opentext

docsys (en) 18 August 2023

OpenText™ Identity Manager CE 24.4 (v4.10) Administrator's Guide to Form Builder

January 2025

About Form Builder

The Form Builder is a new interface that enables you to quickly design the forms. The interface displays the required form components, form component organization, and form component control type. The Form Builder provides a simple and easy-to-use graphical user interface. It is a platform for identity management solution developers and administrators to build their own complex forms for automating provisioning for business process applications. You can quickly drag and drop the form components onto the form component area to design new forms. The form components placed in the form component area determine the appearance of the form. The forms are stored in the JSON format. You can directly edit the forms in the JSON editor. Form Builder combines JavaScript forms with REST API Data Management platform for form-based progressive web applications.

OpenText Identity Manager supports form creation through both the legacy method and the Form Builder. We recommend that you to use the Form Builder.

The Form Builder provides the following advantages:

- Allows the forms to be used by multiple provisioning request definition (PRDs).
- Provides drag and drop features to enable you to quickly create modern and responsive forms.
- Simplifies the connections between forms and REST APIs.
- Provides multiple components (widgets).
- Supports all widgets available in the legacy forms.

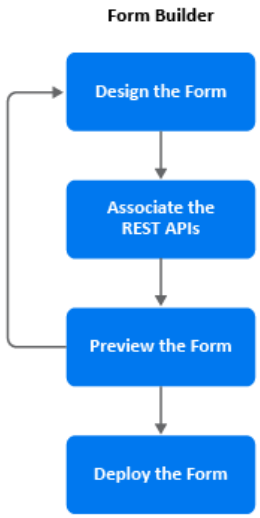
Currently, Designer does not support migration of legacy forms to JSON forms. You must migrate the forms manually. To migrate a legacy form to a JSON form, perform the following actions:

1. Back up the workflows that use the legacy form.
2. Create a JSON form in the Form Builder.
3. Add the existing data item mappings to the newly created JSON form.
4. Deploy the JSON form to the Identity Vault by using Designer.

How does the Form Builder Work?

The Form Builder uses the JSON schema to render the form dynamically within Identity Applications. The schema automatically generates the corresponding APIs to receive the data when the form is submitted. You can preview the saved forms before deploying them and make the necessary changes. OpenText Identity Manager stores the forms in the JSON format on your file system.

Figure 1 *Form Creation Flow*



Launching Form Builder

Perform the following actions in Designer to launch the Form Builder.

1. From the Provisioning view, right-click the Provisioning Request Definitions node and select **New**.
2. Specify the **Identifier (CN)**, **Display Name**, and **Description**.
3. Click **Next**.
4. Select **Create a provisioning request definition using one of the templates**, choose the desired template from the **Available Templates** list, and click **Next**, then click **Finish**.
5. In the page that opens, ensure that **JSON Forms Selection** is selected. Save the workflow.

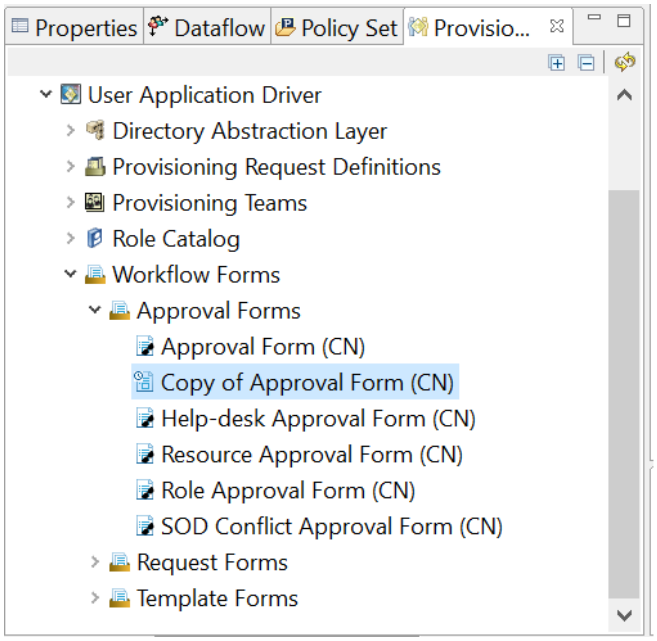
   This will display the **JSON Forms** tab.
6. Click the **JSON Forms** tab.

   The Form Builder is launched.

Creating a Form

You can create a new form (approval, request, or template) for the Provisioning Request Definition from the Workflow Forms container located in the User Application driver or directly in the Form Builder.

Figure 2 *Creating a form*



1 Navigate to the Workflow Forms in the Outline view of Designer and right-click one of the following options to select the type of form you wish to create:

- Approval Forms
- Request Forms
- Template Forms

Alternatively, click **+ New Form** in the Form Builder page and select the type of form you wish to create. For example: Approval Form or Request Form

2 In the **New Workflow Forms** window, specify the **Form Identifier** that uniquely identities the form and click **Finish**.

The Form Builder is launched.

3 Click +.

4 In the **New Form** dialog box, select a template and click **Create**.

5 Drag and drop the required components to design the form. For more information, see Form Components.

NOTE: Few fields are pre-populated based on your selection of the form type. You can edit or delete these fields as required.

---

6 Create the form and save it.

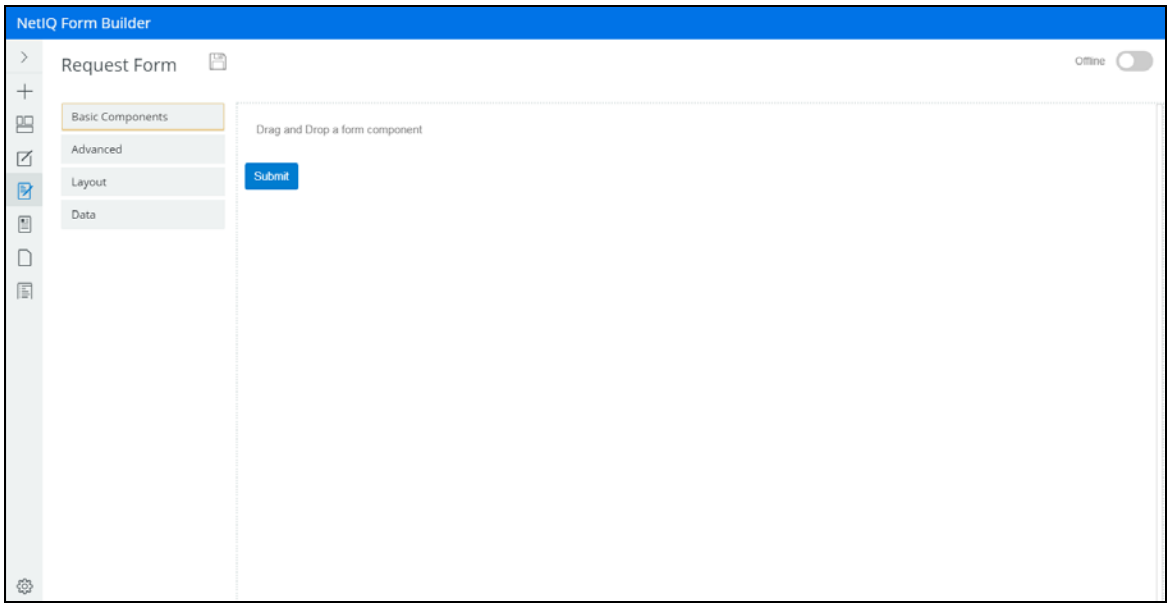
The form is saved in the same location in the Workflow Forms container under the type of the form that you chose to create. For example, an approval form is saved under Approval Forms container.

After saving the form, deploy the form to the Identity Vault through Designer. For more information, seeAbout Forms in the NetIQ Identity Manager - Administrator's Guide to Designing the Identity Applications.

Exploring the Form Builder User Interface

The Form Builder enables you to quickly create forms and resources by using a simple drag-and-drop user interface. You can create, modify, copy, and delete a form in the Form Builder.

Figure 3 *Form Builder*



To design a form, drag and drop the required form components and provide the details. For more information about each component and the corresponding input fields, see "Form Components" on page 6. Once the form is submitted, it is sent to the client in the JSON format. At the client side, the Form Renderer renders the form.

NOTE: The field events other than `OnLoad` and `OnChange` are not directly supported from Form Builder.

The Form Builder page consists of the following components:

- Toolbar
- Form Components

Toolbar

The toolbar consists of the following elements:

Table 1 *Toolbar with description*

Button	Description
Form Templates	Form Templates has replaced the New Form option in version 4.8.1. Starting from this release, you can use a template to create new forms. Click to create a new form from existing templates (Request or Approval form). For more information, see "Creating and Editing Request or Approval Forms" on page 21.
Form Builder	Click to expand the menu layout.
JS Editor	Click to edit the form in the JS editor. The editor enables you to write your logic to change the custom default value and calculate the value using JSON arrays. For more information, see "Editing a form in JS Editor" on page 24.
Form JSON	Click to edit the form in the JSON editor. The editor provides a JSON representation describing a fully-featured form. You can also use the editor to duplicate and edit an existing form. For more information, see "Editing a Form in the JSON Editor" on page 23.
Preview	Click to preview the designed form.
Localization	Select the language you want the fields in the form to be rendered in form renderer (Identity Applications).
External Scripts	If you wish to add a JavaScript to the form from an external source, specify the URL of the external script here.
Save	Click to save the form as a template. Forms are saved as XML documents in the project directory. Templates are available only within the project in which you create them.

Button	Description
Settings	Click to perform the following actions: ◆ Change Language: Allows you to select the supported language in Form Builder. ◆ Preview Settings: Enabling this option enables the buttons in the preview mode. This allows you to approve or deny the form. It is recommended to perform this action (approve or deny) only through Identity Applications. ◆ About: Displays the version number of the build. ◆ Debug Mode: Displays the code and helps to troubleshoot errors.
Offline - Online toggle Offline	On enabling this option, the preview of the form functions is similar to the Form Renderer. For more information see "Offline - Online Toggle" on page 37.

Form Components

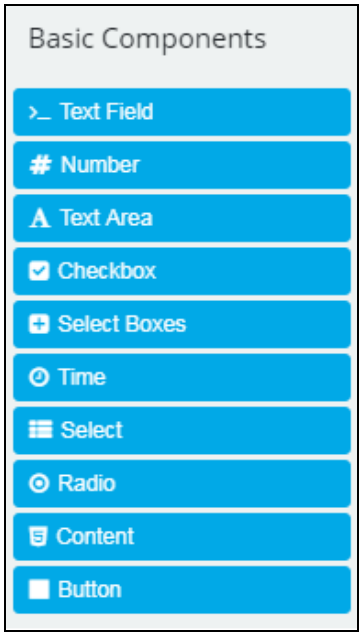
Form components collects data and serves as the display or user interface within the system. It helps you to define the type of widget that is required to enter data and automatically adds a property to the resource endpoint to interact with the Form component.

The Form Builder page consists of the following elements:

Basic Components

The following elements are displayed when you select **Basic Components**.

Figure 4 *Basic Components*

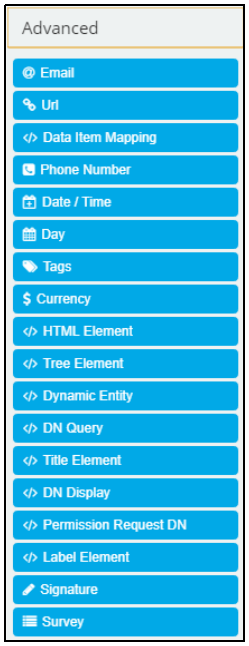


- **Text Field** - Used for adding short and general text input. For more information, see Textfield (https://www.netiq.com/documentation/identity-manager-48/FormioHelp/userguide/form-components/#textfield).
- **Number** - Used when the field is limited to a number type. For more information, see Number.
- **Text Area** - Has the same options as the Text Field element. The only difference is that it provides a multi-line input field to be used in case of longer text. For more information, see Text Area.
- **Checkbox**- Used for a boolean value input. For more information, see Checkbox.
- **Select Boxes** - Works similar to the Radio component. However, it allows you to check multiple values. For more information, see Select Boxes.
- **Time** - Used to enter the time format.
- **Select** - Displays the values as a drop-down list. For more information, see Select.
- **Radio** - Used when you need to choose from a list of options. For more information, see Radio.
- **Content** - Used to provide non-field information. For more information, see Content.
- **Button** - Used to perform various actions within the form. For more information, see Button.

Advanced

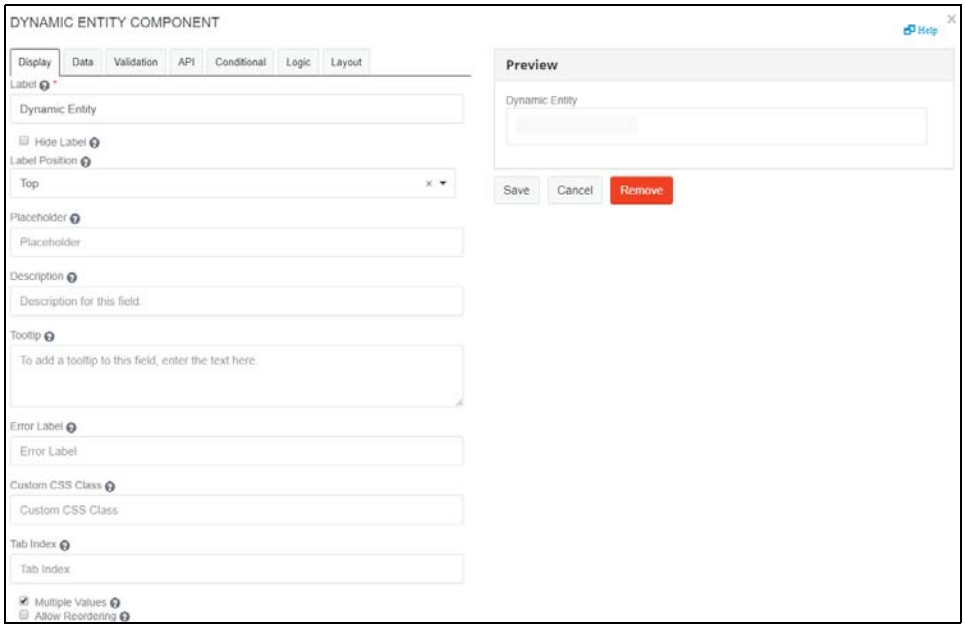
The following elements are displayed when you select **Advanced**.

Figure 5 *Advanced*



- **Email** - Used to add an email address field for your form. It has a custom validation setting that validates the entered email address. For more information, see Email.
- **Url** - Used to add a URL in the form.
- **Data Item Mapping** - Used to map data from the data flow into fields in a form (pre-activity mapping) and to map data from the form back to the data flow (post-activity mapping). For more information, see Defining the Data Item Mappings in the *NetIQ Identity Manager - Administrator's Guide to Designing the Identity Applications*.
- **Phone Number** - Used to add a phone number field in the form. For more information, see Phone Number.
- **Date/Time** - Used to input dates, time, or both. For more information, see Date/Time.
- **Day** - Used to enter the values for the month, day, and year. The day component, by default, takes the value in month/day/year format. However, the order of the fields can be changed. Select the **Day First** check box in the **Display** tab if you want the day field before the month field.
- **Tags** - Used to add custom tags.
- **Currency** - Used when a field should display currency amounts on a form. For more information, see Currency.
- **HTML Element** - Used to display a single HTML element in a form. For more information, see HTML Element.
- **Signature** - Allows you to sign the field with either their finger on a touch enables device or with the mouse pointer. For more information, see Signature.
- **Survey** - Used to ask multiple questions with the same context of answers or values. For more information, see Survey.
- **Dynamic Entity** - This is a multi select drop-down field. This widget allows you to select more than one entity in an entity type. For example, user, group.

Figure 6 *Dynamic Entity*



The following fields are populated in the **Data** tab:

Entity Key: Specify the entity key for the entity type. For example, for the entity type User, the entity key is user.

Display Expression Attribute: Specify the attributes of the entity type you want to be displayed. You can add multiple display attributes. For example, for the entity type User, the **Display Expression Attribute** can be FirstName, LastName.

Entity Type	Display Expression Attribute
User	FirstName, LastName

Service ID: Select the service ID. For example, IDM or IG.

Limit: Specify the number of entities that you want to be displayed. The default value is 20.

Default Value: Specify the value to be displayed in the field before user interaction. Having a default value overrides the placeholder text.

Refresh On: Refreshes data when another field changes.

NOTE: When the Form Builder is Online, a couple of additional fields are displayed in the **Data** tab. For more information, see "Offline - Online Toggle" on page 37.

For more information about the fields populated in the **Display** tab, see "Display" on page 17.

For more information about Custom Default Value and Calculated Value, see "Custom Default Value" on page 19 and "Calculated Value" on page 19.

For more information about the fields populated in the other tabs, see "Validation" on page 19, "API" on page 20, "Conditional" on page 20, and "Logic" on page 21.

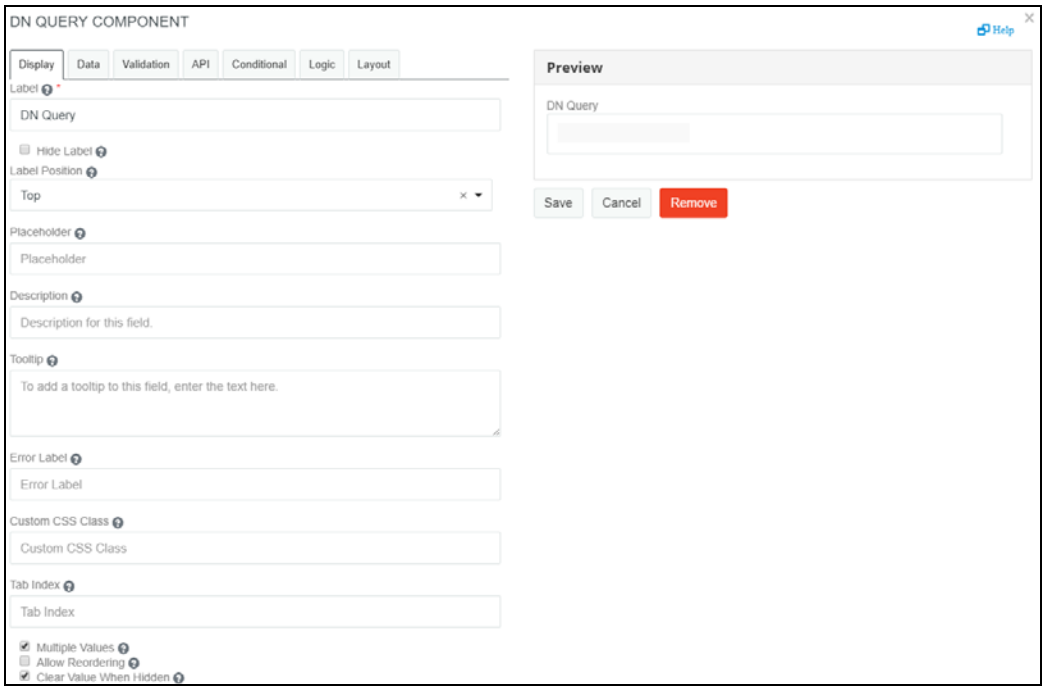
For more information about each tab for the selected component, see "General Settings for the Selected Component" on page 17.

- **DN Query** - Allows you to search and retrieve DNs from the Identity Vault. However, with the DNQuery, the object selector content can be driven by the result of a directory abstraction layer Queries object rather than from properties.

  This widget is used to define a condition on how you want the results to be displayed.

  You must ensure that the query, parameter, and key you specify is present in Designer.

  Figure 7 *DN Query*



The following fields are populated in the **Data** tab:

Service ID: Select the required service ID. For example, IDM or IGA.

Parameters: Specify the parameter key and its value. You can have multiple keys for a parameter, and you can have multiple values for a key.

NOTE: You must provide the parameter value (static value) while configuring this component. Dynamic parameter value such as `data.<other dynamic value>` is not supported. To use the dynamic parameter value, you can use the `IDVault.globalquery` in the **Select** component.

Query Key: Specify the key of the DAL Queries object.

Return Attributes: Specify the attributes of the entity type you want to be displayed. You can add multiple display attributes.

Default Value: Specify the value to be displayed in the field before user interaction. Having a default value overrides the placeholder text.

Refresh On: Refreshes data when another field changes.

For more information on the fields populated in the **Display** tab, see "Display" on page 17.

NOTE: When the Form Builder is Online, a couple of additional fields are displayed in the **Data** tab. For more information, see "Offline - Online Toggle" on page 37.

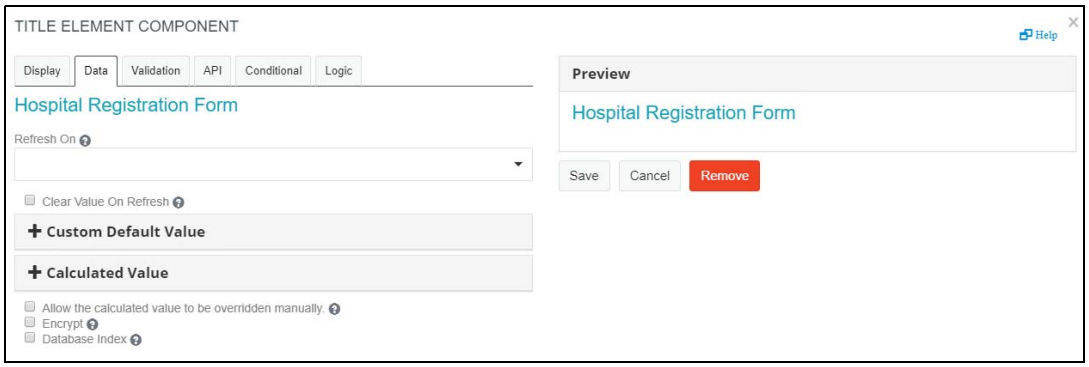
For more information about Custom Default Value and Calculated Value, see "Custom Default Value" on page 19 and "Calculated Value" on page 19.

For more information about the fields populated in the other tabs, see "Validation" on page 19, "API" on page 20, "Conditional" on page 20 and "Logic" on page 21.

For more information about each tab for the selected component, see "General Settings for the Selected Component" on page 17.

- **Title Element** - Used to design a title. You can use the "Custom Default Value" on page 19 option for dynamic value settings.

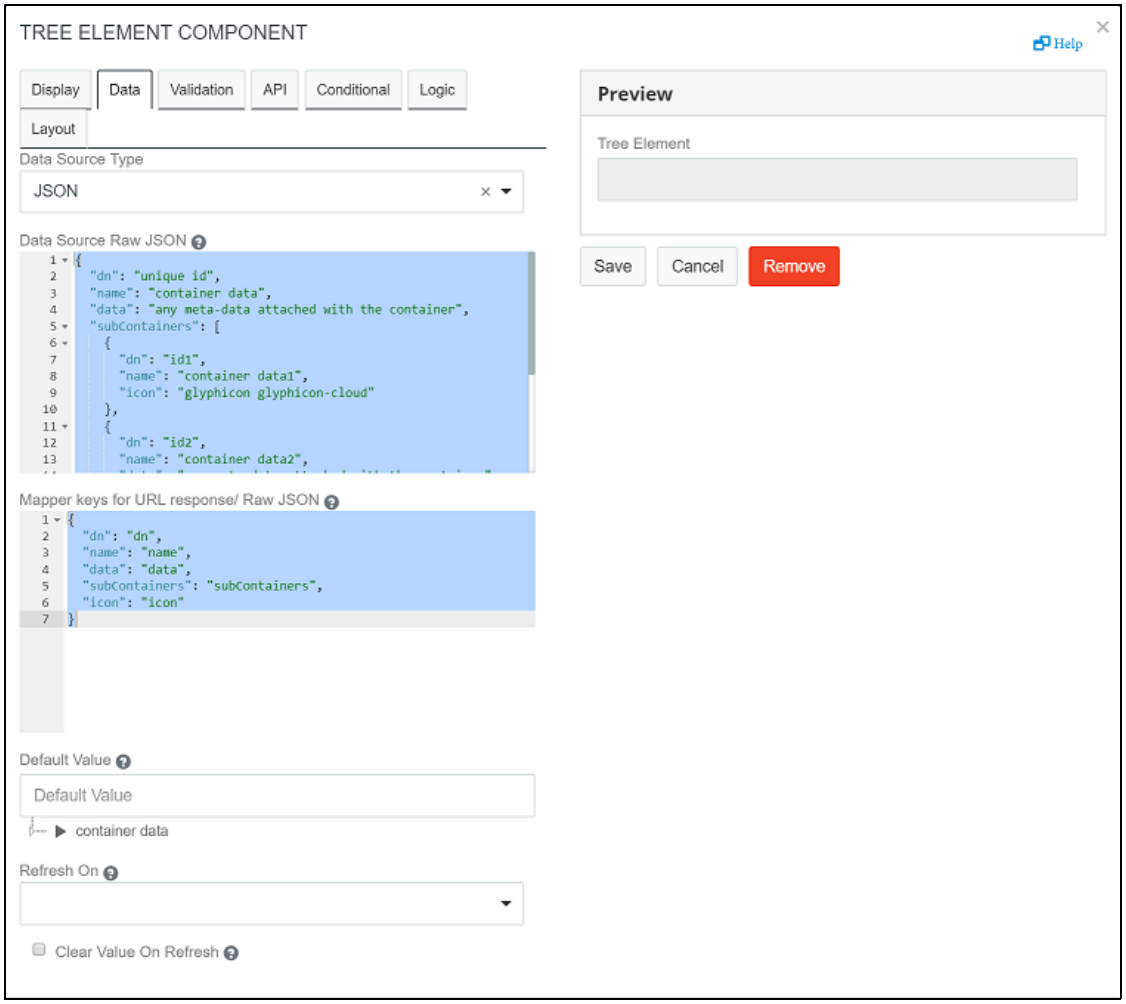
Figure 8 *Title Element - Data*



For more information about the fields populated in the other tabs, see "Display" on page 17, "Data" on page 19, "Validation" on page 19, "API" on page 20, "Conditional" on page 20 and "Logic" on page 21.

- **Tree Element** - Used to design a hierarchy. You are allowed to configure nodes that helps to modify the design of the form in the desired format.

Figure 9 *Tree Element - Data Source JSON*



NOTE: The **Preview** area displays how the form would render on making the changes or edits to any field.

The following fields are populated in the **Data** tab.

Data Source Type: Select the type of data source, that is JSON or URL.

The following fields appear when the **Data Source Type** is **JSON**:

Data Source Raw JSON: Each node configuration is displayed in a JSON file. You must have the following details mentioned:

dn: It is a unique ID that defines the selected node. The value must be a string. This is the data value for the tree element that you add and is passed to the workflow.

name: Enter the name you wish to be displayed on the User Interface. The value must be a string.

data: (Optional) This saves the metadata to the existing field.

subcontainers: It defines the child node.The value must be an array of JSON. Each element of the array is an object following the same structure as that of a node.

icon: The image class to be used for displaying icon on each node. The value must be a string. If you do not define the value, the node uses the image class selected in **Default Icon Class** under the **Display** tab.

NOTE: All nodes must follow either JSON or customized data structure. The parent and child cannot be of different structure.

If JSON does not follow the default structure, it must be mapped accordingly.

Mapper keys for URL response: Allows you to map the JSON parameters with the URL element parameters.

You can use the **Mapper keys for URL response/ Raw JSON** setting to customize the field names provided under **Data Source RAW JSON**.

For example, if you want the field names as ["id","display","description","child","icon"], then modify the details under **Mapper keys for URL response/ Raw JSON** as follows:

```
{
    dn: "id",
    name: "display",
    data: "description",
    subContainers: "child",
    icon: "icon"
}
```

The following fields appear when the **Data Source Type** is **URL**:

HTTP method: Select the required http method. For example, GET, POST.

Service ID: Select the appropriate service ID. For example, IDM, IG.

Data Source URL: The URL that returns a JSON array to use as the data source.

Mapper keys for URL response: Allows you to map the JSON parameters with the URL element parameters.

Request Headers: Set any headers that should be sent along with the request to the URL. This is useful for authentication.You can have multiple keys for a parameter, and you can have multiple values for a key.

Request payload: Enter the request body for the root node in the request payload field. The request body contains the node details that will be used to load the node. It applies in case of POST method.

lazy parameter: Enter the parameter name that will be used to load a sub-node. The lazy parameter such as `nodeid` is appended in the data source URL provided by the user. It applies in case of GET method (Lazy Loading).

Default Value: The entered value is displayed in the field before user interaction. Having a default value overrides the placeholder text.

Refresh On: Refreshes data when another field changes.

Clear Value On Refresh: This text appears below the input field.

For more information about Custom Default Value and Calculated Value, see “Custom Default Value” on page 19 and “Calculated Value” on page 19.

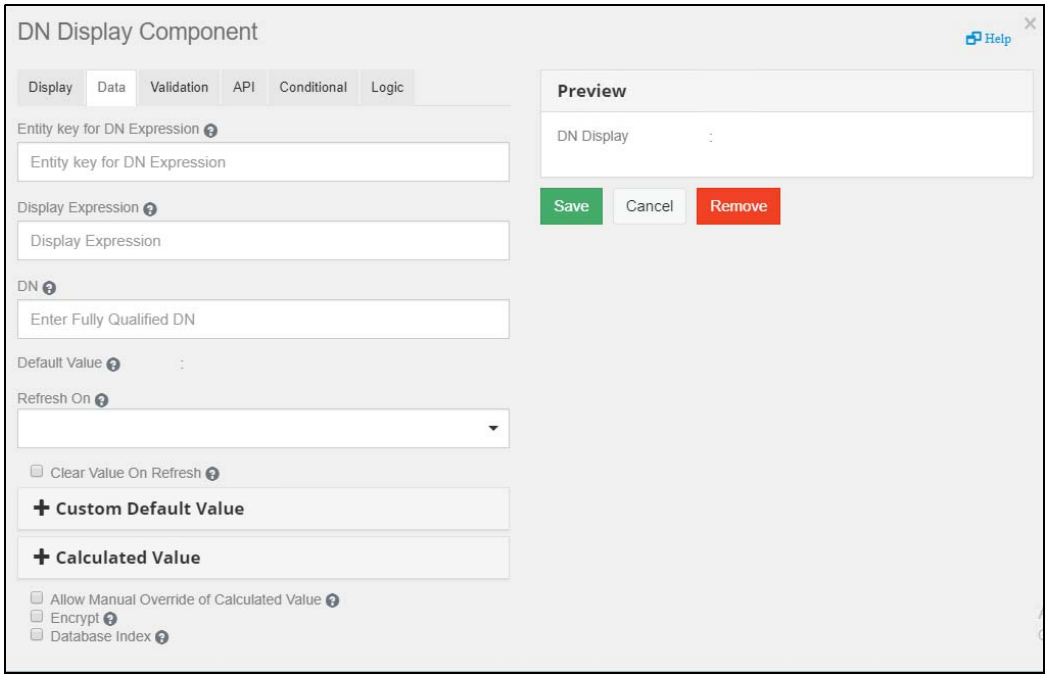
For more information about each tab for the selected component, see “General Settings for the Selected Component” on page 17.

For more information about the fields populated in the other tabs, see “Display” on page 17, “Validation” on page 19, “API” on page 20, “Conditional” on page 20 and “Logic” on page 21.

- **DN Display** - Used to display a read-only DN. It can display the full DN or a set of attributes associated with the DN depending on the properties you set.

This is similar to **Dynamic Entity** mentioned in the "Advanced" on page 7 tab, except that you need to provide a 'DN'. You must enter a valid DN as DN validation is not done.

Figure 10 *DN Display*



The following fields are populated in the **Data** tab:

Entity key for DN Expression: Represents the type of entity used for DN display. Leave this value blank if you want to display the full DN or CN value retrieved from the Identity Vault, else enter an entity.

The entity you choose must:

- Have the directory abstraction layer View property set to **True**.
- Be the entity of the DN you are working with.

Display Expression: Represents the attribute used for DN display. It also displays multiple attributes, provided the attribute is separated by comma. Leave this value blank if you want to display the full DN or CN value. If you want to mask the DN by displaying attributes, you must first specify an **Entity key for DN expression.**

For example, to show the user entity's first and last name attributes, construct an expression like this: `FirstName LastName`.

Ensure the attribute's View, Read, Search, and Required properties are set to True in the directory abstraction layer. For more information, see Attribute Properties in the *NetIQ Identity Manager - Administrator's Guide to Designing the Identity Applications*.

DN: Enter the respective DN.

Default Value: The entered value is displayed in the field before user interaction. Having a default value overrides the placeholder text.

Refresh On: Refreshes data when another field changes.

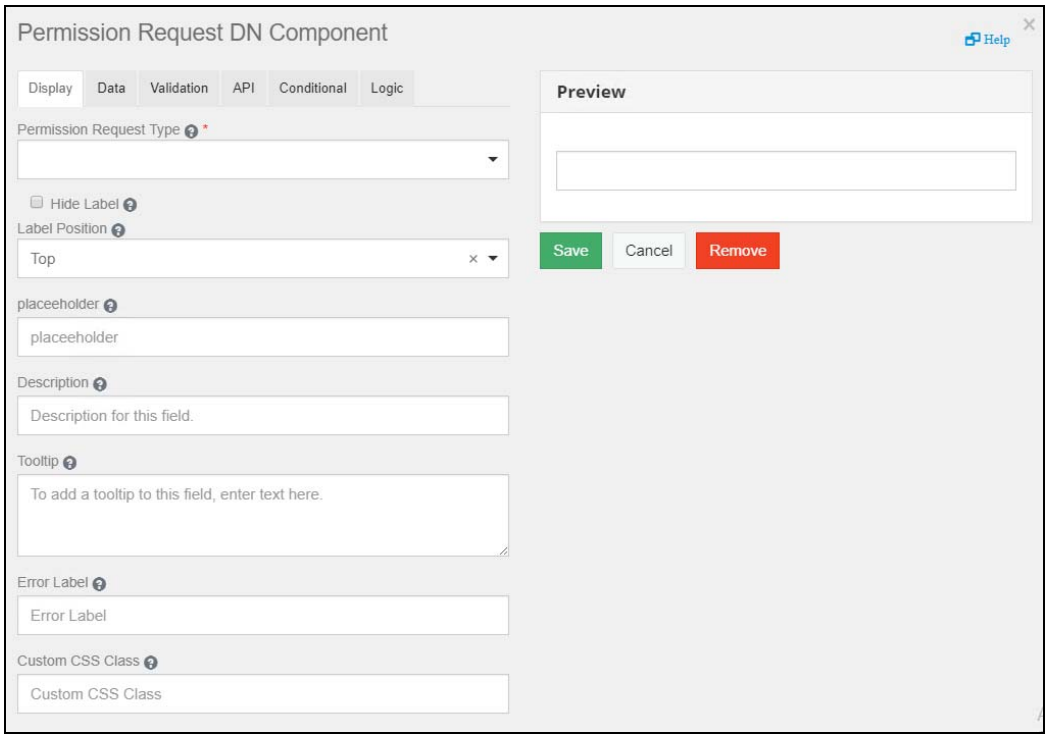
Clear Value On Refresh: When Refresh On field is changed, clear this components value.

For more information on each tabs for the selected component, see "General Settings for the Selected Component" on page 17.

NOTE: When the Form Builder is Online, a couple of additional fields are displayed in the **Data** tab. For more information, see "Offline - Online Toggle" on page 37.

- **Permission Request DN** - Used to provide permission for a default role and resource approval template.

Figure 11 *Permission Request DN*

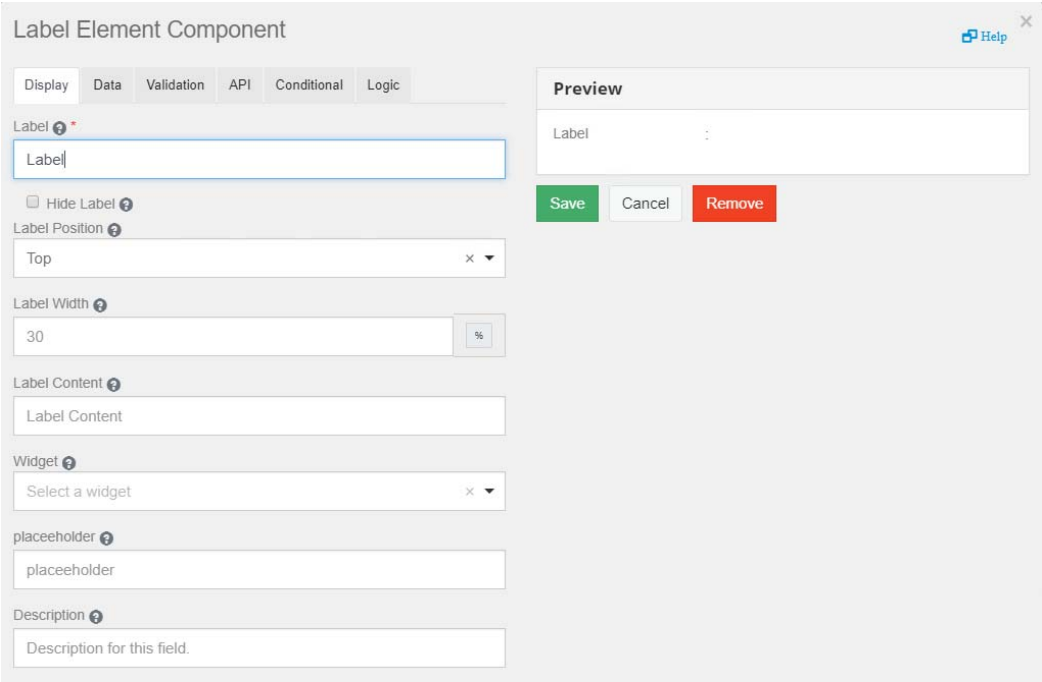


In the **Display** tab, based on the selected **Permission Request type**, the label is automatically populated.

For more information on the fields populated in the **Display, Data, Validation, API, Conditional, Logic** and **Layout** tabs, see Display, Data, Validation, API, Conditional, Logic, and Layout.

- **Label Element** - Allows you to create and design the labels used in the forms.

Figure 12 *Label Element*



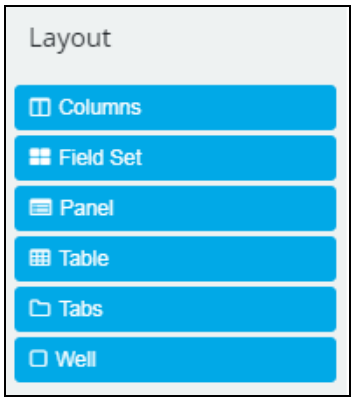
For more information on the fields populated in the **Display, Data, Validation, API, Conditional, Logic** and **Layout** tabs, see Display, Data, Validation, API, Conditional, Logic, and Layout.

Layout

These elements are used to change the general layout of the form.

The following elements are displayed when you select **Layout**.

Figure 13 *Layout*



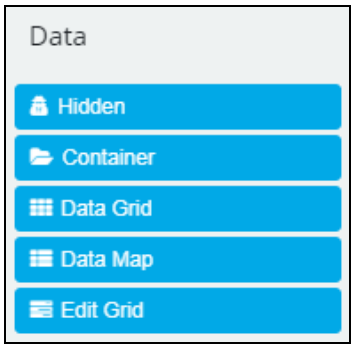
- Columns
- Field Set
- Panel
- Table

- Tabs
- Well

Data

The following elements are displayed when you select **Data**.

Figure 14 *Data*



- Hidden
- Container
- Data Grid
- Data Map
- Edit Grid

General Settings for the Selected Component

When you drag and drop a component on the Form Builder interface, the following general settings are displayed for most of the components.

These settings allow you to change the Title, description of the project, enter required values (can be predefined as well) and so on, depending on the tabs you select.

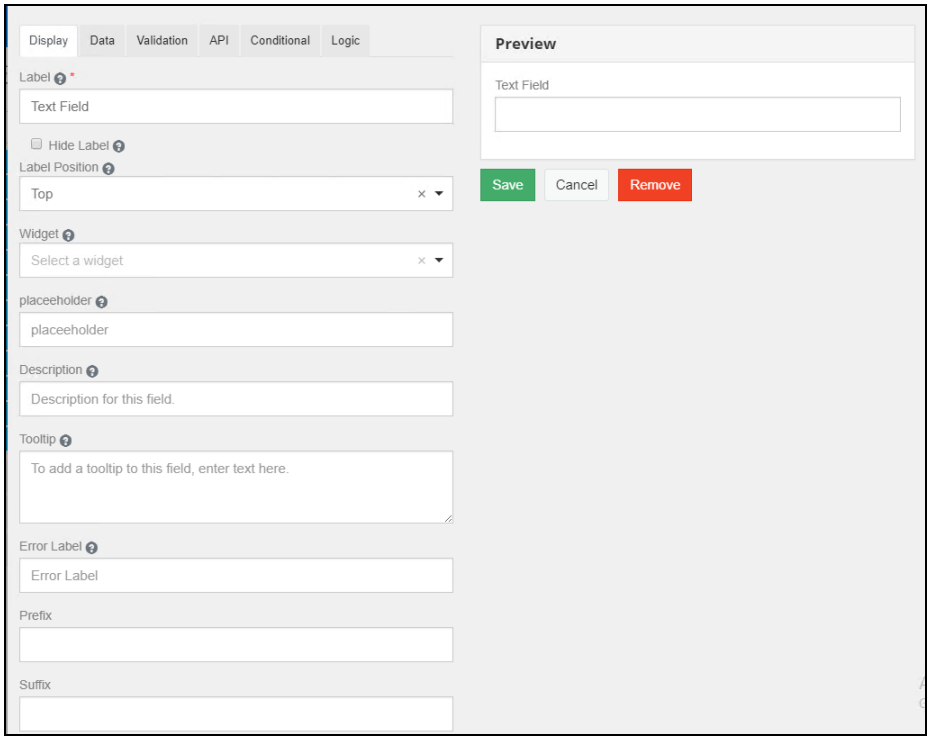
The following sections provide examples of the elements displayed in each tab for a selected component.

Component Details

Display

The following elements are listed in the **Display** tab for the **Text Field** component. This tab consists of elements that defines how the form would appear upon rendering.

Figure 15 *Display tab for Text Field component*



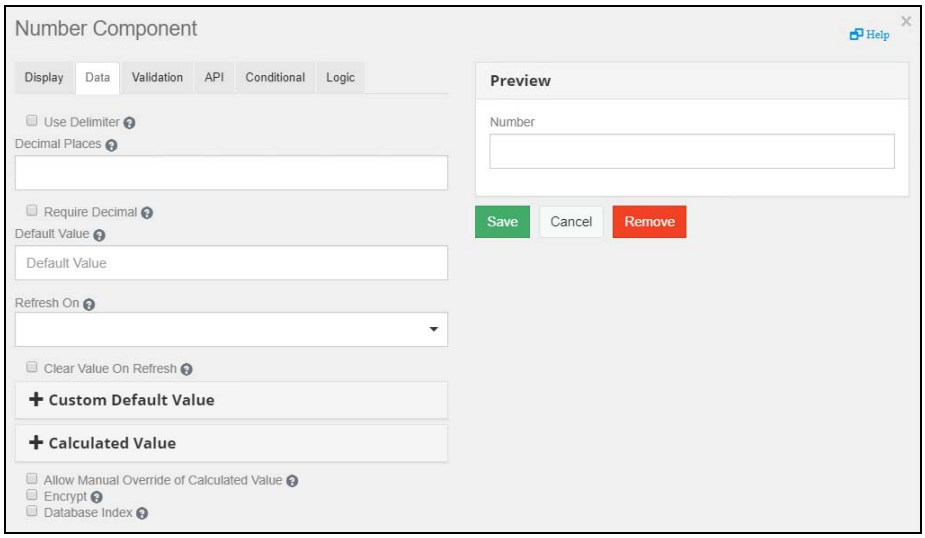
- Label
- Hide Label
- Label Position
- Widget - It is the display user interface, used to input the value of the field.
- Placeholder
- Description
- Tooltip
- Error Label
- Input Mask
- Allow Multiple Masks
- Prefix
- Suffix
- Custom CSS Class
- Tab Index
- Multiple Values
- Protected
- Hidden
- Hide Input
- Disabled
- Initial Focus

- Table View
- Always Enabled - The field will always be enabled for editing.

Data

The following elements are listed in the **Data** tab for the **Number** component. This tab consists of elements that define the default values and format of the component to be displayed on the form.

Figure 16 *Data tab for Number component*



- Use Delimiter
- Decimal Places
- Require Decimal
- Default Value
- Refresh On

Custom Default Value

The variables listed here can be used to customize and write the JavaScript or JSONLogic.

If you have specified the **Data Item Mapping** Value in Designer, the **Custom Default Value** will overwrite the **Data Mapping Value**. In other words, the **Custom Default Value** takes precedence.

Calculated Value

The variables listed here allows you to calculate the values based on the values in other fields of the form. For more information, see Calculated Values.

Validation

The following elements are listed in the **Validation** tab for majority of the selected components. This tab consists of elements that define the required fields that needs to be displayed on the form.

- Required
- Custom Validation

API

The following elements are listed in the **API** tab for the **Tree Element** component. This tab consists of elements that define the property name and configure any custom properties for the selected component.

NOTE: Every component must have a unique **Property Name**.

Figure 17 *API tab for Tree Element component*

- Property Name
- Field Tags
- Custom Properties

Conditional

The following figure displays the **Conditional** tab for the **Text Area** component. This tab consists of elements that determine when the fields should be hidden or displayed on the form. For more information, see Conditional Components.

Figure 18 *Conditional tab for Text Area component*

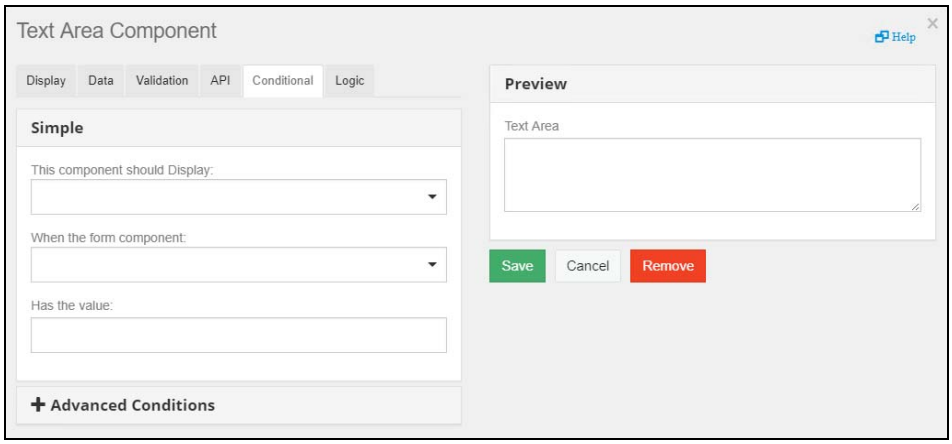
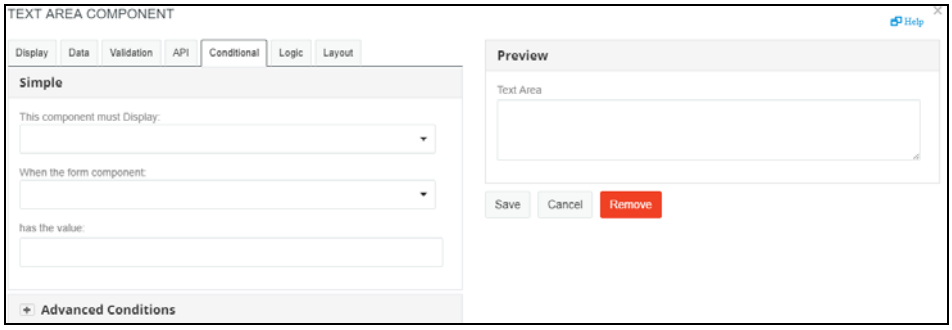


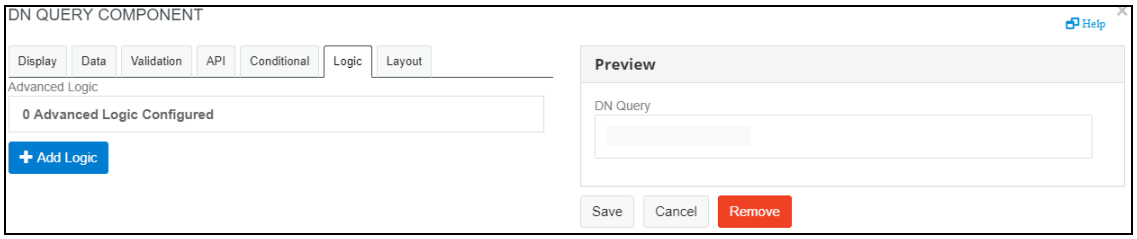
Figure 19 *Conditional tab for Text Area component*



Logic

The **Logic** tab allows you to define and configure multiple logic and action for the selected component. This helps you to design a form which can perform certain defined actions for the defined logic. The following figure displays the Logic tab for the **DN Query** component.

Figure 20 *Logic tab for DN Query component*



Layout

Applies to Form Builder 4.8.1 and onwards.

The **Layout** tab allows you to define the HTML attributes and map those attributes with the component's input element. You cannot edit the component type from the values you specify in this tab. For example, if you select a **Text Field** component, you cannot change the component type to a different value such as a **Checkbox**.

Figure 21 *Layout tab for Data Item Mapping component*

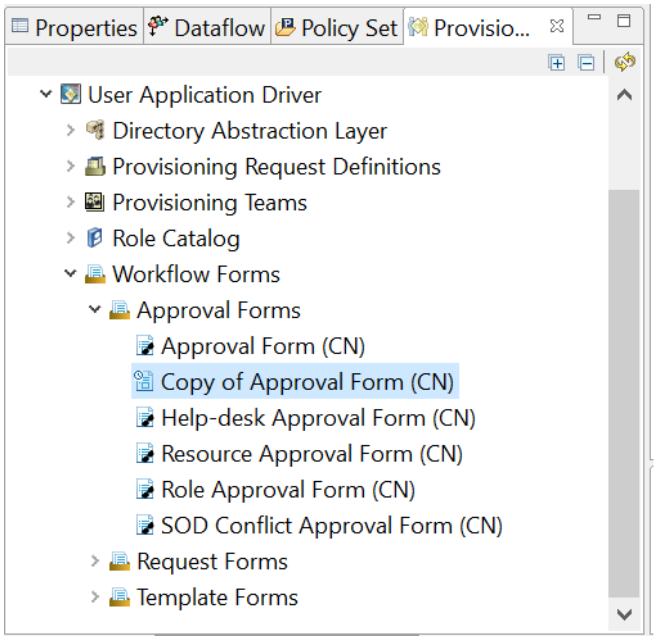
Creating and Editing Request or Approval Forms

- "Creating a Form" on page 22
- "Editing a Form" on page 23
- "Migrating Legacy Forms to New Forms" on page 29

Creating a Form

You can create a new form (approval, request, or template) for the Provisioning Request Definition from the Workflow Forms container located in the User Application driver or directly in the Form Builder.

Figure 22 *Creating a Form*



1 Navigate to the **Workflow Forms** in the Outline view of designer and right-click one of the following options to select the type of form you wish to create and select **New**:

- Approval Forms
- Request Forms
- Template Forms

2 In the **New Workflow Forms** window, specify the **Form Identifier** that uniquely identities the form and click **Finish**.

The Form Builder is launched.

Alternatively, click **+ Form Templates** in the Form Builder page, select the type of form you wish to create (for example: Approval Form or Request Form) and click **Create**.

NOTE:

- Form Templates has replaced the New Form option in version 4.8.1. Starting from this release, you can use a template to create new forms.
- If you try to design a new form when an existing form is open in the Form Builder, a pop-up message is displayed. Click **OK** to overwrite the existing form. To return to the old form, click **Cancel**.

---

3 Drag and drop the required components to design the form. For more information, see Form Components.

NOTE: Few fields are pre-populated based on your selection of the form type. You can edit or delete these fields as required.

---

4 Create the form and save it.

The form is saved in the same location in the **Workflow Forms** container under the type of the form that you chose to create. For example, an approval form is saved under **Approval Forms** container.

After saving the form, deploy the form using Designer. For more information, see About Forms in the *NetIQ Identity Manager - Administrator's Guide to Designing the Identity Applications*.

Editing a Form

You can edit a form through the following ways:

- "Editing a Form Using the Form Builder User Interface" on page 23
- "Editing a Form in the JSON Editor" on page 23
- "Editing a form in JS Editor" on page 24

Editing a Form Using the Form Builder User Interface

Perform the following actions to edit a form component in a form:

1 Click the icon next to the component that you want to edit.

2 Make the necessary edits in the **Settings** form.

NOTE: You can edit, copy, paste and remove the fields, on each field on the form.

3 Click **Save**.

Editing a Form in the JSON Editor

All forms rendered within the Form Builder platform uses JSON Schema. When you add new components onto a form, you are essentially defining a JSON schema in the background. The Form Builder uses this schema to invoke the REST APIs needed to support the form. This section provides a detailed explanation of the structure of the JSON schema and the components that can be rendered within a form.

TIP: Do not directly edit the form in the JSON editor unless you are very comfortable using the editor. You must take a backup of the form before editing it.

The example form described in the "Example of Creating a Forms" on page 31 can be designed using the JSON editor as well. The JSON version of the designed form is shown in the figure.

Figure 23 *Sample request form in JSON Editor*

Request Form

Offline

```
117        "dbIndex": false,
118        "customDefaultValue": "",
119        "calculateValue": "",
120        "allowCalculateOverride": false,
121        "refreshOn": "",
122        "clearOnRefresh": false,
123        "validateOn": "change",
124        "validate": {
125          "required": false,
126          "custom": "",
127          "customPrivate": false,
128          "minLength": "",
129          "maxLength": "",
130          "minWords": "",
131          "maxWords": "",
132          "pattern": ""
133        },
134        "conditional": {
135          "show": null,
136          "when": null,
137          "eq": ""
138        },
139        "mask": false,
140        "inputType": "text",
141        "inputMask": "",
142        "id": "ew6x8d"
143      },
144      {
145        "label": "Name : ",
146        "widget": {
147          "type": "",
```

You can use the same JSON schema to duplicate the form. You can make edits to the form using the JSON editor directly.

For more information, see Workspaces, Perspectives, and Views in the *Understanding Designer for Identity Manager* guide.

Sample template

A sample example JSON form is available in the `JSONsampletemplate.txt` file. Copy the JSON form and paste it in your JSON editor and make edits or additions to view how the changes appear in the form.

Editing a form in JS Editor

Form Builder provides a global JS Editor which enables you to add or modify the JavaScript methods for all the form fields at one place. From this page, you can add or modify (invoke the API) the required HTTP method (GET or POST) and apply the value to the required field. You need to select from the listed APIs.

Forms can connect to REST servers which are integrated with OSP and defined as part of ServiceRegistry in Forms. OpenText Identity Manager service is registered by default.

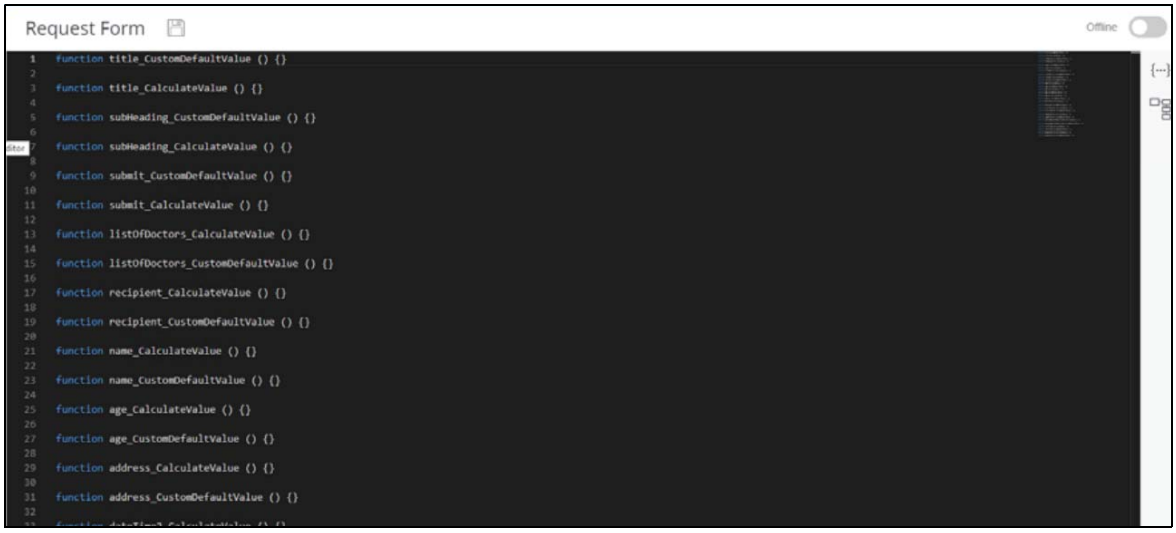
The JS Editor automatically populates the method to set **Custom Default Value** and **Calculated Values** for all the configured fields in the form. You must use this editor to write your own javascript logic for these methods. For any common javascript methods, you must create ECMA script objects and access the methods.

For example, while setting the **value** for a field, the `value` variable is used by default. However, if you use the asynchronous javascript, you must use the `instance.setFieldValue(data)` value. For all the form fields, it is `instance.setFieldValue(value)` except Radio, Select Boxes, and Select fields for which the `instance.setFieldValue` function and its parameters are described as follows:

`instance.setFieldValue (response,valueProperty,labelProperty,defaultValueToSet)`, where:

- `response` {Array}: is an array of objects or strings to be set.
- `valueProperty` {String}: is the value for each option to be set. It is expected that each object in response array has this property.
- `labelProperty` {String}: displays label for the options. It is expected that each object in response array has this property.
- `defaultValueToSet` {String|Number}: when set as string, it selects the corresponding item in the response array that has same valueProperty as the <string>. When set as number, it sets response [<number>] as default selected option. This parameter is optional.

The example of the form described in the "Example of Creating a Forms" on page 31 can be edited using the JS editor as well. The JS version of the designed form is shown in the figure.



On this page, you can include the REST API functions using the REST API icon. For more information, see "REST API" on page 25. Similarly, see "Identity Manager Macros" on page 26 for more information on using the Identity Vault functions.

For more information, see Workspaces, Perspectives, and Views in the *Understanding Designer for Identity Manager* guide.

REST API

The JS editor provides option to include the OpenText Identity Manager Dashboard functions in the form. It lists all the REST APIs available in the OpenTextIdentity Manager Dashboard.

The JS editor provides option to include GET and POST (REST APIs) functions in the form. Click the {...} icon for the supported GET and POST functions to be displayed. You can select the required function to apply the selected value to the required field.

A snippet of the supported functions is shown in the figure.

GET	/rest/access/administration/cprsConfig
GET	/rest/access/administration/delegation
GET	/rest/access/administration/emailApproval
GET	/rest/access/assignments
GET	/rest/access/assignments/advanced
GET	/rest/access/assignments/identities
POST	/rest/access/assignments/item
POST	/rest/access/assignments/list
POST	/rest/access/assignments/list/v2
POST	/rest/access/assignments/resource/list
POST	/rest/access/assignments/role/list
GET	/rest/access/codeMap/{id}/values
GET	/rest/access/config
GET	/rest/access/config/access
GET	/rest/access/config/branding
GET	/rest/access/config/clients

Identity Manager Macros

The JS Editor provides an option to include the Identity Vault functions in the form, similar to building the legacy forms using Designer. Click the icon for the list of supported Identity Vault functions. You can select the required function to design the form accordingly. Migration from legacy to JSON forms will continue to use the Identity Vault APIs.

Although new forms understand only REST APIs, Identity Manager Macros are provided that internally call REST APIs. This approach is similar to building legacy forms in Designer. This eases the form development for developers who are comfortable using these macros while building the legacy forms.

Example: `get(dn, entity-type, attribute)`

This corresponds to the following `IDVault.get()` function that you can use for retrieving data from the Identity Vault:

```
IDVault.get(service-id, api-endpoint, dn, entity-type, attribute)
```

Where,

service-id: Indicates the service identity, such as IDM.

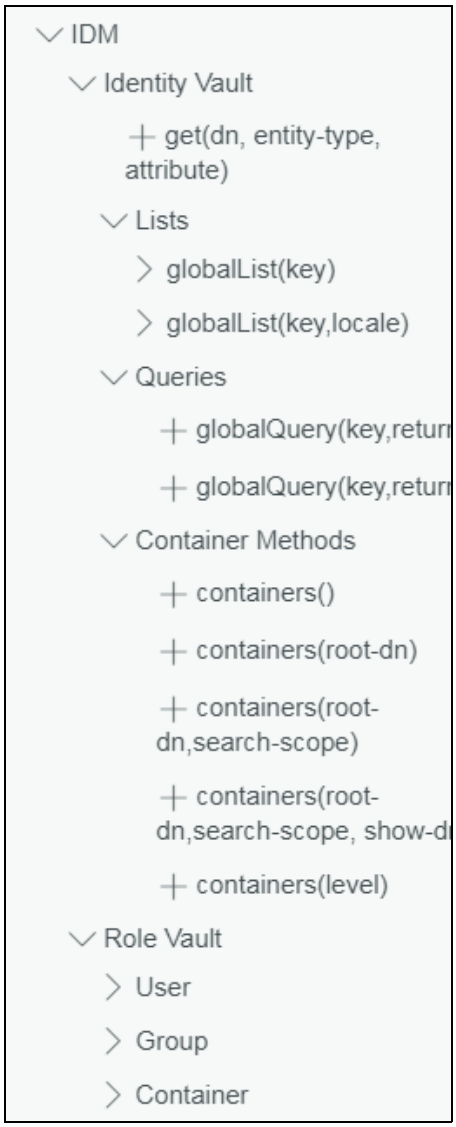
api-endpoint: Indicates the REST endpoint that is called internally. Form Builder populates this field automatically.

dn: Specify the distinguished name (DN) of the entity.

entity-type: Specify the entity type, such as user, group, or resource.

attribute: Specify the attributes of the given entity you want to fetch from the Identity Vault.

A snippet of the supported functions is shown in the figure.



Using Macros to set the Form Fields

Let’s understand how to set options or values dynamically for the **Radio**, **Select**, and **Select Box** elements from a macro or API in the below example.

For Radio element:

```
function radio2_CalculateValue () {
   IDVault.globalList('IDM','/rest/access/entities/globalList','provisioning-
category','en')
.then((data)=>{
    instance.setFieldValue(data,'key','value','nrf'); //Pass the data that you
want to set in the field as a parameter to this function.
})
.catch((error)=>{ })

}
```

For Select element:

```
function select2_CalculateValue () {
     IDVault.globalList('IDM','/rest/access/entities/globalList','provisioning-
category')
    .then((data)=>{
               instance.setFieldValue(data,'key','value',1);  //Pass the data that
you want to set in the field as a parameter to this function.
})
.catch((error)=>{ })
}
```

For Select Box element:

```
function selectBoxes2_CalculateValue () {
IDVault.globalList('IDM','/rest/access/entities/globalList','provisioning-
category')
.then((data)=>{
console.log(data)
    instance.setFieldValue(data,'key','value',{'nrf':true,'groups':true}); //Pass
the data that you want to set in the field as a parameter to this function.
})
.catch((error)=>{ })
}
```

NOTE: In Select Boxes, Select, and Radio elements, when you use macros from JS Editor for custom default value and calculate value, you must add `component.defaultValue` as the 4th argument for `instance.setFieldValue` function for setting the default value from Data item mapping in workflow. For select element, for example:

```
function select2_CustomDefaultValue () {
    IDVault.globalList('IDM','/rest/access/entities/globalList','provisioning-
category')
    .then((data)=>{
      instance.setFieldValue(data,'key','value',component.defaultValue);  //Pass
the data that you want to set in the field as a parameter to this function.
})
.catch((error)=>{ })
}
```

Using Identity Vault Functions to Pre-set the Form Fields

You can pre-load the form fields with data fetched from Identity Vault. The workflow script engine's `IDVault.get()` function helps you to retrieve the attribute values for a given entity from the Identity Vault. You can also retrieve values for multiple attributes for the same entity and set more than one form field, such as a user's telephone number and last name. The following sample expression uses the `IDVault.get()` function with the `entity.instance.root.getComponent('key')` method to illustrate this implementation:

```
function textArea2_CalculateValue () {
IDVault.get('IDM','/rest/access/entities/list','cn=uaadmin,ou=sa,o=data', 'user',
['TelephoneNumber','LastName'])

.then(function (response){

    instance.setValue(response['TelephoneNumber'][0]);

instance.root.getComponent('lastName').setValue(response['LastName'][0]);
});
```

The `IDVault.get()` function retrieves the user's attributes from Identity Vault, namely TelephoneNumber and LastName. It then sets the text field to TelephoneNumber and the lastName with the LastName retrieved in the previous call.

Migrating Legacy Forms to New Forms

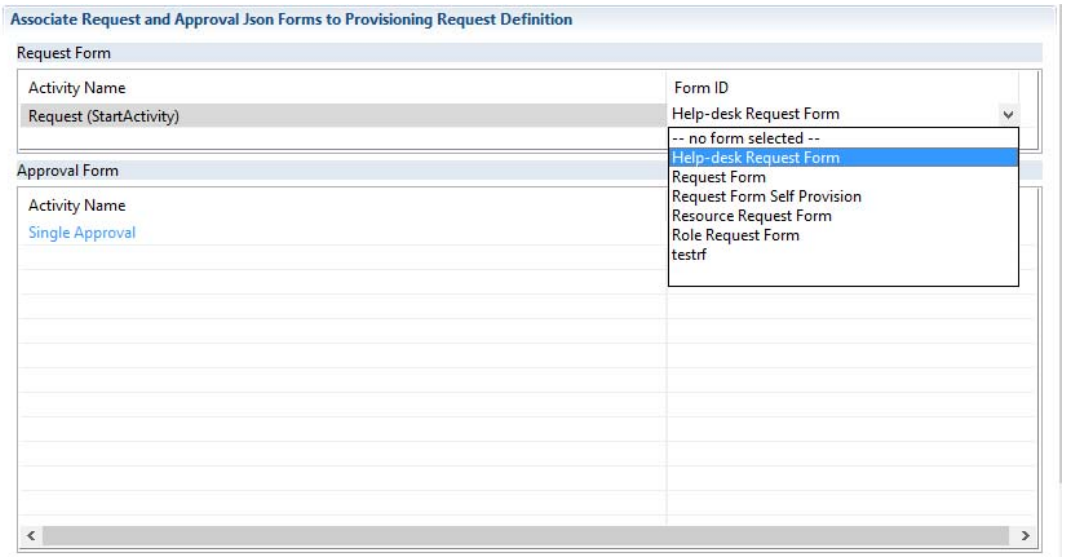
Currently, there is no tool to migrate the legacy forms to the new forms. The only way to accomplish it is by manually creating a new form in the Form Builder and then mapping the data items of the existing form to the new form.

NOTE: When you select the **JSON Forms Selection** check box, Designer deletes all the legacy forms associated with PRDs and the data item mapping fields. This data cannot be retrieved once it is lost. To prevent the data loss, you must back up your workflows before selecting this tab.

---

1. Take a backup of the legacy Provisioning Request Definition by making a copy of the workflow and/or exporting the Provisioning Request Definition to a file.
2. Create a new form in the Form Builder. For more information, see Creating a Form.
3. Associate the new form with your existing workflow.
   - **3a** In Designer, open the editor for the legacy Provisioning Request Definition to be converted to the new forms and select **JSON Forms Selection** check box in the **Overview** tab.
   - **3b** Click **OK** in the warning dialog box. Save the changes.
   - **3c** Navigate to the **JSON Forms** tab and associate the form that you created in Step 2 to the specified workflow activity.

Figure 24 Associating a form



3d Locate the **Form ID** list and select the newly created form from the list.

The new form is associated with the PRD.

4 Perform the Data Item Mapping for the new form.

4a Go to the **Workflow** tab to display the Workflow page.

4b In the workflow, right click the **Start activity** icon or the activity icon to which you wish to associate the form to display the Data Item Mapping view for the selected activity.

4c In the Data Item Mapping view, perform the data item mapping.

The values in the **Target Form Field** are pre-populated from the new form.

5 Deploy the Provisioning Request Definition to the Identity Vault.

Repeat Steps 1-5 for each form that you want to add to the Provisioning Request Definition.

Rendering the New Forms

The forms created in the Form Builder are rendered through the Form Renderer. The Form Renderer provides a better look and feel to the forms in the Identity Applications user interface. Identity Applications continues to render the legacy forms in the Dojo framework. Currently, you cannot render a legacy form in the Form Renderer.

Localizing a Form

After creating the form, click the icon to localize the form to the desired language. The purpose of a form created can vary. Hence, you are allowed to provide the input for each field in the desired language. Specify the required details in the selected language and save the changes. The form fields is displayed with the relevant changes.

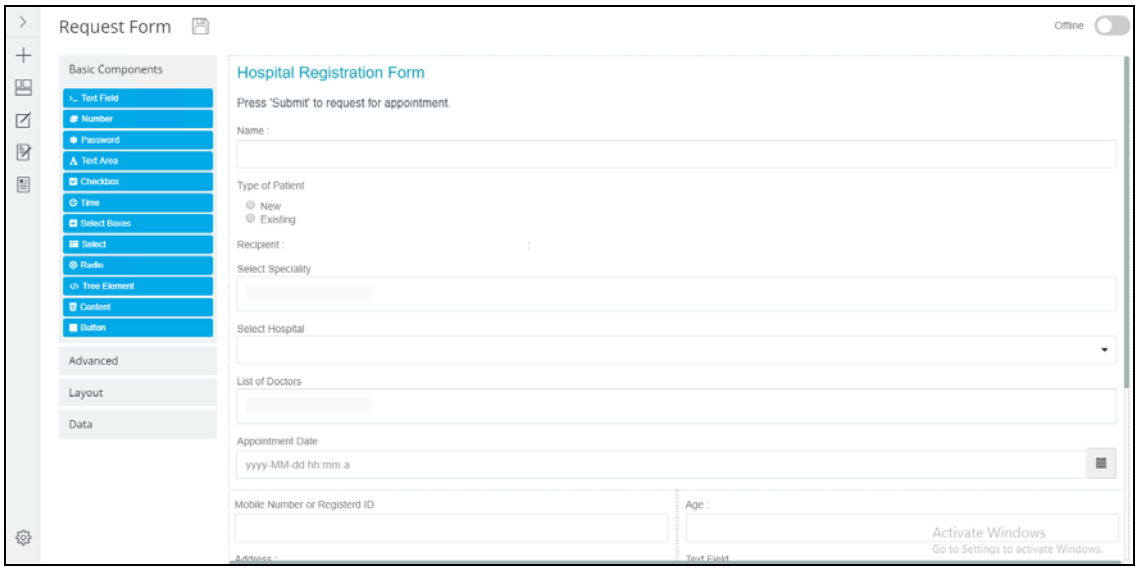
Example of Creating a Forms

This section provides details about creating a from by way of an example.

Let us take an example of creating a hospital registration form that patients use to request for an appointment with a doctor.

Create an Approval from using Form Builder as mentioned in "Creating and Editing Request or Approval Forms" on page 21.

Let us discuss on the widgets required to design the following hospital registration form.



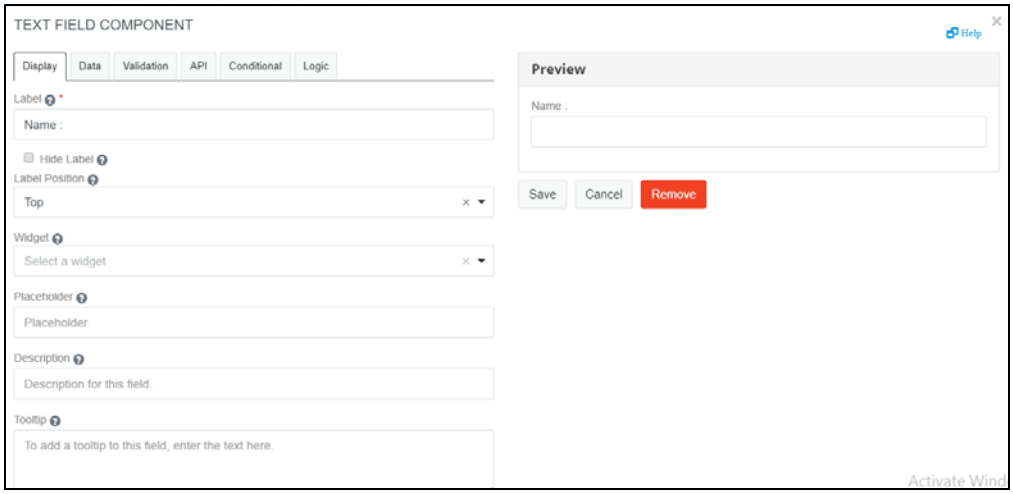
Select the required widgets to design the form. The following components are used to create the registration form.

Field Name	Component/Widget Used	Reference
Name	Text Field (Basic)	For more information, see Text Field.
Type of patient with options	Radio (Basic)	For more information, see Radio.
Select Specialty	Dynamic Entity (Advanced)	For more information, see Dynamic Entity in "Advanced" on page 7.
Select Hospital	Select (Basic)	For more information, see Select.
List of Doctors	DN Query (Advanced)	For more information, see DN Query in "Advanced" on page 7.
Appointment Data	Date/Time (Advanced)	For more information, see Date/Time.
Mobile Number	Number (Basic)	For more information, see Number.
Age and Address	Text Field (Basic)	For more information, see Text Field.

The **Table** widget in the **Layout** component is used to create a tabular layout in the form.

Select the **Text Field** component and in the **Display** tab, enter the **Label** as **Name**. All the other values are as per default values. Save the changes.

Figure 25 *Text Field*



Similarly, select the **Radio** component for the options to appear in a radio button format. Select the **Dynamic Entity** component to create a multi select drop-down field. Select the **Select** component to display a drop down option for the hospital branches to appear in the form. Select the **DN Query** component to provide the list of doctors from the selected department. Select the **Date/Time** component to selected the appointment date and time, and select the **Number** component to display the contact details.

You can move the designed fields across the forms by dragging and dropping. You must save all changes. The Preview of each field is displayed on the right hand side while designing. Click to view the complete designed form.

You can create or make edits to an existing form using the JSON or JS editor. Form more information, see "Editing a Form in the JSON Editor" on page 23 and "Editing a form in JS Editor" on page 24.

Troubleshooting

- "Form Fields Will Populate Incorrect Values If the Provided Value Is Null or Undefined" on page 33
- "Unable to Edit a JSON Form That Uses alert() Form Action Method" on page 33
- "Unable to Create a Form Containing Select Field with Global List Macro" on page 33
- "Data Item Mapping Not Working for Title Element" on page 34
- "Copy of a Form Does Not Reflect the New Name In Form Builder" on page 34
- "The Select Field in a JSON Form Does Not Show All Search Results That Match the Keywords" on page 34
- "Multiple Values Check Box for Some Components Does Not Work" on page 34
- "A JSON Form With Multiple Tabs Layout Does Not Open Properly in the Dashboard" on page 35

Form Fields Will Populate Incorrect Values If the Provided Value Is Null or Undefined

![Bug 1199031] While creating a form field, such as a drop-down list, you can add custom logic to it. The logic is triggered when a specific event occurs, such as selecting an option.

This logic can set values in other dependent fields based on the selected option. For example, if you select a user in the username field, the logic retrieves information about that user, such as their email address and phone number, from the system and populates other fields with that information. However, when you select another user with an email address but no phone number, the email address of the second user is updated, but the phone number still reflects the value of the first user.

Workaround: Include a conditional statement in the custom action logic. For example, if the phone number is undefined or null, the field remains empty.

```
value = data.saveUserDetails['TelephoneNumber'] || '';
if(data.saveUserDetails['TelephoneNumber'] ! = udefined ||
data.saveUserDetails['TelephoneNumber'] ! =null){
  value = data.saveUserDetails['TelephoneNumber']
}
else{
  value = ''
}
```

Unable to Edit a JSON Form That Uses alert() Form Action Method

When you open a JSON form that uses the `alert()` form action method in the Form Builder's Preview mode, a message is displayed in the alert box. However, it has been observed that navigating to the **Form Json** or **JS Editor** thereafter causes the editor to stop working. As a result, you will not be able to add or modify the message.

Workaround: To display an alert message in your JSON form, use `window.alert()` instead of the `alert()` form action method. And, if your window is stuck in Preview mode, click an empty area to switch to different Form Builder mode.

Unable to Create a Form Containing Select Field with Global List Macro

Create a form using the **Select** field in Designer. In the **Data** tab, select 'Data Source Type' as RawJson and provide the item template as `<span>{{ item.value }}</span>`. In JS editor, for the select field custom calculated method, add a global list macro. Deploy the form with a PRD.

Perform the following steps in JS Editor, if the form in OpenText Identity Manager Dashboard does not contain the global list macro for the **Select** field.

1. Navigate to customDefaultValue (Not calculateValue).
2. Add the following code:

```
IDVault.globalList('IDM', '/rest/access/entities/globalList', 'provisioning-
  category').then((response) => {
    instance.setFieldValue(response,'key','value');
  });
```

Data Item Mapping Not Working for Title Element

While creating a form with **Title Element**, if you retain any content in the **Title Content** field, you will not be able to enter value for 'title' form field through Data Item Mapping in Designer.

Workaround: Using Form Builder, modify **Title Element** component by keeping the **Title Content** field as blank.

Copy of a Form Does Not Reflect the New Name In Form Builder

When you copy and paste a workflow form in Designer, the name given to the copied form does not change in the JSON form.

Workaround: You must open the copied form using Form Builder and modify the name using the **Edit** option.

The Select Field in a JSON Form Does Not Show All Search Results That Match the Keywords

The Select element in Form Builder has in-built search functionality. When enabled, the Select drop-down allows users to search from a static list of options. However, the number of search results displayed is restricted to four. As a result, not all of the matching results are shown when a user enters a keyword.![Defect: 513062]

Workaround: Perform the following steps:

1. Open the form in Form Builder.
2. Click **Edit** in the Select component.
3. Click **Data** and add the following line in the **Custom default options** to change the number of search results:

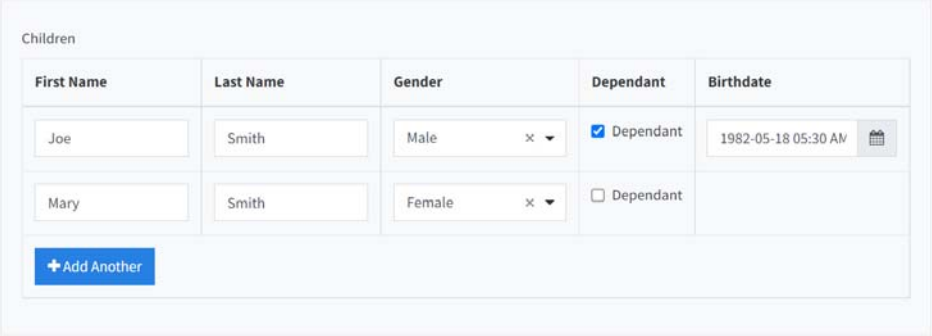
   ```
   {
     "searchResultLimit": <value>
   }
   ```

   Where, `<value>` denotes the maximum number of search results that you want to display in the Select field.

Multiple Values Check Box for Some Components Does Not Work

Issue: When the **Multiple Values** check box in a Number component is selected, the form builder stops responding, and as a result, the data does not get saved. Other form components that offer the multiple input functionality, such as Currency, Phone Number, and Tags, have the same issue.

Workaround: You can provide multiple input capability by placing the same component inside a Data Grid. The Data Grid has an **Add Another** button that duplicates the fields set inside the Data Grid. The following figure shows a sample data grid, Children with multiple components in a line item grid. Users can click the **Add Another** button to enter more than one value in each field.



A JSON Form With Multiple Tabs Layout Does Not Open Properly in the Dashboard

Issue: While requesting a PRD in Identity Applications Dashboard, if the associated JSON form has multiple tabs, the application only renders the first tab (Tab 1) correctly. The remaining tabs do not load. An excerpt from a sample JSON form with three tabs shows that the `"components": []` attribute is missing in Tab 2 and Tab 3.

```
{
  "components": [
    {
      "label": "Tabs",
      "renderAllTabsContent": false,
      "components": [
        {
          "label": "Tab 1",
          "key": "tab2",
          "components": []
        },
        {
          "label": "Tab 2",
          "key": "tab2"
        },
        {
          "label": "Tab 3",
          "key": "tab3"
        }
      ],
```

Workaround: The following procedure shows how to add the missing attribute to the additional tabs:

1. Open the JSON form in Form Builder.
2. Click .
3. Add the "components": [] attribute to both Tab 1 and Tab 2.

```
{
  "components": [
    {
      "label": "Tabs",
      "renderAllTabsContent": false,
      "components": [
        {
          "label": "Tab 1",
          "key": "tab2",
          "components": []
        },
        {
          "label": "Tab 2",
          "key": "tab2",
          "components": []
        },
        {
          "label": "Tab 3",
          "key": "tab3",
          "components": []
        }
      ],
```

4. Save the form.
5. Go to Designer and deploy the form to your Identity Vault.

Appendix

This section provides details on the mapping table between the legacy form widgets and the new Form builder widgets.

Mapping Table Between Legacy Widgets and Form Builder Widgets

The following table lists the legacy widgets and the corresponding widgets that are introduced in the New Form Builder.

Legacy Widgets	Form Builder Widgets
Title	HTML element
Text	Text field
Text area	Text area
Static list	Select
Global list	Select with Global List API
Pick list	Dynamic Entity/Select/Multivalued Select
Label	Label
checkboxPicklist	Select boxes
mvCheckbox	Select boxes

Legacy Widgets	Form Builder Widgets
True/False check box	Check box
Radio buttons	Radio
True/False Radio button	Radio
True/False drop-down	Select
mvEditor	Tags/Multivalued Textfield
DN Maker	Tree with TextField
DN lookup	Dynamic Entity/Select using conditional log
DNquery	DN Query or Select with Global Query API
DNDisplay	DN Display
HTML	HTML element
nrfRequestDN	PermissionRequestDn
nrfResourceRequestDN	PermissionRequestDn
Date	Day

The **Submit**, **Cancel**, **Approve** and **Deny** actions can be performed by defining an appropriate logic in the Form Builder. This can be done by calling an appropriate API for that action. To do so, you must define the API in the **Button Custom Logic** field on the **Display** tab.

Offline - Online Toggle

When the **Online** option is enabled for a form, the **Preview** displays the form in the same way that the form will be rendered in the Identity Applications Dashboard. Note that form association with the PRD does not work in preview. You need to manually configure the service registry for the form to function in the same way as it would render in the identity Application Dashboard. When the form is Online, it retrieves all the entity saved in the Identity Vault and it is populated (as drop-down menu) in the required fields. Hence, you do not need to manually enter the required entity or entity keys.

After you upgrade Designer from 4.8 to 4.8.1 version and launch Form Builder, the `ServiceRegistry.json` is created in the `netiq\idm\apps\Designer48\configuration\` directory.

When the form is Offline, the Form Builder is unable to auto populate the entity. The values (entity or entity keys) needs to be entered manually.

NOTE: The Form Builder preview renders only the form data and does not take care of honoring any script or styling added through workflow directly. However, these are honored accurately when the form actually renders in Form Renderer component.

Perform the following actions to configure the service registry in order to make the form online:

1 (Conditional) If you are using Form Builder 4.8, navigate to `netiq\idm\apps\Designer48\plugins\<name of the form builder><com.mf.linux.gtk.formbuilder_4.0.0.xxxxxxxxxxxx>\lib\commons\ServiceRegistry.json.`

2 (Conditional) If you are using Form Builder 4.8.1 and onwards, navigate to `netiq\idm\apps\Designer48\configuration\ServiceRegistry.json.`

3 Open the `ServiceRegistry.json` in an editor, such as Notepad++. In the `FormsBackedUrl` field, enter the IP address or DNS name of the machine where Identity Applications is installed.

NOTE: If DNS name is provided, make sure the DNS name is configured to a valid IP address.

```
{
  "FormsBackedUrl":"https://<IP address or DNS of the Identity Applications server>:8600/WFHandler",

}
```

4 Save the changes.

5 Launch the form and make the form online. That is, enable the Online toggle. The Identity Applications login page appears. Enter the login credentials. The form designed using Form Builder is now available online. This helps to auto-populate the entity, parameters, and parameter keys.

When the form is online, few widgets display additional fields. For example, Dynamic Entity, DN Query. For more information, see the following sections.

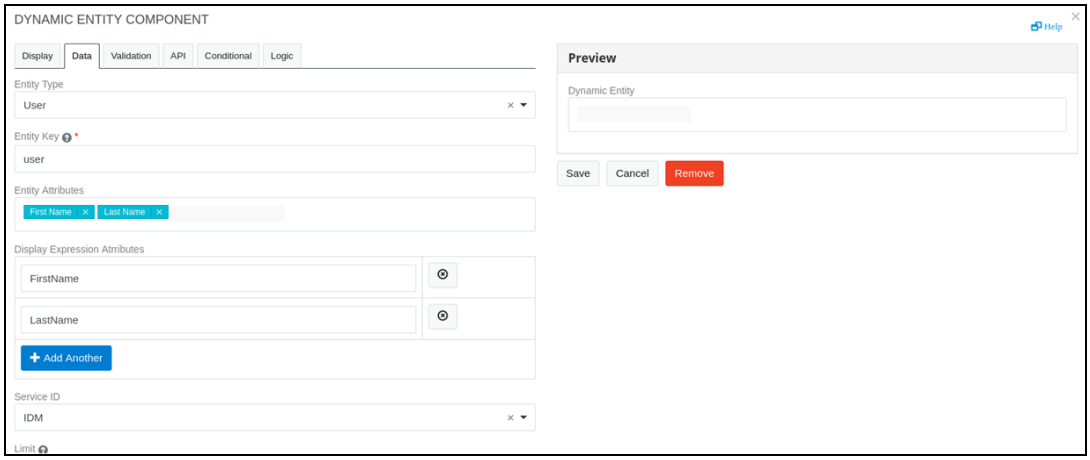
(Widget Changes When the Form is Online) Dynamic Entity widget when the form is Online

Dynamic Entity

The functionality of Dynamic Entity when the form is offline is discussed in "Advanced" on page 7 section.

When the form is Online (enabled), few additional fields are displayed as shown in figure.

Figure 26 *Dynamic Entity when form is Online*



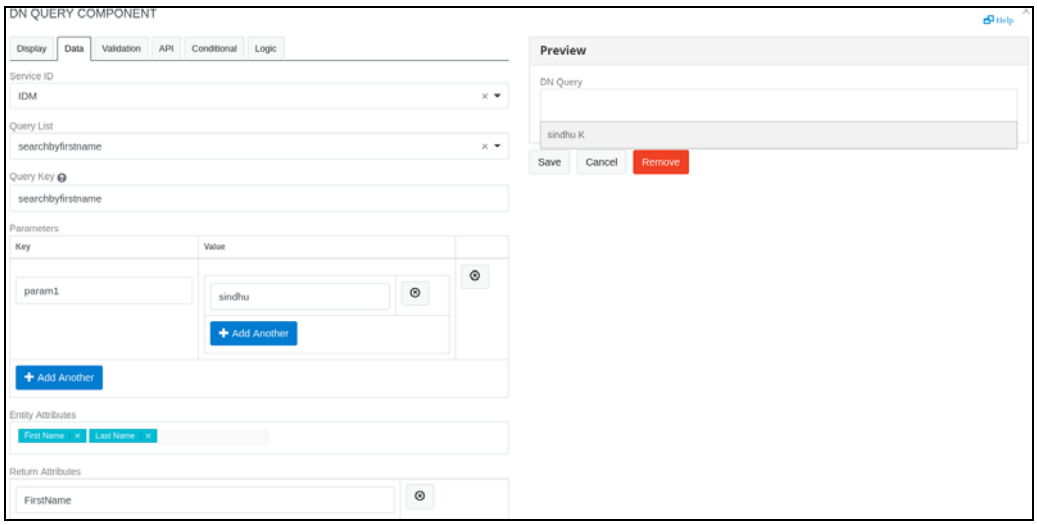
The **Entity Type** and the **Entity Attribute** fields are the new additional fields. As the form is online, the entity and attributes are retrieved from the Identity Vault. Hence, the **Entity Type** option appear as a drop-down. On selecting the Entity Type, the **Entity Key** is auto-populated and the corresponding **attributes** can be selected. You need not enter the **Entity key** and **Display Attributes** values manually.

DN Query

The functionality of DN Query when the form is offline is discussed in "Advanced" on page 7 section.

When the form is Online (enabled), few additional fields are displayed as shown in figure.

Figure 27 *DN Query when the form is Online*



The **Query List** and the **Entity Attribute** fields are the new additional fields. As the form is Online, the queries and attributes are retrieved from the Identity Vault. Hence, the **Query Type** option appear as a drop down. On selecting the Query Type, the **Query Key** is auto populated and the corresponding **Entity Attributes** can be selected. You need not enter the **Query Key** and **Entity Attribute** values manually.

Contacting OpenText

For specific product issues, contact OpenText Support at https://www.microfocus.com/support-and-services/.

Additional technical information or advice is available from several sources:

- Product documentation, Knowledge Base articles, and videos: https://www.microfocus.com/support-and-services/
- The OpenText Community pages: https://www.microfocus.com/communities/

Legal Notice

Copyright 2009 - 2024 Open Text.

The only warranties for products and services of Open Text and its affiliates and licensors ("Open Text") are as may be set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Open Text shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.