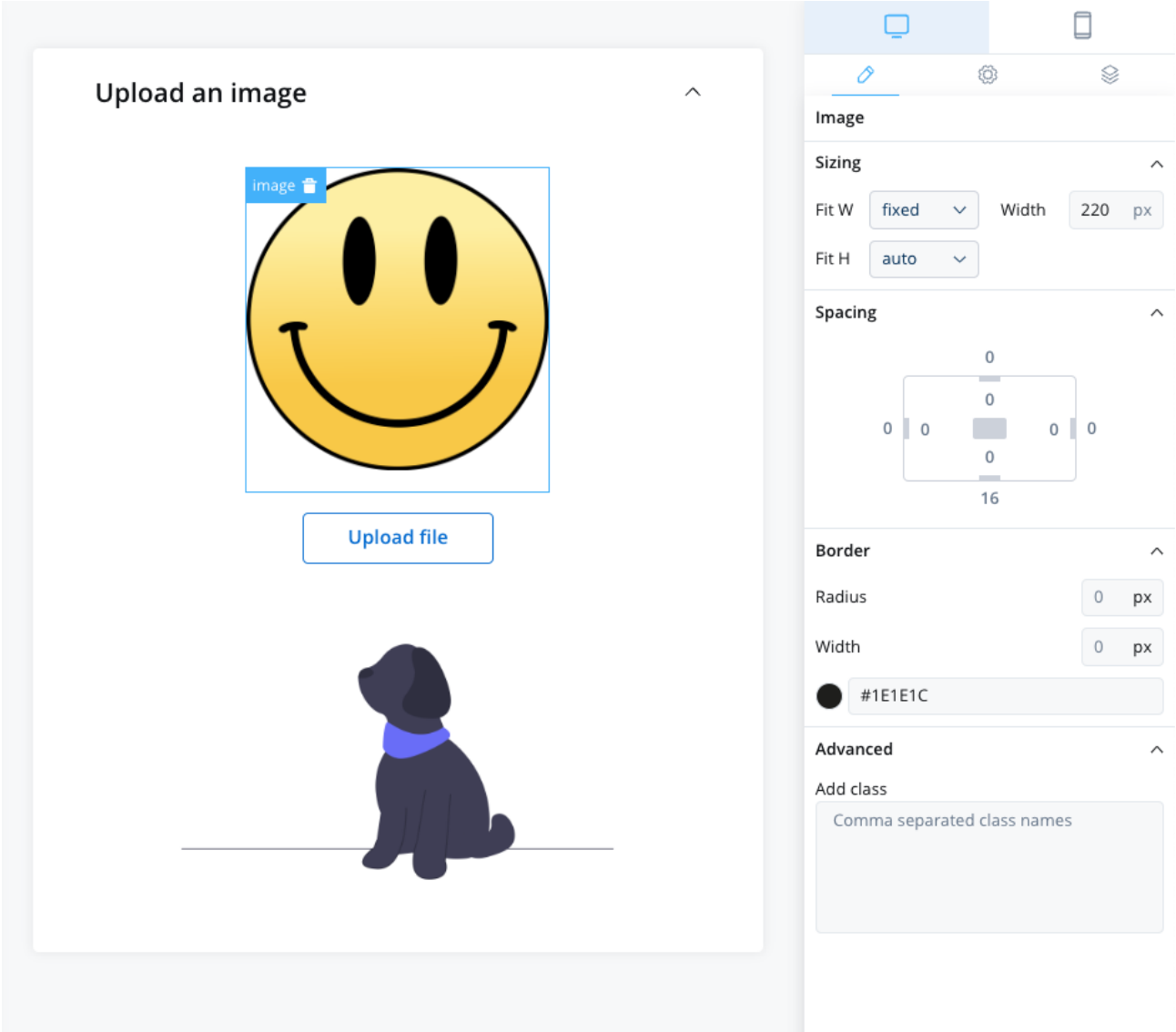


Image styling

The image styling property allows you to add or to specify valid CSS properties for the image. For more details on CSS properties, click here.



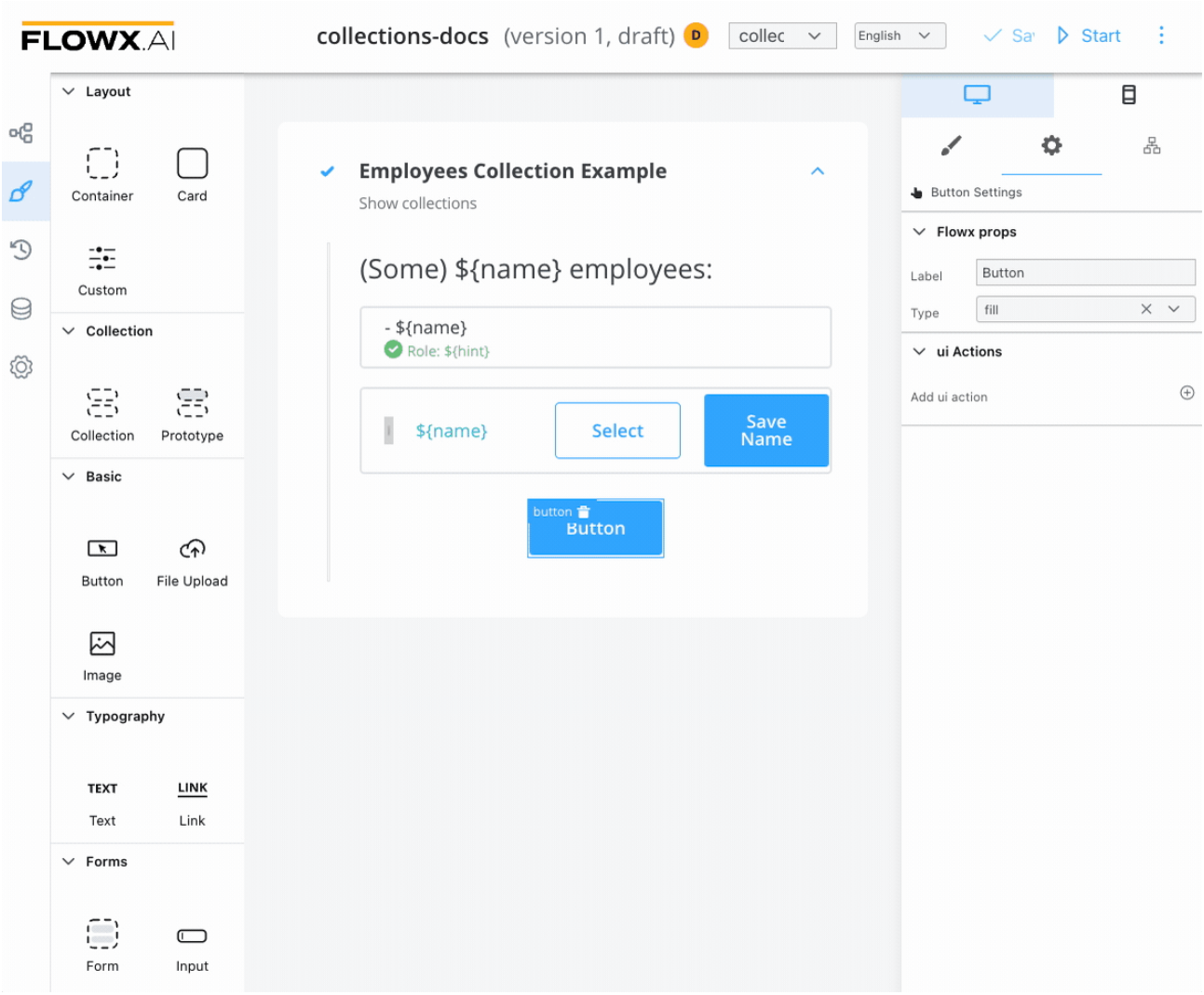
Was this page helpful?

BUILDING BLOCKS / UI Designer / UI actions

Multiple UI elements can be linked to an action via a UI Action. If the action is just a method to interact with the process, the UI Action adds information about how that UI should react. For example, should a loader appear after executing the action, should a modal be dismissed, or if some default data should be sent back to the process.

UI actions create a link between an **action** and a UI component or **custom component**.

The UI action informs the UI element to execute the given action when triggered. Other options are available for configuration when setting an action to a button and are detailed below.



There are two main types of UI Actions:

- Process UI Actions
- External UI Actions

Process UI actions

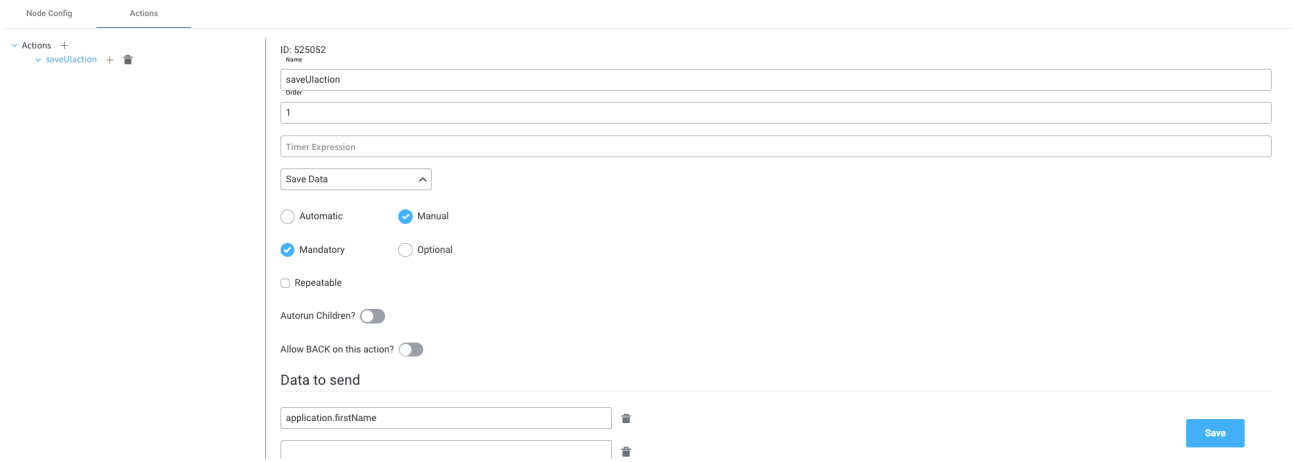
This is a UI action that describes how a **Button** (generated or custom) should interact with a process Manual action.

First, we need to configure the manual action that will be referred from the UI Action. For more information on how to add an action to a node, and how to configure it, check the following section:

» Adding an action to a node

Manual action configuration example - Save Data

1. Add an **action** to a node.
2. Select the action type - for example **Save Data**.
3. The action **type** should be **manual**.
4. **Keys** - it has two important implications:
    - First, this is a prefix of the keys that will send back by the UI Action link to this action. For example, if we have a big form with a lot of elements, but we need an action that just sends the email back (maybe creating email validation functionality) we will add just the key of that field: `application.client.email`; if we need a button that will send back all the form elements that have keys that start with application.client we can add just this part
    - Second, a backend validation will be run to accept and persist just the data that start with this prefix. If we have three explicit keys, `application.client.email`, `application.client.phone`, `application.client.address` and we send `application.client.age` this key will not be persisted



When this prerequisite is ready we can define the UI Action.

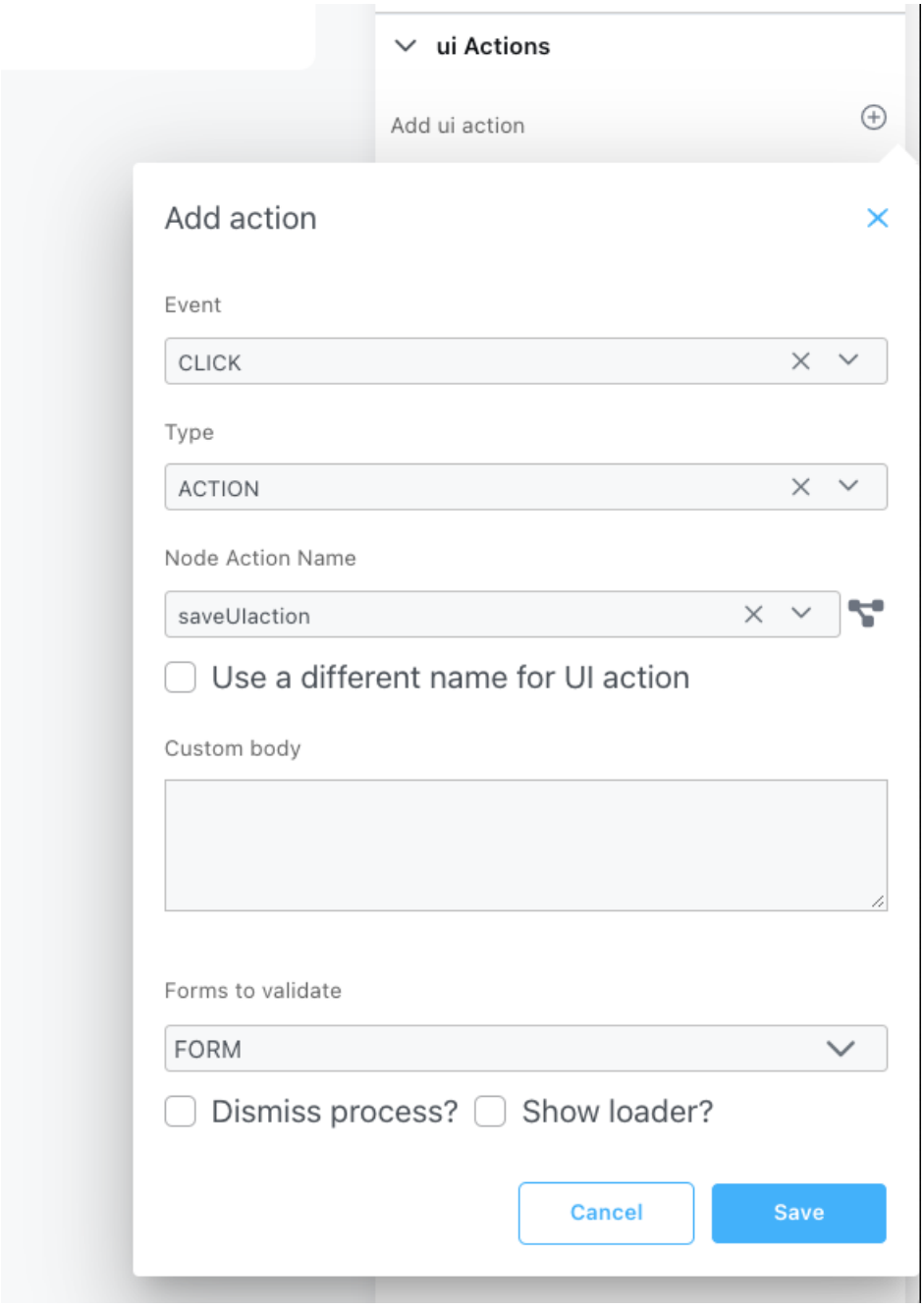
> **CAUTION**
>
> UI Actions and Actions should be configured on the same node.

UI action configuration example

Multiple configurations are available - **ACTION** type example:

- **Event**
- **Type**
- **Node Action Name** - dropdown with available actions for this node. If the dropdown is empty please add a manual action that is required before we create the UI Action
- **Use a different name for UI action**
- **UI action name** - **this becomes** important when the action is used in a **Custom component** as it will be used to trigger the action. For UI actions added on a generated button component this name is just descriptive

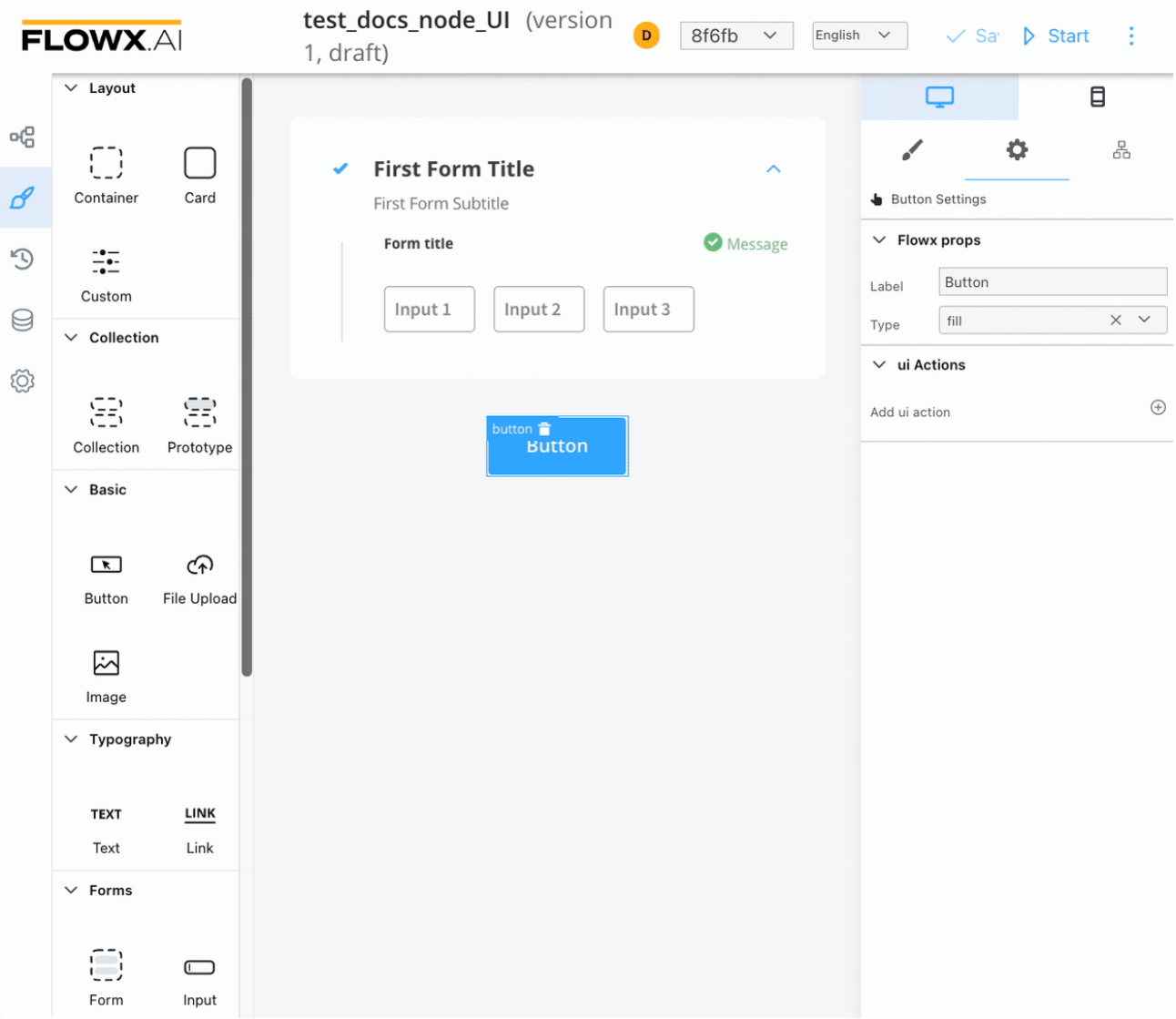
- **Custom body** - this is the default response in JSON format that will be merged with any extra parameters added explicitly when executing the action, by a web application (from a custom component)
- **Forms to validate** - select from the dropdown what element will be validated (you can also select the children)
- **Dismiss process** - if the UI Actions is added on a subprocess and this parameter is true, triggering this UI action will dismiss the subprocess view (useful for modals subprocess)
- **Show loader?** - a loader will be displayed if this option is true until a web-socket event will be received (new screen or data)


ui Actions
Add ui action
Add action
Event
CLICK
Type
ACTION
Node Action Name
saveUIaction
Use a different name for UI action
Custom body
Forms to validate
FORM
Dismiss process?
Show loader?
Cancel
Save

UI actions elements

Events

You can add an event depending on the element that you select. There are two events types available: **CLICK** and **CHANGE**.



INFO

> ! Not available for *UI Actions* on Custom Components.

Action types

The **action type** dropdown will be pre-filled with the following UI action types:

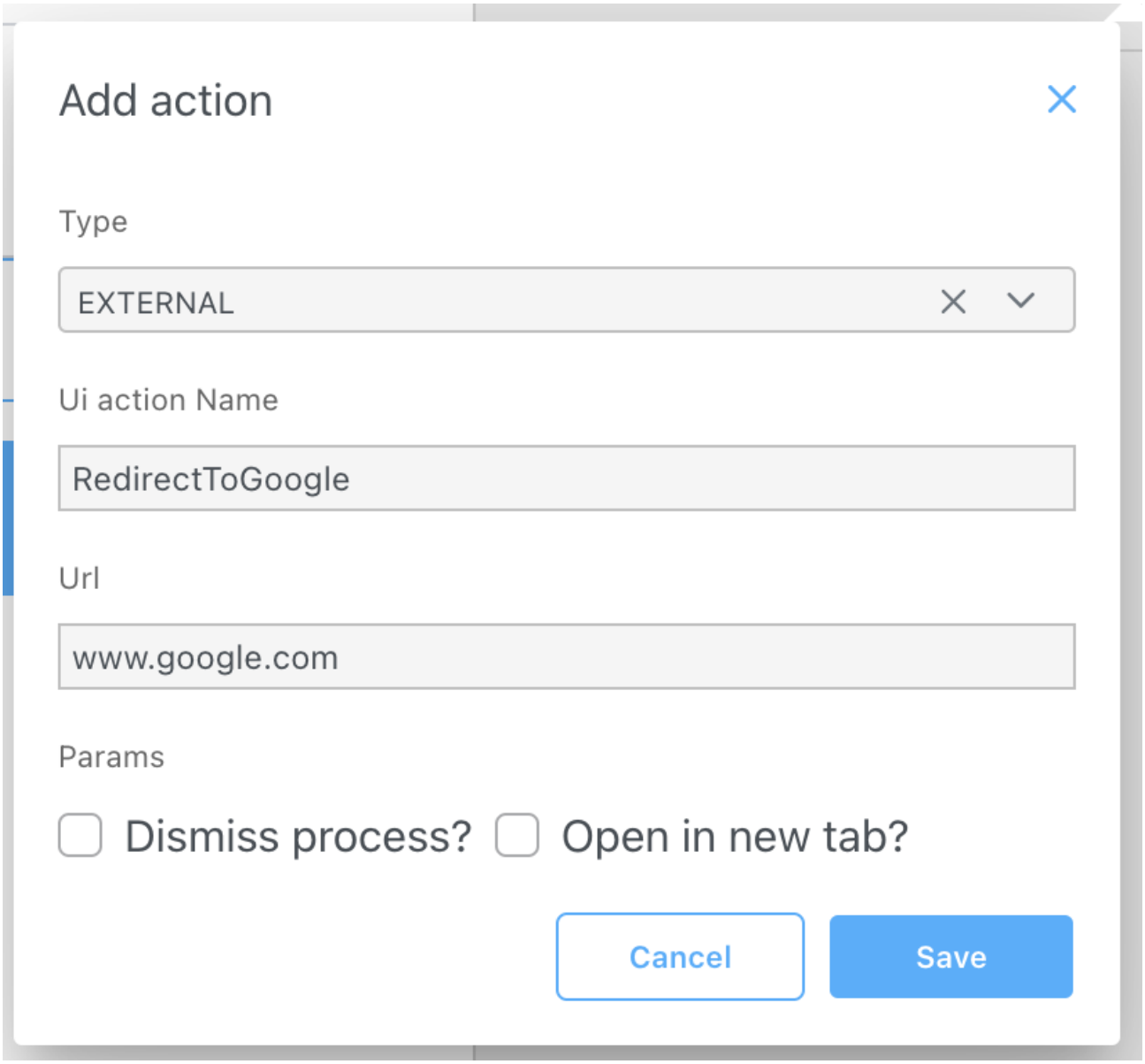
- DISMISS - used to dismiss a modal after action execution
- ACTION - used to link an action to a UI action
- START_PROCESS_INHERIT - used to inherit data from another process
- UPLOAD - used to create an un upload action
- EXTERNAL - used to create an action that will open a link in a new tab

External UI actions

Used to create an action that will open a link in a new tab.

If we toggle the EXTERNAL type, a few new options are available:

1. **URL** - web URL that will be used for the external action
2. **Open in new tab** - this option will be available to decide if we want to run the action in the current tab or open a new one



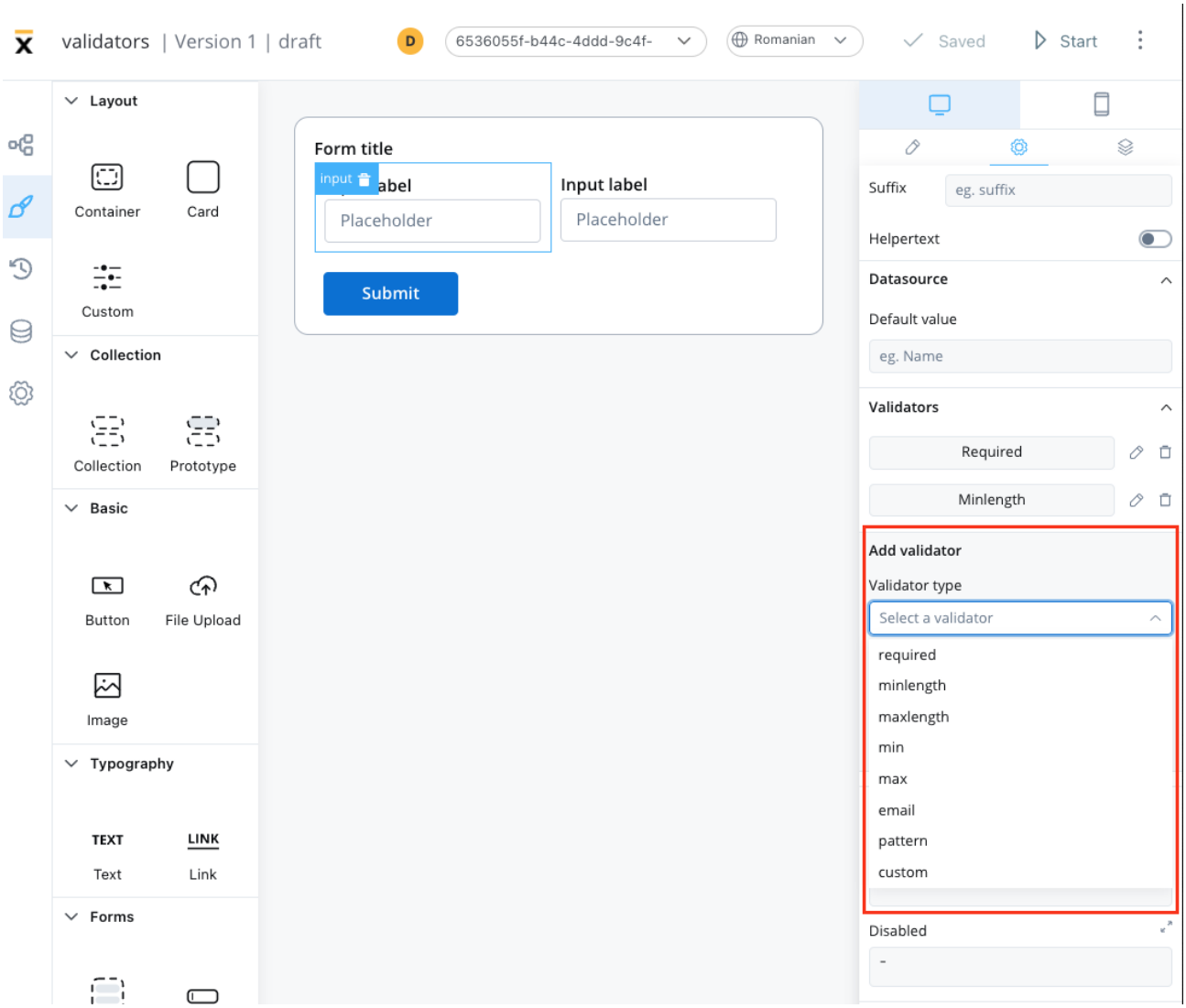
For more information on how to add actions and how to configure a UI, check the following section:

» Managing a process flow

Was this page helpful?

BUILDING BLOCKS / UI Designer / Validators

Validators are an essential part of building robust and reliable applications. They ensure that the data entered by the user is accurate, complete, and consistent. In Angular applications, validators provide a set of pre-defined validation rules that can be used to validate various form inputs such as text fields, number fields, email fields, date fields, and more.



Angular provides default validators such as:

1. min
2. max
3. minLength
4. maxLength
5. required
6. email

Other predefined validators are also available:

1. `isSameOrBeforeToday`: validates that a datepicker value is in the past
2. `isSameOrAfterToday`: validates that a datepicker value is in the future

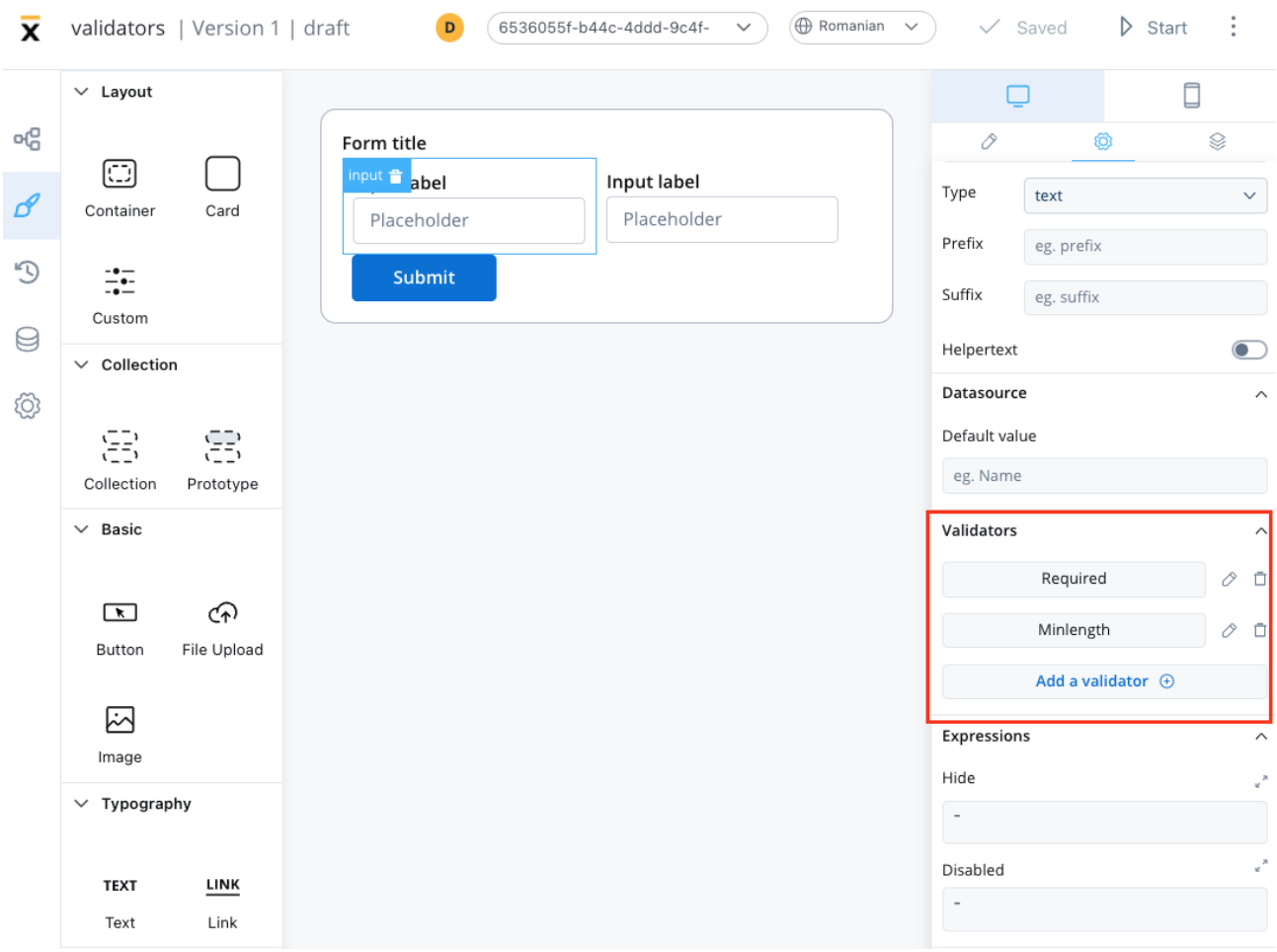
> **INFO**
>
> Form validation is triggered by default when the button set to validate a **form** is pressed.

It's also possible to build custom validators inside the container application and reference them here by name.

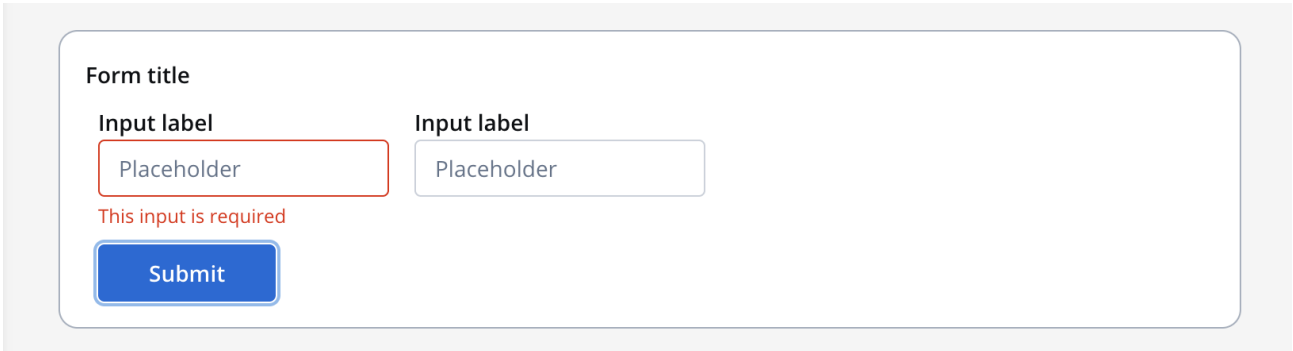
Predefined validators

required validator

This validator checks whether a value exists in the input field.

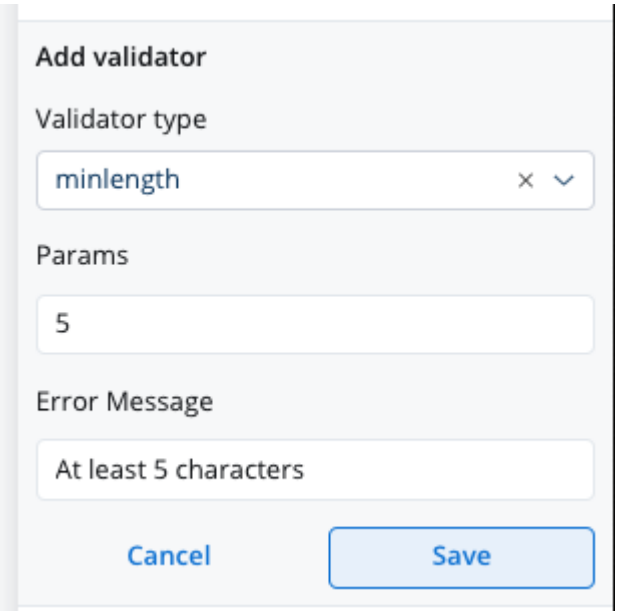


It is recommended to use this validator with other validators like minlength to check if there is no value at all.



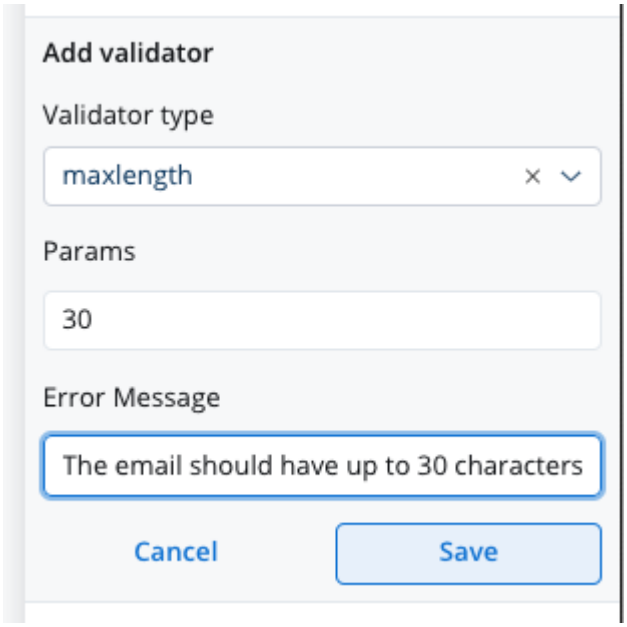
minlength validator

This validator checks whether the input value has a minimum number of characters. If there are no characters at all, this validator will not trigger. It is advisable to use this validator with a required validator.



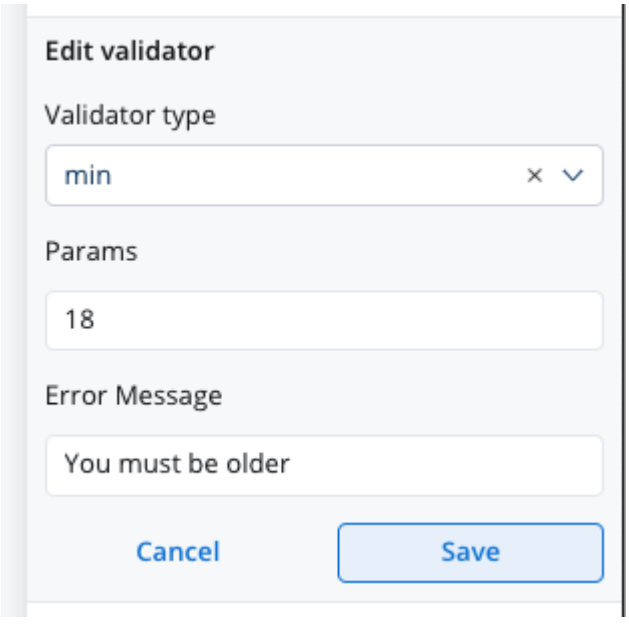
maxlength validator

This validator checks whether the input value has a maximum number of characters. If there are no characters at all, this validator will not trigger. It is advisable to use this validator with a required validator.



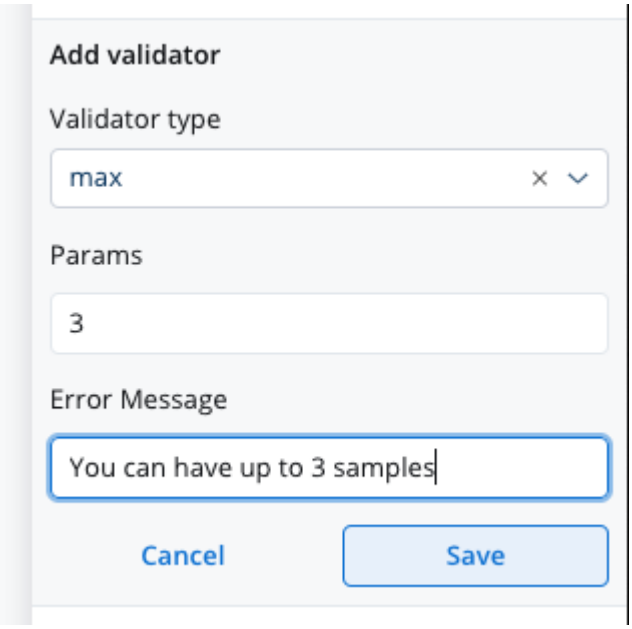
min validator

This validator checks whether a numeric value is smaller than the specified value. If there are no characters at all, this validator will not trigger. It is advisable to use this validator with a required validator.



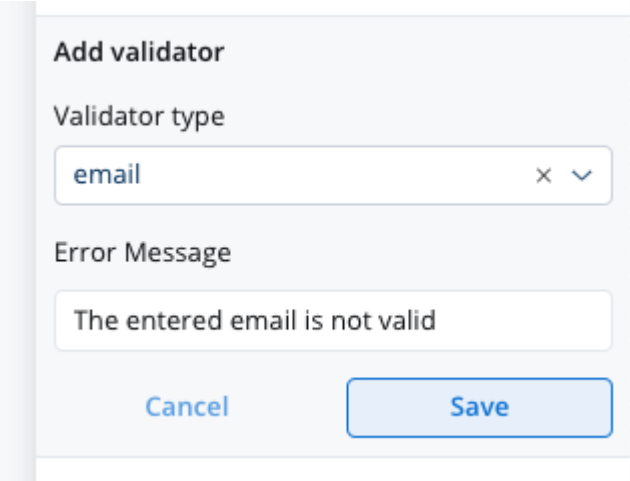
max validator

This validator checks whether a numeric value is larger than the specified value. If there are no characters at all, this validator will not trigger. It is advisable to use this validator with a required validator.



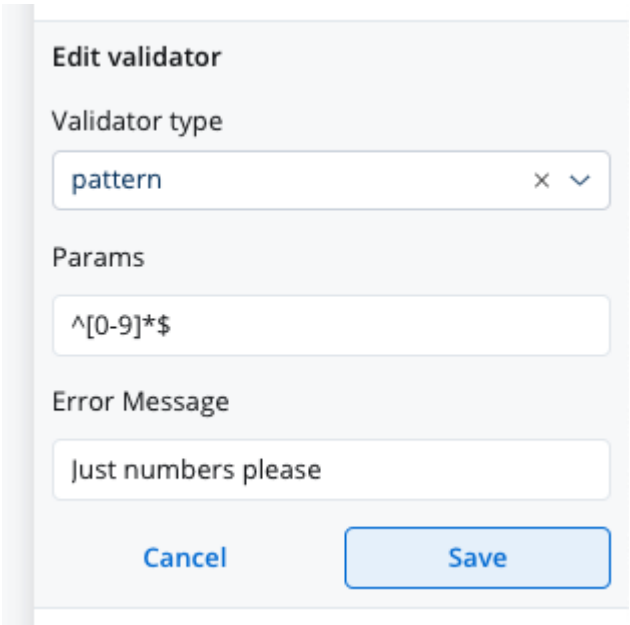
email validator

This validator checks whether the input value is a valid email. If there are no characters at all, this validator will not trigger. It is advisable to use this validator with a required validator.



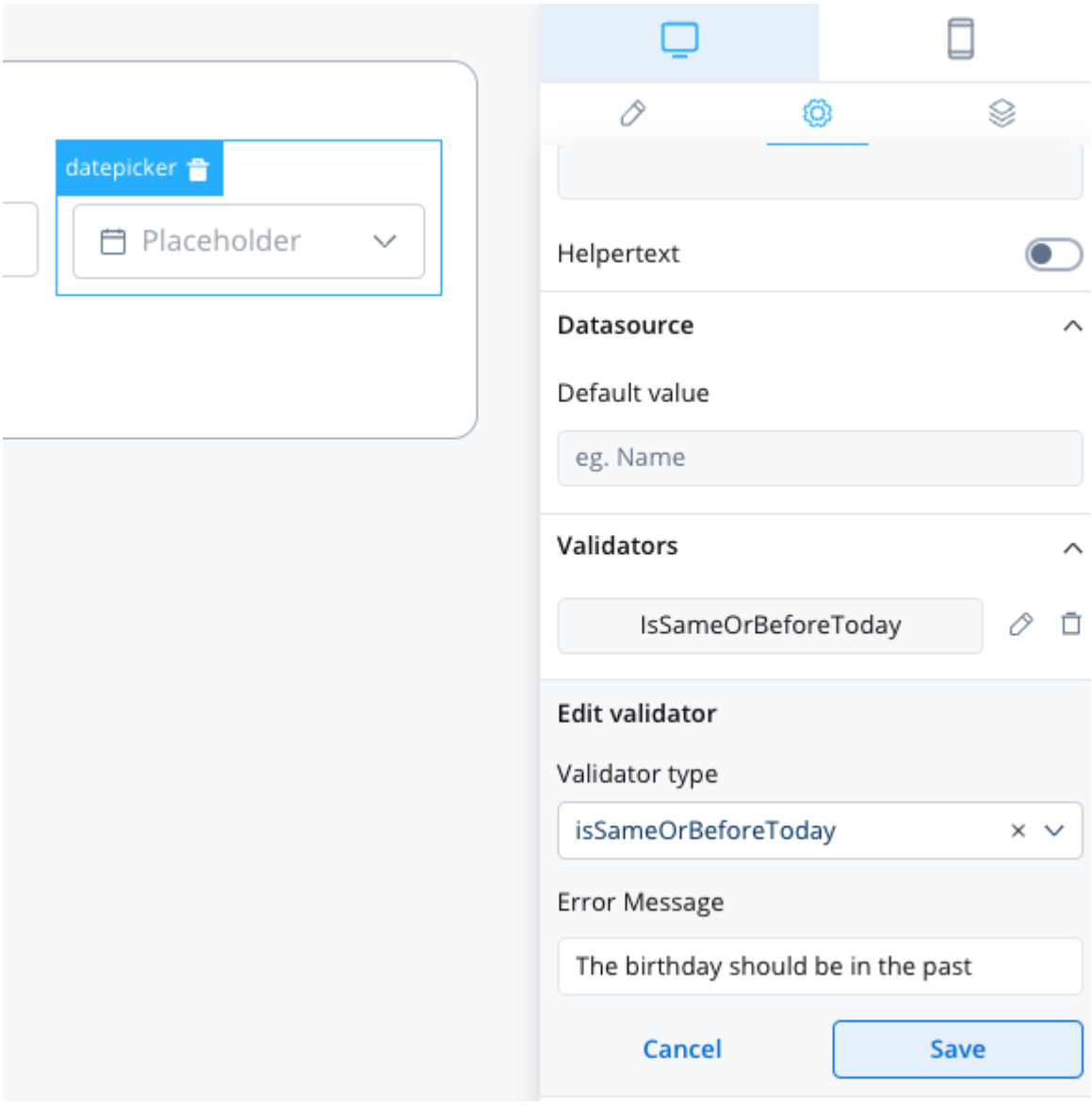
pattern validator

This validator checks whether the input value matches the specified pattern (for example, a regex expression).



datepicker - isSameOrBeforeToday

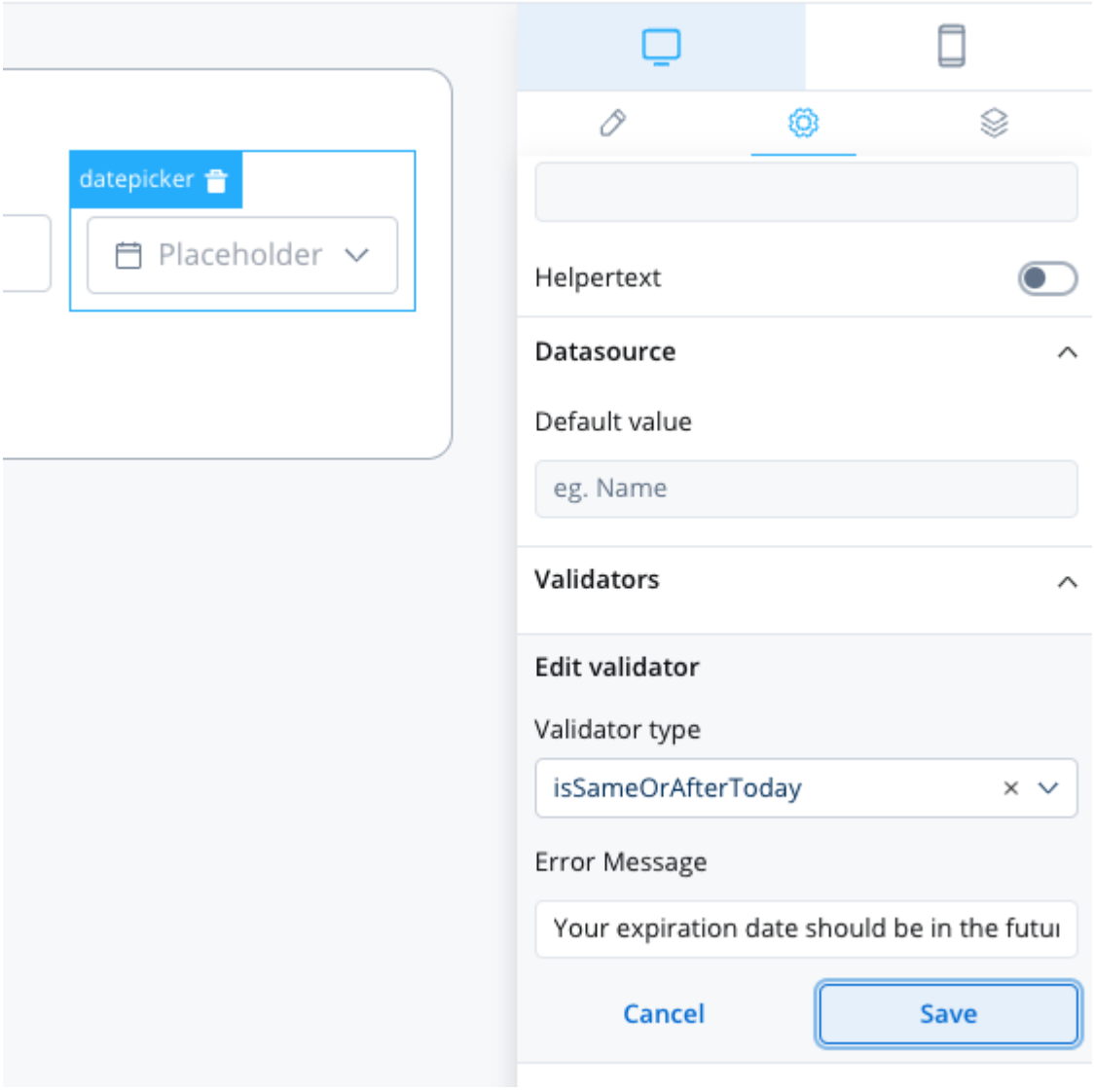
This validator can be used to validate datepicker inputs. It checks whether the selected date is today or in the past. If there are no characters at all, this validator will not trigger. It is advisable to use this validator with a required validator.



datepicker - isSameOrAfterToday

This validator can be used to validate datepicker inputs. It checks whether the selected date is today or in the future. If there are no characters at all, this

validator will not trigger. It is advisable to use this validator with a required validator.

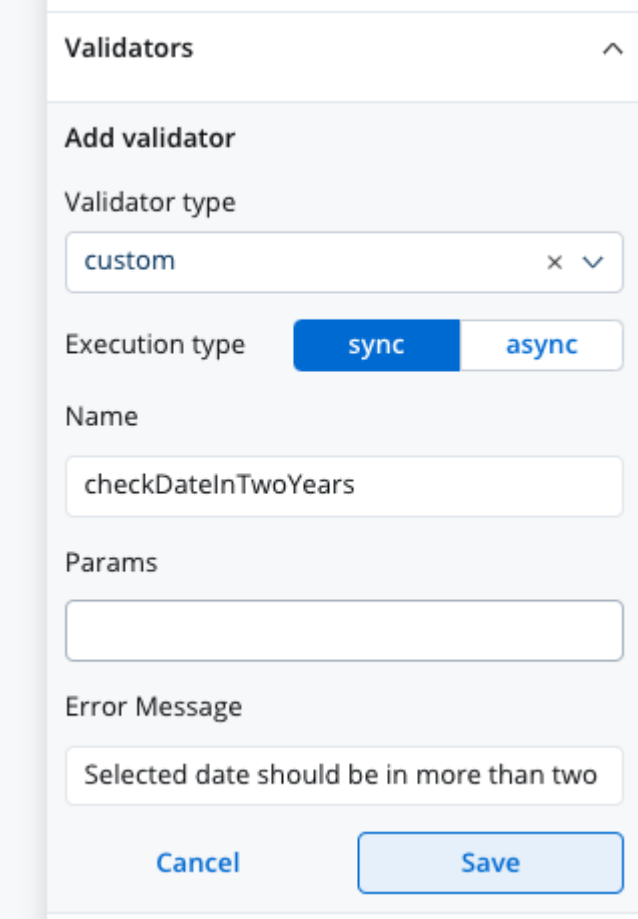


Custom validators

Additionally, custom validators can be created within the web application and referenced by name. These custom validators can have various configurations such as execution type, name, parameters, and error message.

1. **Execution type** - sync/async validator (for more details check this)
2. **Name** - name provided by the developer to uniquely identify the validator
3. **Params** - if the validator needs inputs to decide if the field is valid or not, you can pass them using this list
4. **Error Message** - the message that will be displayed if the field is not valid

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



Custom validator example

Below you can find an example of a custom validator (`currentOrLastYear`) that restricts data selection to the current or the previous year:

currentOrLastYear

```
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;
```

smallerOrEqualsToNumber

Below is another custom validator example that returns `AsyncValidatorFn` param, which is a function that can be used to validate form input asynchronously. The validator is called `smallerOrEqualsToNumber` and takes an array of `params---

sidebar_position: 3

as an input.

> **INFO**
>
> For this custom validator the execution type should be marked as `async` using the UI Designer.

```
export function smallerOrEqualsToNumber (params$:
Observable<any>[]): AsyncValidatorFn {
  return (AC): Promise<ValidationErrors | null> |
Observable<ValidationErrors | null> => {
    return new Observable((observer) => {
      combineLatest(params$).subscribe(([maximumLoanAmount])
=> {
        const validationError =
          maximumLoanAmount === undefined || !AC.value ||
Number(AC.value) <= maximumLoanAmount ? null :
{smallerOrEqualsToNumber: true};

        observer.next(validationError);
        observer.complete();
      });
    });
  };
}
```

If the input value is undefined or the input value is smaller or equal to the maximum loan amount value, the function returns `null`, indicating that the input is valid. If the input value is greater than the maximum loan amount value, the

function returns a `ValidationErrors` object with a key `smallerOrEqualsToNumber` and a value of true, indicating that the input is invalid.

> **INFO**
>
> For more details about custom validators please check this link.

Using validators in your application can help ensure that the data entered by users is valid, accurate, and consistent, improving the overall quality of your application.

It can also help prevent errors and bugs that may arise due to invalid data, saving time and effort in debugging and fixing issues.

Overall, enforcing data validation using validators is a crucial step in building high-quality, reliable, and user-friendly applications.

Was this page helpful?

BUILDING BLOCKS / UI Designer / Dynamic & computed values

In modern application development, the ability to create dynamic and interactive user interfaces is essential for delivering personalized and responsive experiences to users. Dynamic values and computed values are powerful features that enable developers to achieve this level of flexibility and interactivity.

Dynamic values

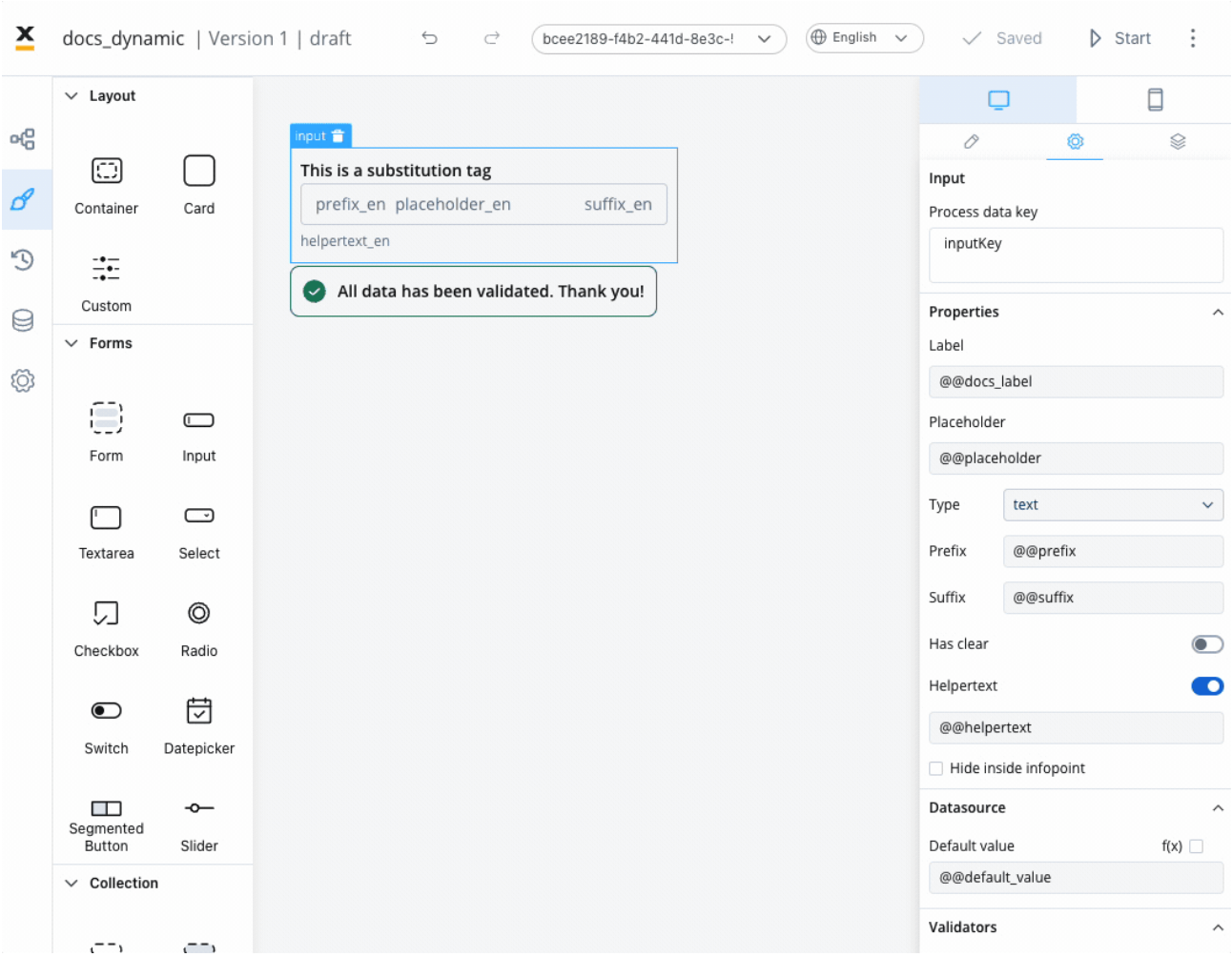
Dynamic values refer to the capability of dynamically populating element properties in the user interface based on process parameters or substitution tags. These values can be customized at runtime, allowing the application to adapt to specific scenarios or user input. With dynamic values, you can personalize labels, placeholders, error messages, and other properties of UI elements, providing a tailored experience for users.

You can now utilize process parameters or **substitution tags** with the following UI elements and their properties:

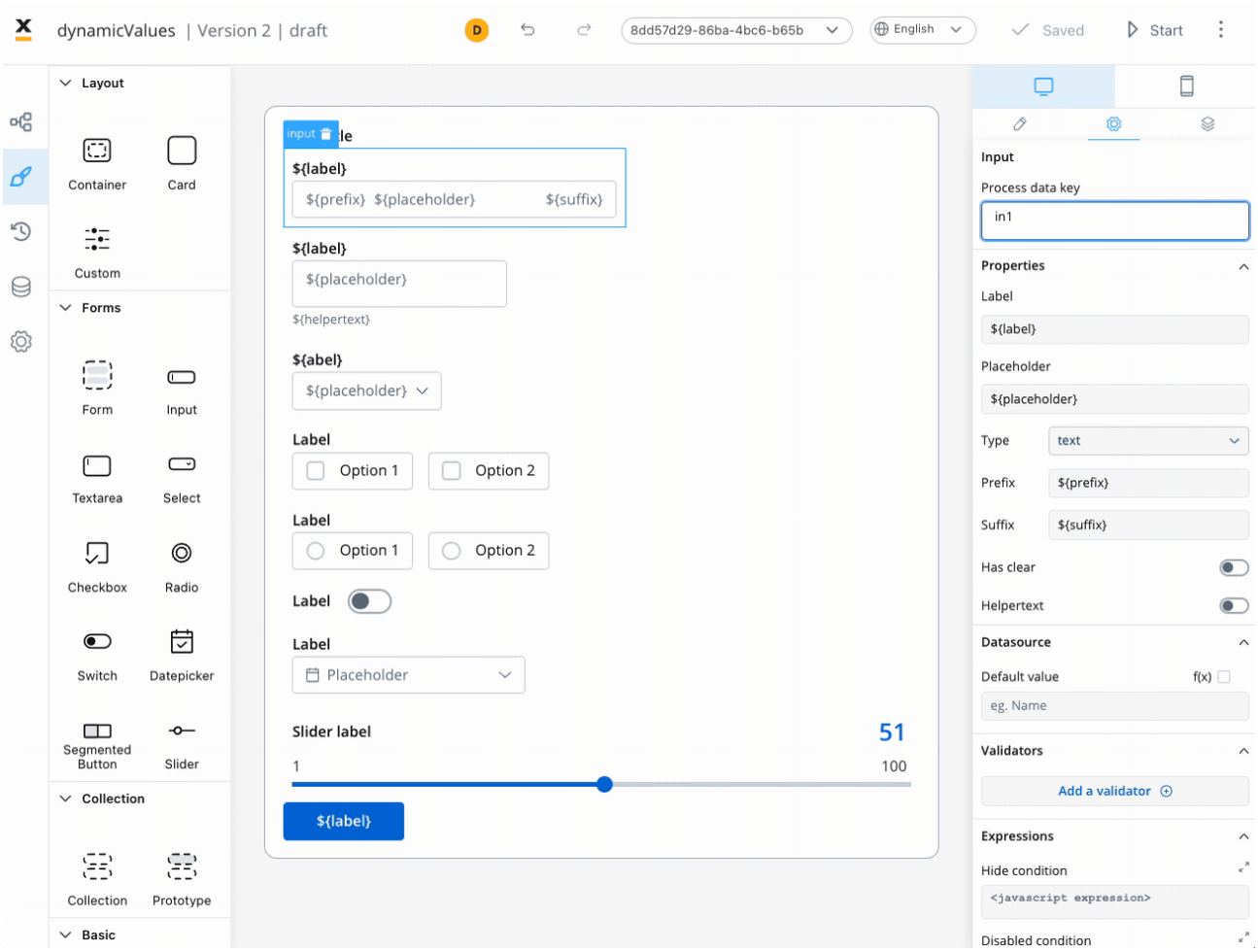
Element	Property	Accepts Params/Subst Tags
Form Elements	Default Value (except switch)	Yes
	Label, Placeholder	Yes
	Helper Text, Validators	Yes
Document Preview	Title, Subtitle	Yes
Card	Title, Subtitle	Yes
Form	Title	Yes
Message	Message	Yes

Element	Property	Accepts Params/Subst Tags
Buttons	Label	Yes
Select, Checkbox, Radio,Segmented Button (Static)	Label, Value	Subst Tags Only
Text	Text	Yes
Link	Link Text	Yes
Modal	Modal Dismiss Alert, Title, Message, Confirm Label, Cancel Label	Yes
Step	Label	Yes

Example using Substitution tags



Example using process parameters



Business rule example

In the above example, the following MVEL business rule was used to populate the keys with values from the task:

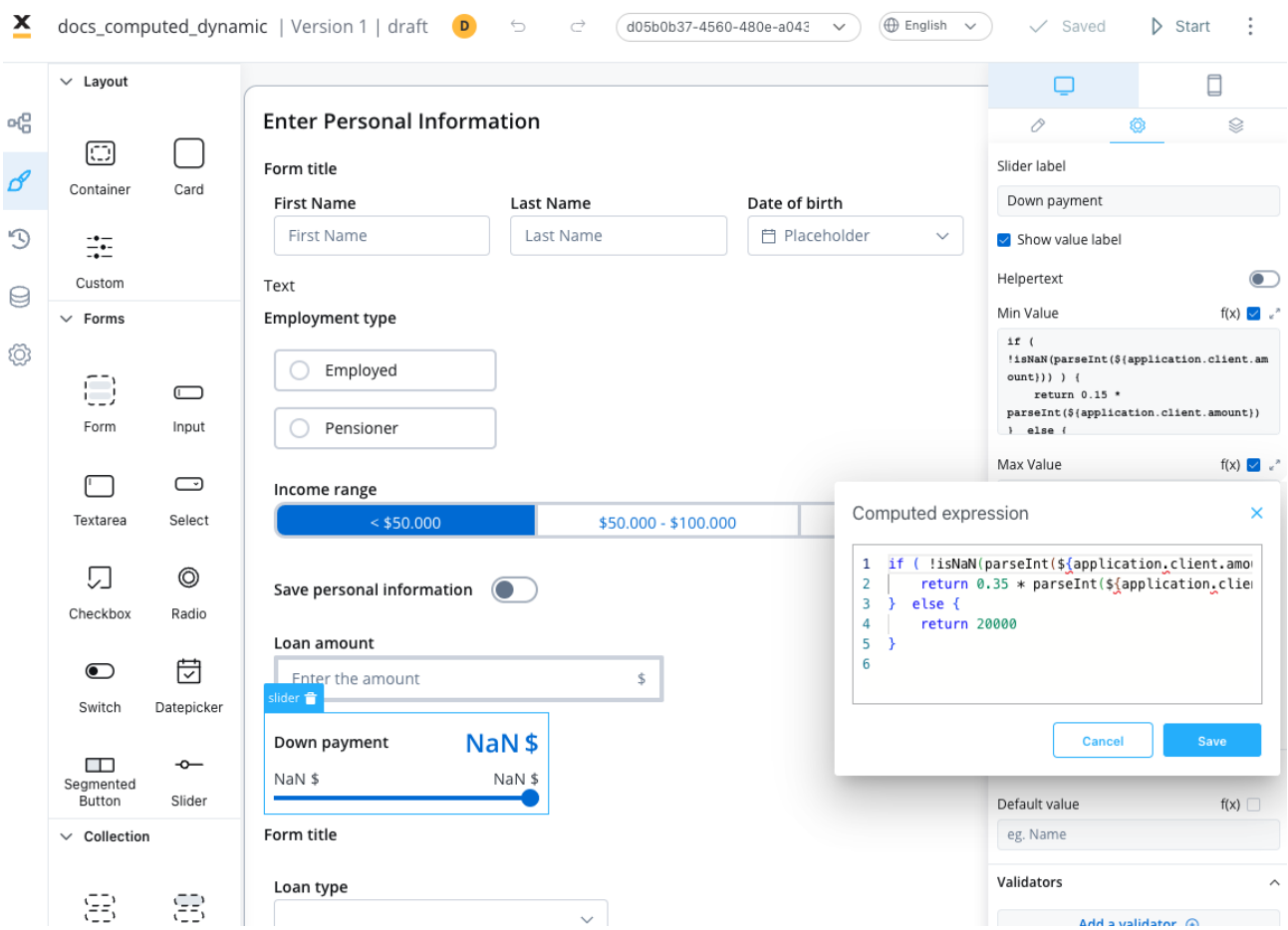
```
///assigning a JSON object containing dynamic values for the specified keys to the "app" key

output.put("app",{"label":"This is a label",
                  "title":"This is a title",
                  "placeholder":"This is a placeholder",
                  "helpertext":"This is a helper text",
```

```
                    "errorM":"This is a error message",
                    "prefix":"prx",
                    "suffix":"sfx",
                    "subtitile":"This is a subtitle",
                    "message":"This is a message",
                    "defaultV":"defaultValue",
                    "value":"Value101",
                    "value":"Value101",
                    "confirmLabel":"This is a confirm
label",
                    "cancelLabel":"This is a cancel label",
                    "defaultValue":"dfs",
                    "defaultDate":"02.02.2025",
                    "defaultSlider": 90});
```

> **CAUTION**
>
> Please note that for releases **<= 3.3.0**, it is not possible to concatenate process parameters with substitution tags when using dynamic values.

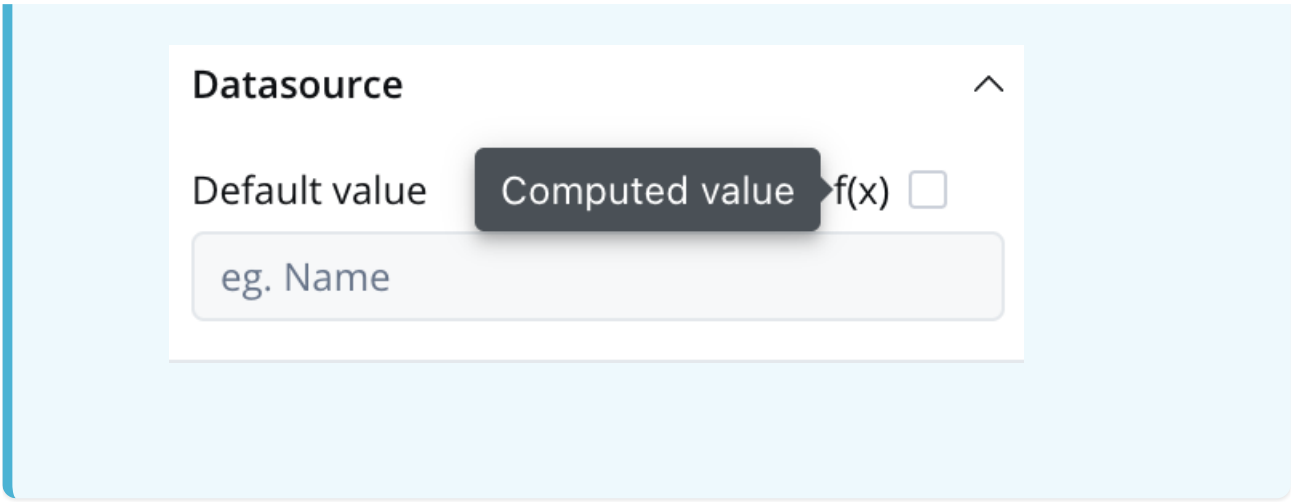
Computed values

Computed values take the concept of dynamic values a step further by allowing you to generate values dynamically using JavaScript expressions. Rather than relying solely on predefined values, computed values enable the calculation, transformation, and manipulation of data based on specific rules or conditions.

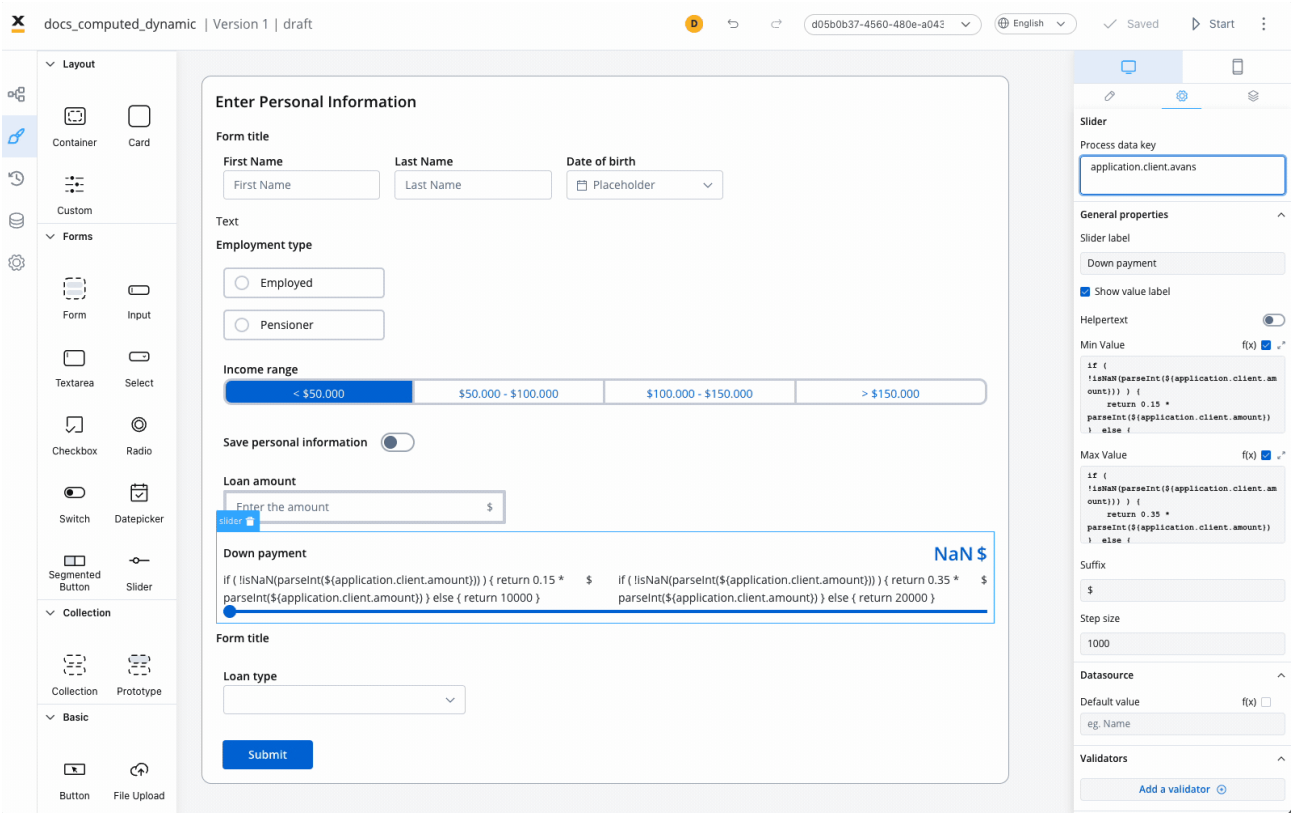


Computed values can be created by writing JavaScript expressions that operate on process parameters or other variables within the application.

> **INFO**
>
> To add a computed value, you have to explicitly check “Computed value” option (represented by the **f(x)** icon), which will transform the desired field into a JavaScript editor.



By enabling computed values, the application provides flexibility and the ability to create dynamic and responsive user interfaces.



Slider example

The above example demonstrates the usage of computed values for a Slider element, where JavaScript expressions are used to compute the minimum and maximum values based on a value entered in an input UI element (linked by the process key `${application.client.amount}`).

Min Value

```
if ( !isNaN(parseInt(${application.client.amount})) ) {
    return 0.15 * parseInt(${application.client.amount})
}  else {
    return 10000
}
```

Max Value

```
if ( !isNaN(parseInt(${application.client.amount})) ) {
    return 0.35 * parseInt(${application.client.amount})
}  else {
    return 20000
}
```

Example details

The code snippets check whether the value of `${application.client.amount}` key can be successfully parsed as an integer. Here's a step-by-step explanation:

- The `parseInt()` function is used to attempt to convert `${application.client.amount}` into an integer.

- The `isNaN()` function is then used to check if the result of the conversion is `NaN` (not a number).
- If the value is not `NaN`, it means `${application.client.amount}` is a valid numeric value.
- In that case, the code calculates the computed value by multiplying the parsed integer by `0.15` (the minimum percentage value for the down payment or with 0.35, the maximum percentage value of the down payment). The result is returned as the computed value for the expression.
- If the value is `NaN` (or `${application.client.amount}` couldn't be successfully parsed as an integer), the code executes the else block and returns a default value of `10000`.

In summary, the JS expressions demonstrates how a computed value can be derived based on a conditional calculation. It first checks if a specific process parameter `(${application.client.amount})` is a valid numeric value, and if so, it computes the value by multiplying it by 0.15 or 0.35. Otherwise, it falls back to a default value of `10000` or `20000`.

Usage

The UI Designer now allows JavaScript expressions to create computed values used on the following UI elements with their properties:

Element	Properties
Slider	min Value, max Value, default Value
Input	Default Value

Element	Properties
Any UI Element that accepts validators	min, max, minLength, maxLength
Text	Text
Link	Link Text

- **Slider**: The min value, max value, and default value for sliders can be set using JavaScript expressions applied to process parameters. This allows for dynamic configuration based on numeric values.
- **Any UI Element that accepts validators min, max, minLength, maxLength**: The "params" field for these elements can also accept JavaScript expressions applied to process parameters. This enables flexibility in setting validator parameters dynamically.
- **Default Value**: For input elements like text inputs or number inputs, the default value can be a variable from the process or a computed value determined by JavaScript expressions.
- **Text**: The content of a text element can be set using JavaScript expressions, allowing for dynamic text generation or displaying process-related information.
- **Link**: The link text can also accept JavaScript expressions, enabling dynamic generation of the link text based on process parameters or other conditions.

Please note that these settings are specifically applicable to numeric values and are not intended for date or string values.

CAUTION

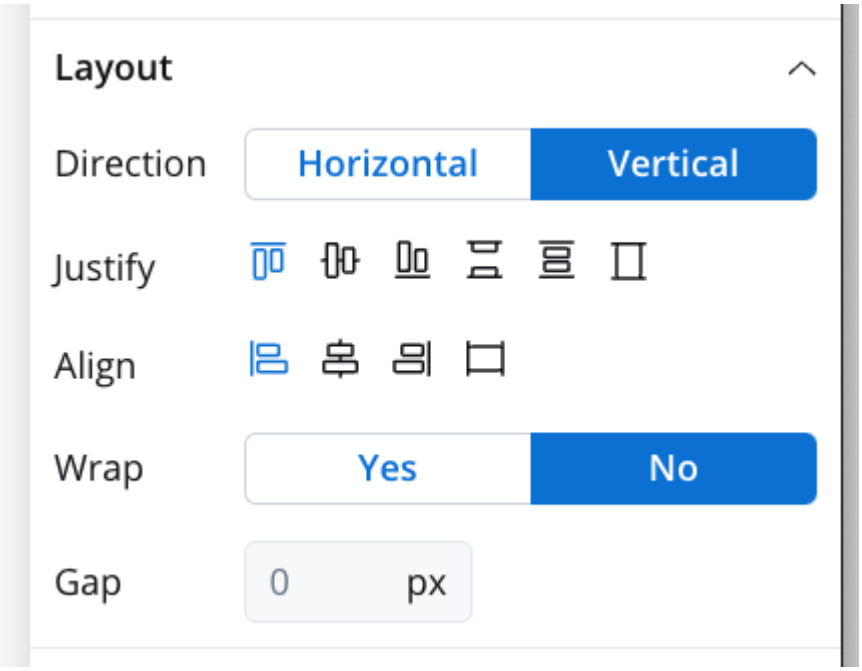
> For input elements (e.g., text input), you may require a default value from a process variable, while a number input may need a computed value.

Was this page helpful?

BUILDING BLOCKS / UI Designer / Layout configuration

Layout settings will be available for all components that can group other types of elements (for example, Container or Card).

These settings allow users to customize properties as the layout direction, alignment, gap, sizing, and spacing.



These settings can be applied practically in various ways, depending on the context and purpose of the design. For example:

- The layout direction and alignment settings can be used to ensure that the content is displayed in a logical and visually appealing way. For instance, a left-to-right layout direction may be more appropriate for languages that read from left to right, while a center alignment may be better for headings or titles.
- The gap, sizing, and spacing settings can be used to control the distance between elements in a design. This can help create a sense of hierarchy and balance, as well as improve readability and usability. For example, increasing the spacing between paragraphs or sections can make the content easier to scan and read, while reducing the size of certain elements can help prioritize others.
- Customizing these properties can also help ensure that a design is accessible and inclusive. For instance, adjusting the layout direction and alignment settings can make the design more readable for users with certain disabilities or who use assistive technologies. Similarly, increasing the spacing and sizing of interactive elements can make them easier to click or tap for users with motor impairments.

To better understand how these layout configurations work and see real-time feedback on different combinations, please refer to the following link:

» Layout configuration

Was this page helpful?

BUILDING BLOCKS / UI Designer / Rendering and UI Designer changelog

> **CAUTION**
>
> This changelog is relevant to the 3.0 release, which introduces significant changes to the UI configuration.

Please be aware that a process (when it comes to UI configuration) may look different from previous releases and that certain updates may not be compatible with older configurations. The migration process may take longer than usual.

Notes for post-migration

After migrating to the 3.0 release, please take in consideration the following changes to ensure smooth operation:

1. Verify font sizes where float values were set.
2. Verify line heights where scale values were set.
3. Verify border radius values where values other than px were set.
4. Review and set padding and margin values where needed. Deleted keys include "margin" : "8px 0" and "padding" : "16px 0 0 16px".
5. Check radio and checkbox elements and update the new `label` prop that has been added.
6. Configure the new `column` prop for Layout (available for checkbox and radio), which allows for grouping many enumerations.
7. The height prop (from Container, Form and Card) has been removed.

8. Update the width prop by configuring the new `Fit W` prop with values such as fill, fixed, or auto. `Min H` and `Max H` props have been removed.
9. The hint and message UI components were combined into a single component called 'message' with the following properties: `type`, `fill`, `border`, and `text`.
10. The button element no longer has the `fill`, `border`, and `flat` types. It now has 4 types: `primary`, `secondary`, `ghost`, and `text`.
11. Added Helpertext (to replace Info tooltip) - this new element can be found on Form elements and provides additional information about each element, which can be hidden within an infopoint.

Was this page helpful?