Forms

Version: 8.2

Revised: June 2021

Copyright and Trademark Notices

Copyright © 2021 SailPoint Technologies, Inc. All Rights Reserved.

Contents

Forms

Forms are used to present items to users for input in several components of IdentityIQ. They are used with:

- Application and role provisioning policies
- Identity provisioning policy (only applicable for installations using Lifecycle Manager)
- Data entry and approvals in workflow steps
- Present simplified views for process variable and step argument editing in workflows
- Report filter specification

The form implementation and available features varies slightly in these areas, so some features might apply to one use and not to the others, this is noted throughout this section.

For more information about using Forms with IdentityIQ, refer to the forms documents on the SailPoint customer support site or contact your support agent for more information.

Specifying Custom Forms

Form specification is different for each available use. All types of provisioning policies can be specified through the IdentityIQ user interface. In all cases, some of the more advanced and custom forms for workflows can be generated through the Business Process Editor. Some of the more advanced options. however, are only available through subsequent editing of the XML definition. Workflow forms created through the Business Process Editor are embedded within the workflow XML. Workflow forms created through the Business Process Editor are embedded within the workflow XML. Alternatively, they can be defined as independent form objects which can be referenced by multiple workflows, by creating them directly in XML and importing them into IdentityIQ. Report forms must be created as external XML documents and imported into IdentityIQ.

Role/Application Provisioning Policies

Provisioning forms are presented to a user when a provisioning request cannot be completed without user input. The data collection fields that are presented on the form come from the role or application's Provisioning Policy, which is defined by the <Form> element inside the Bundle (role) or Application object's XML. The actual form presented to the user during provisioning of roles or application accounts are system-generated at run-time based on skeleton forms that are pre-defined in IdentityIQ. Requests made through LCM are built with the Identity Update form. Requests that come through the Identity Refresh workflow use the Identity Refresh form. These forms contain a read-only section at the top that displays identifying information about the request, for example, Account ID, First name, Last name. The fields defined in the provisioning policy forms are added to the form at run time, when the form is presented to a user.

Provisioning policy forms define the fields required for the role or application account to be provisioned, often including a default value or script/rule for calculating a value. When a field cannot be calculated by the system during provisioning of an account or role, it must be presented to a user through a form to get the required value. When multiple accounts or roles are part of the same provisioning request, the form might display a collection of fields pulled from various provisioning police forms. On the form, the fields are, by default, grouped in sections according to the application or role to which they belong; this grouping can be overridden, by specifying a section attribute on each of the fields, naming the section into which each field should be placed. See the section attribute description in Fields.

Defining Application Provisioning Policy

An application provisioning policy can be defined for an application on the Provisioning Policies tab of the Application Configuration page, Applications -> Application Definition. Separate policies can be defined for create, update, and delete requests. Additionally, provisioning policies can be specified for creating or updating groups.

The required fields should be specified in the policy with the appropriate field attributes defined. These attributes can include a default value or a script/rule to calculate a default value for the field that can be based on the Identity attributes for the Identity for whom the request is being made. The field **Name** should match the corresponding native attribute on the application. If **Review Required** is selected, the field is always presented on a form during provisioning-request processing, even if a default value is provided or calculated successfully.

For creation-type operations you can specify dependencies between applications and application attributes that imply ordering of the provisioning requests.

Field Properties and Value Properties

The provisioning policy field attributes are grouped into two categories: Field Properties and Value Properties.

Field Properties describe field meta data. This includes the field's name, display name, tool tip help text, type, and owner. It also includes indications of whether the field is single or multi-valued, read-only, hidden, required, or review required. Fields can also be marked with a flag to indicate whether changes to the field value should cause the form to be reloaded. The Read-Only and Hidden attributes can be set to a static value (True or False) or can be defined

programmatically through a rule or script. The rule and script options are used to dynamically hide and show the field, or change its edit properties, when the form is reloaded based on changes to values of other fields.

The Value Properties section includes properties specifically related to the field's value. A default value, a set of permitted values, and the field's validation logic can all be set here. The Dynamic attribute determines whether the field's value should be reevaluated on every form reload, when the form is reloaded based on a change in another field's value. It should only be selected when the field's value is rule or script based, such that it might change during the form processing based on other field values entered there.

The default value can be specified as a static value or can be calculated programmatically by a rule or script. In an account creation provisioning policy, an additional option, Dependent, is available as part of the ordered provisioning implementation, which is only available on account create provisioning policies. When the dependent option is selected, an application and attribute must also be selected and the value of the field is set to that attribute value for the Identity. Only applications on which this application is dependent are available for selection here.

The **Allowed Values** list can be specified as a list of values or can be set dynamically by a rule or script. Field validation is optional and can be managed by a reusable rule or with a script.

Defining Role Provisioning Policies

Role provisioning policies are specified through the Role Editor: go to Setup -> Roles, select the role name, and click **Edit Role**. Then click **Add Provisioning Policy** and specify the fields for the policy.

Select the application to which the role provisioning policy applies and then specify the fields for the policy. Fields are specified for role provisioning policies exactly as they are specified for application provisioning policies. Role provisioning policies and application provisioning policies are not the same or to be used interchangeably, however.

Role provisioning is not intended for initial role assignment or for the provisioning of account attributes that are not entitlements. Using role provisioning and application provisioning interchangeably cause conflicts and should be avoided.

Role provisioning is designed to be used for profiles that use complex logic, where it is unclear what should be provisioned or de-provisioned. The role provisioning policy is used to state what to provision, "x and y" or "p and q," and to use the contents of the Identity to make that decision.

Identity Provisioning Policy

The Identity Provisioning Policies are optional forms that can be specified to define the fields that must be provided when an Lifecycle Manager Create or Edit Identity request is submitted. When no Identity Provisioning Policy is defined for the create function, IdentityIQ automatically builds a form that includes the entire set of defined Identity attributes (standard and extended) for the installation. The auto-generated update provisioning policy form contains only identity attributes marked as editable. An Identity Provisioning Policy can be defined to select a subset of those fields, to affect the presentation of those fields, for example, grouping in sections or multi-column layout, or to build in some logic to auto-populate some of the fields.

A third identity provisioning policy also exists to support self-service registrations for IdentityIQ. This form is presented when self-service registration is enabled and a new user requests an IdentityIQ account. The form prompts the user for the information required to create a new user account for the installation.

To create an Identity Provisioning Policy, go to Identity Provisioning Policy of the Lifecycle Manager configuration page. Three policies are available: Create Identity, Update Identity, and Self-service Registration. If a policy has already been defined, the name is displayed. Click the name to open and edit the policy. If no policy has been defined for one of these types, click **Add Policy** to add a new one. Add fields to the policy, defining field attributes as needed on the field definition parallels for an application or role provisioning policy.

Identity Provisioning Policy forms are saved as independent form objects. System Configuration entries (entry key-y="createIdentityForm", "updateIdentityForm", and "registerForm") point to the appropriate forms for each identity provisioning policy by name. The identity provisioning policy forms are saved as <Form> objects inside the UIConfig attributes map under the keys lcmCreateIdentityProvisioningPolicy and lcmUpdateIdentityProvisioningPolicy on the IdentityIQ Debug pages. These form definitions can be edited directly to implement some of the presentation options, for example, multi-columns or sections. The configurable option available on the user interface do not include these features.

> Form features related to the Section attribute (which includes subdividing the form into sections and creating multi-column form configurations) are not supported through the user interface. These must be managed directly in the Form Object XML. Any fields added through the user interface after dividing the form into sections are automatically added to the first section. These fields can be moved to other sections by editing the XML.

Workflow Forms

Several standard work item renderers are provided with IdentityIQ for presenting approvals or other data requests to users. These are written as JSF pages. It is possible to write custom forms in JSF, specifying the JSF page as the renderer for the approval. This is rarely done. Customers who want to use custom forms generally specify these through a Form object.

Forms are used in workflows to present data-gathering pages to a user and define data presentation for approval activities. In many cases, implementations rely on the standard approval work item forms for normal approval actions so do not need to implement custom forms for their approval steps, but they still might choose to use custom forms for non-approval data-gathering activities to which the normal approval forms do not apply. A custom form can be added to a workflow through the Business Process Editor (Setup -> Business Processes) by right-clicking a step and choosing **Add Form** or by adding a form element to a step in the Workflow XML.

Whether the form is specified for an approval or a data-gathering activity, the form element must be embedded within an approval element in the XML. The user interface auto-creates it within an approval element. The workflow XML to specify a custom form looks like this

```
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE SailPoint PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<sailpoint>
<Workflow  explicitTransitions="true"  libraries="Identity"  name="Example Workflow" type-
e="IdentityLifecycle">
…
  <Step name="Display Form">
     <Approval name="Please enter some data" owner="admin" return="selectedApprover" send-
d="trace ">
       <Form>
        …         <!-- Form content goes here -->
      </Form>
    </Approval>
  </Step>
```

A custom form can also be created as an independent form object, defined in a separate XML document and imported into IdentityIQ, visible through the console or the Debug pages by viewing the Form objects, and referenced in the approval element as an argument like this:

```
<Approval…>
     <Arg name='workItemForm' value='Custom Form Name'/>
      …
 </Approval>
```

This option promotes form reuse across workflows. However, these independent form objects cannot be edited through the Business Process Editor like the embedded forms can.

Process Variable and Step Forms in Workflows

While forms added to steps on the Process Designer tab of the Business Process Editor are used to request data required by the process from a user, such as a value for a missing attribute, the process variable and step forms are used to define the information presented on the Basic Views of the Process Variables tab and the Arguments tab of the Step Editor.

These forms are created as an independent form object, defined in a separate XML document and imported into IdentityIQ. They are visible through the console or the Debug pages by viewing the Form objects.

The process variables forms are used to simplify the information displayed on the Process Variables tab by hiding those variables that are rarely, if ever, modified and displaying variables in more logical groups. Changes made in the Basic View are persisted to the Advanced View and more complex configuration can be performed there if needed.

The step forms are referenced from the workflows or stepLibraries. These forms define the form that is presented on the Arguments tab of the Step Editor panel and works similarly to the process variable forms.

Both of these forms are referenced from workflows using the configForm variable. The forms can be defined and viewed and edited on the IdentityIQ debug page.

Report Forms

Report definitions often include a reference to a Form object for displaying the report-specific filter options to the report user. In the Report XML, the form is referenced with a <ReportForm> tag:

```
<ReportForm>
<Reference class="sailpoint.object.Form" id="39535985298ff9839ff98dd" name="My Custom
Report Form" />
</ReportForm>
```

The Form is defined separately in its own XML document and imported into IdentityIQ as a Form object. Each section within the form is created as a separate page in the Edit Report window, where you specify the filters that are applied to the report. The report-specific forms are always merged with the Report Form Skeleton, which defines the Standard Properties and Report Layout pages that apply to every report.

Components of a Form Definition

The basic elements in a Form definition are:

```
<Form>
   <Attributes>    (map of name/value pairs that influence the form renderer)
   <Button>     (determine form processing actions)
   <Section>    (Subdivision of form; may contain nested Sections and Fields)
     <Field>    (may contain Attributes map, Script to set value, Allowed Values Defin-
ition script, and Validation Script)
```

Within each of these sections of the form definition, certain attributes might apply to some form uses and not to others. The table below provides a high-level overview of which of the available form elements can be specified for each.

Form Usage	Form Component Availability			
	Field	Button	Section	Field
Application and role provisioning policies (Form)				✓
Identity provisioning policy	✓*	✓*	ü	ü
Workflow approval	ü	ü	ü	ü
Report			ü	ü

* Limitations on the Attribute and Button elements for Identity provisioning policy are discussed in Attributes and Buttons.

Form

The Form element should contain a single attribute to define the form: a name.

```
<Form name="My Custom Form">
```

If the form is stored in the database as an independent Form object, the name must be unique; no two Form objects can share the same name. This restriction does not apply to Forms defined in-line within a workflow approval step. While a name is required for independent Form objects, it is recommended but is not required on in-line forms.

Attributes

Forms can include a map of attributes that are used by the renderer. These are applicable only to workflow forms and, in a limited way, to the identity provisioning policy form.

Attributes are specified with the following keys:

Key	Description
pageTitle	Title to render at top of page, typically larger and a different color than

Key	Description
	the form title; also displayed in browser window header bar in some cases
title	FormTitle, shown at top of form body
subtitle	Form subtitle, shown below title
readOnly	`<entry key="readOnly= "" value="true""/>` makes form read-only so the fields are rendered as uneditable text or as disabled HTML components
hideIncompleteFields	`hideIncompleteFields="true"` hides any fields that do not have all of their dependencies met. Usually only set programmatically to control field presentation in provisioning policy forms, though can be specified in a workflow form XML. This does not create dynamically displayed fields on forms. Fields are not displayed on a form after their dependency values on the same form are entered. Use the new hidden attribute on Fields and Sections to achieve this dynamic display functionality.

The Boolean attributes are only specified if they are true; they default to false when omitted.

Attributes maps are specified as shown here:

```
<Attributes>
  <Map>
     <entry key="pageTitle" value="Review Non-Employee Request"/>
     <entry key="readOnly" value="true"/>
  </Map>
</Attributes>
```

Attributes do not apply to report forms because the sections in a report form are pulled out of the report's form definition and combined into the Report Form Skeleton for display in the user interface. Even if they are specified in the report form, these attributes are never applied to the resultant form that is displayed to the user.

On an identity provisioning policy form, the pageTitle attribute is ignored because the main page title is programmatically set based on the other action being performed (Create New Identity, Edit Identity Attributes for [Username], or New User Registration). The title and subtitle attributes are displayed in the user interface when specified in the form's attributes map. The readOnly and hideIncompleteFields function on this form type should not be used because they do not provide useful functionality for this type of form.

Buttons

Buttons enable the user to indicate which action to take next and how to process the data on the form. Buttons only apply to workflow forms. Buttons can be specified on identity provisioning policy forms, but the window does not support any action on them other than next (submit). Since **Submit** and **Cancel** buttons already exist on the window and perform the appropriate functions for the window, additional buttons are unnecessary. They cannot be specified in a

role or application provisioning policy form, and they are not used by the report executor when it combines the specific report's form with the Report Form Skeleton.

Buttons require two attributes, a label and an action. The label determines the text displayed on the button. The action determines what the system does in response to clicking that button. There are four available actions:

- **next**: save (and validate fields with validation scripts where specified) any entered form data and set the work item status to approved. This can then be used in the Transition logic to advance the workflow to the next step (OK/Save/Approve/Submit functionality).
- **back**: save entered form data (no validation is performed) and set the work item status to rejected. This can then be used in the Transition logic to return to a previous step or any other appropriate action for a rejection. Saved value is redisplayed on this form if the workflow logic process back through this step again.
- **cancel**: close the form, suspend the workflow and return to previous page in user interface, this leaves work item active. awaiting a different action choice by the user.
- **refresh**: save the entered form data and regenerate the form; not a state transition - just a redisplay of the form (rarely used).

These are examples of button elements.

```
<Button label='Submit' action='next'/>
<Button label="Cancel" action="cancel"/>
```

Sections

Sections divide a form into logical groupings that are visually marked on the window with boxes around the fields in each section. They can be specified in the XML for all policies except application and role provisioning policies. By default, a separate section is created on the provisioning form for each application (each application's provisioning policy form is rendered in its own section). However, fields in a provisioning policy form can be specified with a section attribute that causes them to be displayed in different sections from the defaults. Sections are treated differently on report forms, each section becomes a separate page on the Edit Report window rather than just a separate section on a contiguous form.

Sections are specified in the form object's XML with a `<Section>` tag and can be modified by the attributes shown in the table below.

Section Attributes	Description
name	Internal name for section (might be referenced by field objects in some forms).
label	If non-null, the label is displayed above the section fields in a box on the section border. For report forms, the label is specified in the Edit Report window's sections list. Labels can be specified with text, message catalog keys, or variables (specified with $(varName) notation).

Section Attributes	Description
type	Rendering type (optional). When no type is specified, fields in the section are editable fields, displayed one field per row, unless the columns attribute specifies otherwise. Other type options are: **datatable**: makes fields in the section non-editable; generally used to display a read-only informational table to give the form user a context for the form's requested data **text**: indicates the section is a block of informational text; if multiple fields are included in a text section, each field is rendered on a separate line with line breaks between them
columns	Number of columns contained in the form section; fields are placed in columns left to right, one field per column before moving to the next row. For example, in a 2 column layout (columns="2"), 4 fields are displayed: Field 1 Field 2 Field 3 Field 4

These are examples of Section elements in the XML for forms.

```
<Section name="authorizations" label="Authorizations" type="datatable">
<Section columns="2" label="rept_app_section_label" name="customProperties">
```

Sections contain nested field elements and might contain nested sections when sub-groupings are needed.

Fields

Fields are the core element of forms, they are the mechanism by which data gets communicated to and from the user. Fields offer options that affect the appearance or functionality of the field. Some of these are commonly used and others are used very infrequently. Some of these are specified as in-line attributes in the `<Field>` tag and others are specified as nested elements within the Field.

Field attributes appropriate to all

Field Attributes	Description
name	Name for the field that can be referenced in code as the variable name in which the field's value is stored. Avoid using the following field names: accept accept-charset action

Field Attributes	Description
	autocomplete enctype method name novalidate target As well as global attributes: accesskey class contenteditable contextmenu,data-* dir draggable dropzone hidden id itemid itemprop itemref itemscpe itemtype lang spellcheck style tabindex title
displayName	Label for the field; can be text or a message key.
helpKey	Tool tip help text; can be text or a localizable message key. Example: `<Field name="firstName" displayName="First Name" helpKey-y="Enter the person's first name" />`
type	Field datatype; influences the display widget used to display the field on a form. **Valid values are:** string, int, long, boolean, date, and SailPoint object types (Identity, Bundle, Permission, Rule, Application), default is string. SailPoint objects are displayed as drop-down lists or combo boxes if multi="true" is also specified. Specifying type="boolean" renders the field as a checkbox; specifying type="date" adds a calendar from which the date can be selected.

Field Attributes	Description
	To pre-select an object in the list, specify the name of the object (not an actual object) as the "value" attribute. Example: `<Field name="role" displayName="Role" type="Bundle" value-e="TRAKK Basic" />`
multi	Boolean indicating whether the field is multi-selectable. This attribute is only appropriate to drop-down lists, which are then displayed as combo boxes. Used this with `SailPoint` object field types or with a nested AllowedValues / AllowedValuesDefinition element that populates a selection list for the field. Example: `<Field name="apps" displayName="Applications" type-e="Application" multi="true"/>`
readOnly	Boolean indicating that the field cannot be edited on the form. The value is displayed as text not in an editable box. Not necessary to specify on fields in a datatable section, since they are already read-only.
hidden	Boolean that, when true, prevents the field from being displayed on the form. This attribute is used in reporting to make fields available for inclusion on the report detail grid but not actually include them by default. This can be used in any form, but might not be commonly implemented: • In role/application provisioning policies, fields are only shown if the user needs to enter data, so forcing fields to be hidden is not helpful. • In workflows, the hideIncompleteFields attribute on the Form object is more likely to be used with the dependencies attribute on Field to defer field display until dependencies are fulfilled. • In Identity provisioning policy, fields that should be hidden can be omitted from the form instead, however, this could be used for fields that always contain the same value for all users to set that value and suppress the field from the data entry form. For example, `<Field name="status" hidden="true" value="NewHire" /` Attributes mark hidden are not included in the plan. You must manually add the includeHiddenFields property to the form to include the hidden fields in the plan. `<entry key="includeHiddenFields" value="true"/>`

Field Attributes	Description
required	Boolean indicating whether a value is mandatory for the field; required="true" marks field with * on form to indicate required and prevents form submission without a value for the field Example:`<Field name="myfield" displayName="My Field" required="true"/>`
postBack	Boolean that, when true, causes the form to refresh when the field's data value changes, running any rules or scripts that run on form load `<Field name="application" displayName="Application" type-e="Application" postBack="true"/>` Supports conditional display/editing of sections or fields based on other field attributes values, automatic population of fields based on other fields, and validation of fields based on actions taken on the form. It only runs when a field loses focus, so it can be used on selection fields or text entry fields.

columnSpan	Used when the section is configured with multiple columns; specifies the number of columns the field should span
	
	`<Section columns="2" label="Identity Info" name-e="identInfo">`
	
	`<Field name="fname" displayName="First Name" colum-nSpan="2"/>`

filterString	Used for fields where type is a `SailPoint`Object class to specify a filter to restrict the set of selectable objects presented in the drop-down list. filterString is specified according to the filter string syntax and should be specified in single quotes so double quotes can be used within the string. Example: `<Field displayName="Role" name="role" type="Bundle" fil-terString='name.startsWith("TRAKK")' />`
section	Statically defined fields in a form's XML are defined within a section element, so any section attribute specified on those fields is ignored. However, the section attribute can be used to organize fields in an application or role provisioning policy or on a dynamically rendered form. Application and role provisioning policy forms do not have section elements, so the section attribute can be used to force fields to be grouped differently than the default (default is by application or by role). Example: These two fields are put on the form in separate sections, labeled "Important

Field Attributes	Description

	Items" and "Optional Items" respectively.
	
	`<Field name="myField" displayName="My Field" section="Important Items"/>`
	
	`<Field name="optField" displayName="Optional Field" section="Optional Items"/>`
	
	The section attribute on fields is also used in dynamically created forms (such as in Reports where fields are added to the form programmatically through an initialization script). This attribute enables the code to specify the section of the form into which the field should be added.

displayType	Forces string fields to display as specified, used only for string fields
	
	Valid displayTypes are: radio, combobox, textarea, and label
	
	displayType="radio" and "combobox" are used to override the default display format for permitted-values fields (radio is the default for 2 options while >2 options is rendered as comboBox by default). textarea is used to make a string field display as a text area instead of a regular entry field.
	
	`For label, you can use the field displayName for the text/message key of the label.`
	
	`<Field name="dept" displayName="Department" displayType="radio"<AllowedValues>`
	`<String>Accounting</String>`
	`<String>Manufacturing</String>`
	`<String>Engineering</String>`
	`</AllowedValues>`
	`</Field>`
	`<Field name="comments" displayName="Comments" displayType="textarea" />`

value	Sets the default/initial value for the field. This can be overwritten on the form in most cases as long as the field is not marked readOnly. This is used within sections of type="text" to specify the text to display For application or role provisioning policies, setting a value (whether with this attribute or through a nested <Value>, <RuleRef>, or <Script> element) prevents the field from being included on the form unless reviewRequired is specified since provisioning policies only collect values from a user that they cannot determine or calculate independently. In workflow approvals, value can be specified by string, rule, script, call, or reference (string is default).

Field Attributes	Description
	In reports forms, the value is a reference to the report taskDefinition's input parameter from which to retrieve the starting / default value for the field, for example, value="ref:applications". Example: `<Field name="role" displayName="Role" type="Bundle" value-e="TRAKK Basic"/>`

dynamic	This attribute performs two separate functions:
	
	(1) For fields with an AllowedValuesDefinition, delays running of allowed values scripts/rules until the field is clicked, instead of running at form load, so it can make use of other data entered on the form instead of just data available on initial form load.
	
	(2) During form refresh in response to a value change of a field marked for postBack, only fields marked as dynamic (dynamic="true") have their value scripts/rules re-run; otherwise, the initial value calculated for the field on form load remains in effect as the field's default value
	
	Example:
	
	`<Field name="fullName" displayName="Full Name" dynamic="true">`
	`<Script>`
	`<Source>`
	`return (firstName + " " + lastName);`
	`</Source>`
	`</Script>`
	`</Field>`
	
	Field value recalculations for fields marked as dynamic are not processed on postBack if a value has been entered manually in the field, based on the assumption that if a user manually enters a value, they generally do not want that overridden by an automatic process. To override this behavior, manually clear the fields using a hidden script elsewhere in the form like this:
	
	`<Section>`
	`<Attributes> <Map>`
	`<entry key="hidden">`
	`<value> <Script> <Source>`
	`boolean hidden = false;`

Field Attributes	Description

	`// Null out field - add condition here if desired`
	`form.getField("nickname").setValue(null);`
	`return hidden;`
	`</Source> </Script> </value>`
	`</entry>`
	`</Map> </Attributes>`
	`<Field displayName="Nickname" dynamic="true" name="nickname" type="string">`
	`<Script><Source>`
	`if ("Robert".equal(firstName))`
	`return "Bob";`
	`</Source></Script>`
	`</Field>`
	`</Section>`

These attributes only apply to the application and role provisioning policies:

Field Attributes	Description
dependencies	List (CSV) of other fields that must be evaluated before this field. Dependencies on the provisioning policy (form) field cause that field to be deferred to a subsequent form that is presented after the form on which its dependencies are presented. The field might also be calculated based on those dependencies instead of presenting it on a later form. This attribute can also be used with dynamic/allowedValues fields. Values of dependencies fields are made available to the allowedValues script or rule, even if the field is presented on a different form.
reviewRequired	Enables a default value to be assigned to the field while still including the field on the form displayed to a user. This enables the default to be edited. If reviewRequired="true" is not specified, provisioning policy form fields with a default value (or value script/rule that returns a value) are omitted from the user-facing form and the default value is automatically used.
authoritative	Boolean that specifies whether the field value should replace the current value rather than be merged with it. Valid for multi-valued attributes only: `<Field name="costCenter" displayName="Cost Center" multi="true" authoritative="true"/>`

Fields can also contain nested elements that help control the display or use of the field.

Nested Elements within Field Elements	Description

Description	Field description, used for XML self-documentation. Not displayed in user interface.
	
	`<Description>`
	`This field stores the Identity's first name.`
	`</Description>`

Attributes	Attribute map used to control field rendering, specific to the field type. The most common attributes are height and width which are usually specified for textarea fields and for entry boxes that need to be other than the default rendering size. Units for height and width are in pixels
	
	`<Attributes>`
	`<Map>`
	`<entry key="height" value="200"/>`
	`<entry key="width" value="100"/>`
	`</Map>`
	`</Attributes>`
	
	Two special attributes - xtype and vtype - are discussed in the section below.

Value	Alternative to "value" attribute on <Field>. This is required when specifying complex datatypes such as Map or List.
	
	`<Value>`
	`<List>`
	`<String>Thing 1</String>`
	`<String>Thing 2</String>`
	`</List>`
	`</Value>`
	
	Also needed for fields of type Date, which are specified as the utime representation of the date:
	
	`<Value>`
	`<Date>1231971297</Date>`
	`</Value>`
	
	Can be used to specify simpler types like String, Boolean, etc. but not commonly done because value attribute is simpler.

Script	Script used to initialize the value of the field, alternative to the value ele-

Nested Elements within Field Elements	Description

	ment/attribute. Automatically created for fields whose value is set by script through user interface specification.
	
	Example:
	
	`<Script>`
	` <Source>`
	` [beanshell code goes here]`
	`return [value or variable that contains value to assign to the field];`
	` </Source>`
	`</Script>`

RuleRef	Reference to a reusable rule for initializing field value. Alternative to <Script> (and value attribute). Automatically created for fields whose value is set by script through user interface specification.
	
	`<RuleRef>`
	`<Reference class="Rule" name="My Rule" id="402839343985ff930d" />`
	`</RuleRef>`

AllowedValues	Specifies a set of values from which the user can select to assign the field value. Automatically created for fields with an allowed values property set to Value (with a list of values specified) through user interface specification.
	
	`<Field name="dept" …>`
	` <AllowedValues>`
	` <String>Accounting</String>`
	` <String>Manufacturing</String>`
	` <String>Engineering</String>`
	` </AllowedValues>`
	`</Field>`
	
	The list renders as radio buttons when only two options exist (and multi is not allowed), as a listbox for more than two options, and as a combobox for multi-selectable fields.

AllowedValuesDefinition	Populates a list of values from which the user can select a value for the field. This field contains either a <Script> block that specifies the list programmatically or a <RuleRef> that points to a rule containing the beanshell for generating the list. Automatically created for fields with an allowed values property set to Script or Rule through user interface specification.

Nested Elements within Field Elements	Description

	<AllowedValuesDefinition>
	<Script>
	<Source>
	import sailpoint.object.*;
	List l = new ArrayList();
	for(WorkItem.State enumItem : WorkItem.State.values
	()) {
	List l2 = new ArrayList();
	l2.add(enumItem.toString());
	l2.add(enumItem.getMessageKey());
	l.add(l2);
	}
	return l;
	</Source>
	</Script>
	</AllowedValuesDefinition>
	
	Alternative to AllowedValues element and more commonly used. The list renders as radio buttons when only two options exist (and multi is not allowed), as a listbox for more than two options, and as a combo box for multi-selectable fields.

ValidationScript	Script used to examine and validate the field value entered by the user. The value entered is passed to the validation script in the variable named "value."
	
	<ValidationScript>
	<Source>
	if (value > 10) {
	return "Value must be less than or equal to 10";
	else
	return null;
	</Source>
	</ValidationScript>
	
	Returns null if no errors and an error message (as string or `SailPoint`**.tools.message** object) if validation errors exist.

ValidationRule	Reference to reusable rule for field validation. This is the alternative to ValidationScript. <ValidationRule> <Reference class="Rule" name="My Validation Rule" id="4028392342f5ff9301" /> </ValidationRule>

Nested Elements within Field Elements	Description

OwnerDefinition	Used only for application and role provisioning policies to determine the Identity to whom the fields should be presented. This enables specification of a RuleRef,
	script, a Value element or a Value attribute:
	
	`<OwnerDefinition>`
	` <RuleRef>`
	` <Reference class="rule" name="My Owner Rule" id="4038293483598523" />`
	` </RuleRef>`
	`</OwnerDefinition>`
	`or`
	`<OwnerDefinition>`
	` <Script>`
	` <Source>`
	` import sailpoint.object.*; Identity myIdentity=context.getObjectByName(Identity,"Walter.Henderson");`
	` return myIdentity;`
	` </Source>`
	` </Script>`
	`</OwnerDefinition>`
	`or`
	`<OwnerDefinition value="IIQApplicationOwner"/>`
	
	Can provide either the string name of owning Identity or the Identity object.
	
	As with Field value, OwnerDefinition value can also be expressed as a nested element. It can be a string Identity name or an Identity object:
	
	`<OwnerDefinition>`
	` <Value>`
	` <String>Walter.Henderson</String>`
	` </Value>`
	`</OwnerDefinition>`
	
	Three special names exist that are translated by **IdentityIQ** into the appropriate Identity so an OwnerDefinition script is not required for them:
	- **IIQ**ParentOwner - owner of the role or application containing the provisioning policy form
	- **IIQ**ApplicationOwner - owner of the application associated with the provisioning policy form
	- **IIQ**RoleOwner - owner of the role containing the provisioning policy form
	
	`<OwnerDefinition value=""/>` assigns and presents the field to

Nested Elements within Field Elements	Description
	the access requester. The user interface offers these options for setting field owners: **Requester** (sets OwnerDefinition to ""), **Application Owner** (sets to "IIQApplicationOwner"), **Role Owner** on Role Provisioning Policies(sets to "IIQRoleOwner"), and **Rule** and **Script** (save as OwnerDefinition with nested RuleRef or Script, as shown above).

AppDependency	Applies only to application provisioning policies as part of the ordered provisioning function; sets the value for a field based on the value of an attribute on another application on which it is dependent
	
	`<Field displayName="Login ID" name="login" type-e="string">`
	
	`<AppDependency applicationName="LDAP" schem-aAttributeName="employeeNumber"/>`
	
	`</Field>`
	
	This can only be specified when the application has dependencies declared and can only reference attributes on an application on which the application is dependent. The user interface option for field value named **Dependency** creates this element in the field definition.

Working with the Form Editor

The Form Editor provides a graphical user interface enabling you to create and edit forms without having to edit the xml directly.

The Form Editor contains the following sections:

- Detail View — detailed information about the selected form
- Expandable Tree View — provides an ordered, hierarchical view of the form components
- Edit Options — the available attributes for the selected form item

Detail View

This section displays the detail information about the selected form on clicking the **Details** button. The following table lists displayed attributes for the respective Form Type:

Form Type	Attributes
Application Provisioning Policy Form	Title, Subtitle, Wizard, Owner
Role Provisioning Policy Form	Title, Subtitle, Wizard, Application, Owner
Workflow Form	Title, Subtitle, Wizard

Expandable Tree

The expandable tree section provides an ordered and hierarchical view of the form components.

The tree section can be subdivided into the following components:

Action buttons

Buttons for following actions:

- **Add Section** — adds section to the expandable tree view
- **Add Button** — adds button to the bottom of the expandable tree view. The **Add Button** is applicable to Workflow Approval Forms.
    - **Preview Form** — displays the form layout for all included form components in editor. Helps to preview the form while developing a form to see how it renders on actual operations.

Components in the tree view

These are the different components of the tree panel:

- **Section** — Multiple sections can be added to the tree panel through the **Add Section** button. Using **+** icon Fields and Row with Columns can be added. The Section item can be expanded or collapsed by clicking on them.

- **Add Field** — Fields can be added under the Section.
- **Add Row with Columns** — Rows with a maximum of four columns can be added under the Section using the **Choose how many columns in this row** drop down list under the **Edit Options** section.

> When using Rows, the columns attribute of Section and columnSpan attribute of Field would be calculated by Form Editor and existing values would be overwritten.

- **Button** — All the defined Buttons are added at the end in the tree panel.

Reordering Form Components

Form components can be reordered using drag/drop feature in the following way:

- Sections — sections can be reordered. Sections cannot have sections within them.
- Rows — rows can be reordered within the Section or dragged and dropped into any Section.
- Fields — fields can be reordered within Rows or dragged and dropped into any Section.
- Buttons — can be reordered only within Buttons.

> For inappropriate moves of the form components the not allowed icon is displayed.

Edit Options

The Edit Options section on the Editor page displays the attributes that must be modified for the respective actions.

Click on the **Apply** button after the attributes are modified.

Section

Attributes	Description
Basic	
Name	Internal name for section.
Label	Label of Section determines Section text on Edit page. Labels can be specified as text, message catalog keys, or variables (specified with $(variableName) notation).
Subtitle	Section subtitle as description (displayed at the top of the section, above all fields in the section).
Settings	
Hidden	Boolean that, when true, prevents the field from being displayed on the form.
Read Only	Section properties are read only.
Hide Nulls	When set to true, hides fields within the section which have a null value.

Fields and Rows

Attributes	Description
Settings	
Name	Name for the field that can be referenced in code as the variable name in which the fields value is stored
Display Name	Label for the field; can be text or a message key.
Help Text	The text that appears when hovering the mouse over the help icon.
Type	Select the type of field from the drop down list. Select from the following: Boolean — true or false values field. Date — calendar date field. Integer — only numerical values field. Long — similar to integer but is used for large numerical values. Identity — specific identity in IdentityIQ field. Secret — hidden text field. String — text field Application — list of existing application Role — existing type of bundles
Type Settings	
Multi-Valued	Enable this to have more than one selectable value in this field of the generated form.
Refresh On Change	Boolean that, when true, causes the form to refresh when the fields data value changes, running any rules or scripts that run on form reload.
Authoritative	Enable to have the field value override the current value rather than merge with it. Applicable only for multivalued attributes.
Required	Boolean indicating whether a value is mandatory for the field; required="true" marks field with * on form to indicate required and prevents form submission without a value for the field.
Read Only	Determine how the read only value is derived: True — value based on the selection from the drop down list Rule — value is based on a specified rule Script — value is determined by the execution of a script
Hidden	Boolean that, when true, prevents the field from being displayed on the form. True — value based on the selection from the drop down list

Attributes	Description
	Rule — value is based on a specified rule Script — value is determined by the execution of a script
owner	The owner of the field/row. This is determined by selecting from the following: None — no owner is assigned to this field/row Requester — sets the Owner to this field/row Rule — use a rule to determine the owner of this field/row Script — use a script to determine the owner of this field/row
Value Settings	
Value	Sets the default/initial value for the field/row. This can be overwritten on the form if the field/row is not marked as Read Only. Select None, Value, Rule or Script option.
Allowed Values	Specifies a set of values from which the user can select to assign the field value. Automatically created for fields with an allowed values property set to Value (with a list of values specified) through user interface specification. Select Value, Rule or Script option.
Validation	Ability to specify a script or rule for validating the user's value by selecting None, Rule or Script.

Button

Attributes	Description
Settings	
Action	Action determines what the system does in response to clicking the associated button. Select one from the drop down list: Next — used in the Transition logic to advance the workflow to the next step Back — used in the Transition logic to return to a previous step or any other appropriate action for a rejection. Refresh — save the entered form data and regenerate the form; not a state transition just a redisplay of the form
Read Only	Determines whether to show a button or not on the form renderer.
Skip Validation	Determines if client-side required item validation is necessary based on the button clicked by the user. Validation is required if the button is not configured to skip validation, the action is NEXT and there are required items.
Label	Determines the text displayed on the button.
Parameter	Action-parameter of the button.
Value	Action-parameter value.

Form Examples

This section contains examples of XML specifications for the various types of forms.

Application and Role Provisioning Policies and Workflow Forms can all be created through the user interface, though some advanced features might require XML editing to implement. All form types are recorded as XML objects that can be edited through the debug pages as needed. This section reviews the form types in their XML format and shows how they are rendered as a form in the user interface based on that XML definition.

Application and Role Provisioning Policy

Application provisioning policies are specified as `<Form>` within the `<Application>` definition. Role provisioning polices are `<Form>` within the `<Bundle>` definition. Applications might have more than one provisioning policy form - one for (account) creation, update, and delete provisioning activities plus additional policies for group creation and update. Roles might only have one for role assignment to an Identity.

This sample `<Form>` definition provides examples of fields slotted into separate sections, assigned to different owners by value or by script, with an permitted values set, and with a validation script. Application provisioning policies are specified within a `<Forms>` element that wraps all of the specified provisioning policy forms together.

```
<?xml version='1.0' encoding='UTF-8'?><!DOCTYPE Form PUBLIC "spt.dtd" "spt.dtd">
<Form name="New Acct Policy" type="Create">
  <Field displayName="Name" name="name" required="true" reviewRequired="true" type-
e="string">
    <OwnerDefinition value="IIQApplicationOwner"/>
  </Field>
  <Field displayName="Phone" name="phone" required="true" section="Extra Info" type-
e="string">
    <OwnerDefinition value="IIQApplicationOwner"/>
  </Field>
  <Field displayName="Office Number" name="off_no" required="true" section="" type-
e="integer">
    <OwnerDefinition>
      <Script>
        <Source>return identity.getManager();</Source>
      </Script>
    </OwnerDefinition>
    <ValidationScript>
      <Source>
   try {
int number=Integer.parseInt(value);
if (number &lt; 100) {
return "Office numbers are all 100 or greater.";
} else{
return null;
}
} catch (NumberFormatException e) {
return "Non-numeric value provided; must be numeric.";
}
</Source>
    </ValidationScript>
  </Field>
  <Field displayName="Region" name="region" required="true" type="string">
```

```
    <AllowedValues>
      <String>Americas</String>
      <String>EMEA</String>
      <String>APAC</String>
    </AllowedValues>
  </Field>
```

Application Provisioning Policies can render on multiple forms, depending on the field Owners. Multiple provisioning policy forms can be combined into one form if a request spans multiple applications or roles that each need to gather additional data from the same user.

Identity Provisioning Policy

The XML below creates an identity provisioning policy which implements many of the available form options, including:

The form includes multiple field types (: string, object, and secret -. Secret hides enteredthe text). as it is entered. Object fields are rendered as drop-down list boxes pre-populated with all available items of that type.

- Multi-column configurations
- Multi-column spans for some fields
- Allowed values lists
- Tool tip help prompts
- Field validation (runs when user clicks Submit)
- Filter on object lists for example, show only Manager Identities in Manager drop down list
- Conditional display of sections based on entered field values
- Population of fields based on values entered in other fields

The form includes multiple field types: string, object and secret. Secret hides the text as it is entered. Object fields are rendered as drop-down list boxes pre-populated with all available items of that type.

```
     <?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Form PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Form name="Identity Create Policy" type="CreateIdentity">
  <Description>This is the provisioning policy used when creating a new identity thru
LCM.</Description>
  <Section columns="2">
    <Field displayName="First Name" name="firstname" required="true" reviewRe-
quired="true" type="string"/>
    <Field displayName="Last Name" name="lastname" postBack="true" required="true" type-
e="string"/>
    <Field columnSpan="2" displayName="Username" dynamic="true" helpKey="cube name" name-
e="name" required="true" type="string">
      <Script>
        <Source>
           if ((null != firstname) &amp;amp;&amp;amp; (null != lastname)) {
               return (firstname + "." + lastname);
           }
```

```
            return null;
          </Source>
        </Script>
        <ValidationScript>
          <Source>
              // validation variable comes in as "value"; messages value returned
              // is displayed on screen below field on validation; success should return
              // empty messages list
              import sailpoint.tools.Message;
              import sailpoint.object.Identity;

              List messages = new ArrayList();

              Identity existing = (Identity)context.getObjectByName(Identity.class,value);
              if (existing == null) {
                 // No Identity found with that name, so return empty messages -
                 // validation successful
                 return messages;
              } else {
                 Message msg = new Message();
                 msg.setKey("Username: " + value + " already exists. Modify this name to
make it unique.");
                 messages.add(msg);
                 return messages;
              }
          </Source>
        </ValidationScript>
      </Field>
      <Field displayName="Password" name="password" reviewRequired="true" type="secret"/>
      <Field displayName="Password Confirmation" name="passwordConfirm" reviewRe-
quired="true" type="secret"/>
      <Field displayName="Employment Type" displayType="combobox" name="status" postBack-
k="true" type="string">
        <AllowedValues>
          <String>Employee</String>
          <String>Contractor</String>
        </AllowedValues>
      </Field>
    </Section>
    <Section label="Employee Only Fields">
      <Attributes>
        <Map>
          <entry key="hidden">
            <value>
              <Script>
                <Source>
                   if ("Employee".equals(status)) {
                       return false;
                   } else {
                       return true;
                   }
                  </Source>
              </Script>
            </value>
          </entry>
```

```
      </Map>
    </Attributes>
    <Field displayName="Manager" filterString="managerStatus == true" name="manager" type-
e="sailpoint.object.Identity"/>
    <Field displayName="att_email" dynamic="true" name="email" reviewRequired="true" sec-
tion="" type="string">
      <Script>
        <Source>
                if (("Employee".equals(status)) &amp;&amp; (null != firstname) &amp;&amp;
(null != lastname)) {
                    return (firstname + "." + lastname + "@demoexample.com");
                }
                return null;
            </Source>
      </Script>
    </Field>
    <Field displayName="Location" name="location" reviewRequired="true" type="string"
value="Austin">
      <AllowedValues>
        <String>Austin</String>
        <String>Brazil</String>
        <String>Munich</String>
        <String>London</String>
        <String>Brussels</String>
        <String>San Jose</String>
        <String>Chicago</String>
        <String>Taipei</String>
        <String>Tokyo</String>
      </AllowedValues>
    </Field>
  </Section>
</Form>
```

Workflow Form

This example XML creates a custom form that displays the Identity's name and asks the user to select a region to which the Identity should be assigned. It demonstrates use of an AllowedValuesDefinition and a ValidationScript as well as Sections of all three types, text, datatable, and default. This form is embedded in the Workflow XML, as it would be if the form were created through the Business Process Editor **Add Form** option. The form could alternatively be created as a standalone form object and referenced as an argument to the approval, as described in Workflow Forms.

```
<Step name="Need Region" posX="359" posY="182">
  <Approval name="Need Region" owner="ref:launcher" return="region"
          send="identityName">
    <Arg name="workItemDescription"
           value="string:Fill in Region for $(identityName)"/>
    <Form>
     <Attributes>
        <Map>
          <entry key="pageTitle" value="Get Region"/>
          <entry key="title" value="Need Region for Identity"/>
        </Map>
     </Attributes>
```

```
     <Button action="back" label="Abort"/>
     <Button action="next" label="Submit"/>
     <Button action="cancel" label="Return Item to Inbox"/>

     <Section name='userInstructions' type='text'>
        <Field value="Employees must be assigned to a region.  Please provide the correct
region for this employee."
/>
     </Section>

     <Section type="datatable">
       <Field displayName="Employee Name" name="identityName"/>
     </Section>

     <Section name="Edit These Fields">
        <Field displayName="Region Value" name="region" required="true"
          type="String">
         <AllowedValuesDefinition>
           <Script>
             <Source>
                  import java.util.ArrayList;
            import sailpoint.api.*;
            import sailpoint.object.*;

            List regions = new ArrayList();
            QueryOptions qo = new QueryOptions();

            qo.setDistinct(true);
            qo.addOrdering("region", true);

            List props = new ArrayList();
            props.add("region");

            Iterator result = context.search(Identity.class, qo, props);
            while (result.hasNext()) {
              Object [] record = result.next();
            String region= (String) record[0];
            regions.add(region);
            }
            return regions;
             </Source>
           </Script>
         </AllowedValuesDefinition>
         <ValidationScript>
             <Source>
            // validation variable comes in as "value"
              import sailpoint.tools.Message;
               List messages = new ArrayList();
               if(value.length() &lt; 6) {
                  Message msg = new Message();
                  msg.setKey("New region must be at least 6 characters.");
                  messages.add(msg);
               }
               return messages;
```

```
            </Source>
          </ValidationScript>
        </Field>
      </Section>
    </Form>
  </Approval>
</Step>
```

Report Forms

Report forms are used to display report-specific filters to the user in the Edit Report window. The form must be created as an independent Form object and referenced from the report definition in a `<ReportForm>` element.

At run-time, the form is combined with the Report Form Skeleton, which defines the Standard Properties and Report Layout pages. Each section named in the form is created as its own Report Properties page, displayed between the Standard Properties and Report Layout pages. The page name, shown in the **Sections** list and at the top of the form, is specified as the Section's label attribute.

```
   <Form name="Uncorrelated Accounts Report Custom Fields">
      <Section label="Uncorrelated Accounts Parameters" name="customProperties">
<Field displayName="report_input_correlated_apps" filterString="logical==false &amp;&amp;
authoritative==false" helpKey="rept_input_uncorrelated_ident_report_correlated_apps"
name="correlatedApps" type="Application" value="ref:correlatedApps"/>
      </Section>
    </Form>
```

An example of a simple report form is shown below. It contains only one section, formatted in two columns with several datatypes represented (dates, objects, and boolean). The displayName and helpKey values on this report are localizable message keys. The values are all pulled from the TaskDefinition's input arguments, if any are provided there, to set the fields' default values.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Form PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Form created="1346776069392" id="4028460239921ba40139921bf510019a" mod-
ified="1346776080142"
name="Application Owner Access Review Report Form">
  <Section columns="2" label="rept_cert_custom_section_title" name="customProperties">
    <Field displayName="rept_cert_field_create_start" helpKey="rept_cert_help_create_
start"
name="createStartDate" type="date" value="ref:createStartDate"/>
    <Field displayName="rept_cert_field_create_end" helpKey="rept_cert_help_create_end"
name="createEndDate" type="date" value="ref:createEndDate"/>
    <Field displayName="rept_cert_field_signed_start" helpKey="rept_cert_help_signed_
start"
name="signedStartDate" type="date" value="ref:signedStartDate"/>
    <Field displayName="rept_cert_field_signed_end" helpKey="rept_cert_help_signed_end"
name="signedEndDate" type="date" value="ref:signedEndDate"/>
    <Field displayName="rept_cert_field_due_start" helpKey="rept_cert_help_due_start"
name="dueStartDate"
```

```
type="date" value="ref:dueStartDate"/>
    <Field displayName="rept_cert_field_due_end" helpKey="rept_cert_help_due_end" name-
e="dueEndDate"
type="date" value="ref:dueEndDate"/>
    <Field displayName="rept_cert_field_apps" helpKey="rept_cert_help_apps" multi="true"
name="applications" type="Application" value="ref:applications"/>
    <Field displayName="rept_cert_field_tags" helpKey="rept_cert_help_tags" multi="true"
name="tags"
type="Tag" value="ref:tags"/>
    <Field displayName="rept_cert_field_cert_group" helpKey="rept_cert_help_cert_group"
multi="true"
name="certificationGroups" type="CertificationGroup" value="ref:certificationGroups"/>
    <Field displayName="rept_cert_field_show_exclusions" helpKey="rept_cert_help_show_
exclusions"
name="exclusions" type="boolean" value="ref:exclusions"/>
  </Section>
</Form>
```

This form is rendered as shown in the **Report Properties** section of the **Edit Report** window.

In report forms, sections can be created without Field definitions, allowing the report's taskDefinition's initialization rule/script to create the form fields. Several of the standard reports, for example, use an initialization rule to generate a pages of Application and/or Identity attribute filters based on the installation's system data, the defined standard and extended attributes, so the report forms themselves are defined with empty sections. The Privileged Access Report form provides an example of a dynamically built form.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Form PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Form created="1346776069392" id="4028460239921ba40139921bf510019a" mod-
ified="1346776080142"
name="Application Owner Access Review Report Form">
  <Section columns="2" label="rept_cert_custom_section_title" name="customProperties">
    <Field displayName="rept_cert_field_create_start" helpKey="rept_cert_help_create_
start"
name="createStartDate" type="date" value="ref:createStartDate"/>
    <Field displayName="rept_cert_field_create_end" helpKey="rept_cert_help_create_end"
name="createEndDate" type="date" value="ref:createEndDate"/>
    <Field displayName="rept_cert_field_signed_start" helpKey="rept_cert_help_signed_
start"
name="signedStartDate" type="date" value="ref:signedStartDate"/>
    <Field displayName="rept_cert_field_signed_end" helpKey="rept_cert_help_signed_end"
name="signedEndDate" type="date" value="ref:signedEndDate"/>
    <Field displayName="rept_cert_field_due_start" helpKey="rept_cert_help_due_start"
name="dueStartDate"
type="date" value="ref:dueStartDate"/>
    <Field displayName="rept_cert_field_due_end" helpKey="rept_cert_help_due_end" name-
e="dueEndDate"
type="date" value="ref:dueEndDate"/>
    <Field displayName="rept_cert_field_apps" helpKey="rept_cert_help_apps" multi="true"
name="applications" type="Application" value="ref:applications"/>
    <Field displayName="rept_cert_field_tags" helpKey="rept_cert_help_tags" multi="true"
name="tags"
type="Tag" value="ref:tags"/>
```

```
    <Field displayName="rept_cert_field_cert_group" helpKey="rept_cert_help_cert_group"
multi="true"
name="certificationGroups" type="CertificationGroup" value="ref:certificationGroups"/>
    <Field displayName="rept_cert_field_show_exclusions" helpKey="rept_cert_help_show_
exclusions"
name="exclusions" type="boolean" value="ref:exclusions"/>
  </Section>
</Form>
```

Form Models

Form models are used to simplify that process of passing values between the workflow variables and the form. Form models enable the specification of a Map through which a set of variables can be handed to the form by the workflow. The model is defined in the workflow (or pre-defined model is used), enabling the workflow and form to pass a collection of variables at one time through the specified model. The form renderer is set up to use the model so form fields can name the desired attribute directly without having to reference the model name as well.

Since actions in workflows often center around Identities, a map for the identity object, called IdentityModel, is pre-built in IdentityIQ. A workflow library method - getIdentityModel - can be called by a workflow step to create an IdentityModel map to use in a subsequent step that renders a form. To create an empty map, call this method with no arguments. To pre-populate the map with an identity's current values, specify an identity name or ID as an argument to the step. This method is used with no arguments in the new self-service registration workflow to prepare to create a new identity from the data the user enters on the form.

For an example of the simplification offered by a form model: A workflow form needs to display and permit the user to edit 10 identity attributes. Without form binding, all 10 would have to be defined as individual business process variables and all 10 would have to be sent to and returned from the approval in the workflow. The form would also require all 10 to be defined as form arguments. With a model, the whole Identity can be automatically stored in a single business process variable with a single method call, only one variable (identityModel) must be passed to and returned from the form, and no form arguments need to be defined at all.

Use these steps to use the IdentityModel in a workflow form:

1. Go to **Setup > Business Process > Process Variables** tab.
2. Define a process variable in the workflow (identityModel).
3. In an early step in the workflow, initialize and populate the identityModel by calling the getIdentityModel method in the IdentityLibrary workflow library. Specify the identityModel process variable as the Result Variable for that step.

   To pass an identity name or ID to the method, specify it as an argument to the step (identityName or identityId).
4. In the approval that contains the form, create an argument called workItemFormBasePath and specify the identityModel process variable as its value. The form base path is understood by the form renderer and is automatically applied to permit the form access to the map fields. This enables the passing of the model to and from the form.
5. Reference the components of the identityModel in the form as though they were passed as individual variables, for example, as "firstname", not as "identityModel.firstname".

> When a base path is specified as a form argument, the form renderer assumes all fields on the form are accessed through that base path, so all attributes to be included in the form or returned from it must be included in the model.

```
<Section>
<Field displayName="user_name" name="name" required="true" type="string"/>
<Field displayName="first_name" name="firstname" required="true" type="string"/>
```

```
<Field displayName="last_name" name="lastname" required="true" type="string"/>
<Field displayName="email" name="email" required="true" type="string"/>…
```

6. (Optional) To provision changes to the identity based on the form, call the buildPlanFromIdentityModel() method in the Identity Library. This examines the versions of the model passed to the form and back from it, identifies differences between them, and creates a provisioning plan to make the required changes.

Refer to the LCM Registration workflow, which ships with IdentityIQ Lifecycle Manager, for a full example of implementing the identityModel.

No other models currently ship with the product, but custom models can be created through some manual coding in the initialization stage.

1. Declare the model variable as a process variable (same as the identityModel), for example appModel.
2. Initialize the model manually, since no library method exists to populate custom models. Instead of a method call in the initialization step, the step executes a script or rule written to populate the desired data into a HashMap that is stored in the custom model variable.
3. Specify the custom model variable as the workItemFormBasePath argument to the workflow's form step.
4. Reference components in the custom model by name in the form. As with identityModel, no reference to the base path should be specified in the form field names.

Identity Model Structure

The IdentityModel map delivered with IdentityIQ contains the following entries:

- all standard Identity attributes and all extended Identity attributes (most as strings; lists when multi-valued)
- detectedRoles (List)
- assignedRoles (List)
- manager (String name, rather than ID
- info map which contains:
- o manager map (includes ID, name, and displayName of Manager Identity)
- lastRefresh, lastLogin, and passwordExpiration dates
- isWorkgroup, managerStatus, correlated, and correlatedOverriden flag values
- assignedScope name and controlsAssignedScope flag value
- transformerOptions (map of primer identityName or identityId used to populate the IdentityModel)
- class (sailpoint.object.Identity)
- transformerClass (sailpoint.transformer.IdentityTransformer)

Accessing Identity Model Attributes

Any identity model attributes can be displayed on a form or set based on data entered in a form field by supplying the model attribute name as the field name.

Access any single-valued attribute at the top level by specifying its name in the field's name attribute:

```
<Field displayName="first_name" name="firstname" type="string"/>
```

To display the contents of a multi-valued extended identity attribute, use the following syntax. Multi-value extended identity attributes are shown in the identityModel as a list of string values.

```
<Field displayName="Cost Centers" multi="true" name="costcenter" type="string"/>
```

Access any nested attribute, for example, those with a map within the map, using dot notation:

```
<Field displayName="Manager ID" name="info.manager.id" type="string"/>
```

> Values in the info map should not be altered through the form, as they will not be updated in the model; they are treated as read-only data that provides supplementary data for the corresponding top-level attribute, and they are automatically refreshed based on updates to that top-level attribute.

Display the contents of an object list in the map, such as assignedRoles, detectedRoles, or workgroups, on a form by creating it as a combo box. Do this by specifying the type as the correct object type and specifying `multi="true"`:

```
<Field displayName="Detected Roles" filterString="type==&apos;it&apos;" multi="true" name-
e="detectedRoles" type="iiq.object.Bundle"/>


<Field displayName="WorkGroups" filterString="workgroup == true" multi="true" name-
e="workgroups" type="Identity"/>
```

The application of a filterString to the workgroups list ensures that only workgroup identities display.

The links list contains a map for each link (account) held by the identity. Access attributes inside that list by referencing the name of the desired link and using dot notation to traverse the map:

```
<Field displayName="App Owner" name="links['HR_Employees'].sys.nativeIdentity" type-
e="string"/>
```

In development and debugging, it can be helpful to examine the identityModel in XML or as a string representation to clearly see its structure. The identityModel is visible in the workflowCase for any workflow where it is used and is printed to stdout if the trace variable is set to true for the workflow. It can be printed as a string from a workflow step with a `System.out.println(identityModel.toString());` statement.

Referencing a Form Model

Form models can be accessed by rules and scripts within workflows or forms within workflows using the `$()` parsing tokens, for example, `$(identityModel.name)`. When variables are referenced with this syntax, the ScriptPreParser expands the short hand path references into the proper `MapUtil.get()` reference. This notation can be used in scripts run from fields, variables, steps, transitions or step actions. The script can explicitly specify the full path to the variable in the model, for example, `$(identityModel.name)`, or it can reference the variable directly when a `modelBasePath` has been defined, for example `$(name)`.

In order to set a basePath in a form, an argument named modelBasePath (defined as Rule.MODEL_BASE_PATH) must be set declaring the path to be used for all the expanded variables in the form. For example, <Arg name='modelBasePath' value='identityModel'/>. The `modelBasePath` does not have to be specified at the top level; for example, it can point to a list or map within the top-level map such as `<Arg name='modelBasePath' value='identityModel.links[AD]'/>` (this is the map representing the user's AD account link).

This `$()` notation can only be used for retrieving values from the model; it cannot be used to set or change values in the model.

Syntax

Use the following syntax rules when writing references within the scripts:

- A dollar sign with parentheses `[$()]` is used as the parsing token to indicate what contents should be expanded. For example, `$(foo.bar)`.
- Double quotes are valid when enclosing spaces within the variable: `$(foo."bar baz")`. However an expansion token within a quoted string is not processed: `"$(not.expanded)"`.
- Brackets can be used within a variable to access elements in a list: `$(foo.bar[baz=bingo].buzz)` `$(foo.bar[baz="path with spaces"].buzz)`
- When the `modelBasePath` is set to a sub-map or list within the model, the forward slash escape character (/) can be used to jump to the root of the basePath. This escape character must be the first character after the expansion token. If `basePath` is set to `'identityModel.links[AD]'` and the desired reference is for `identityModel.firstname` the variable would be written as `$(/firstname)` which would be converted to `IIQ.tools.MapUtil.get(identityModel, "firstname")`. Otherwise `$(firstname)` is converted to `IIQ.tools.MapUtil.get(identityModel, "links[AD].firstname")`.
- If no basePath is set and the variable only contains a single word, no expansion occurs and a warning is written to the log indicating a possible error condition.

Example Syntax

The following are all valid:

No base path:

- `$(foo.bar)` —> `IIQ.tools.MapUtil.get(foo, "bar")`
- `$(foo."bar baz")` —> `IIQ.tools.MapUtil.get(foo, "\"bar baz\"")`
- `$(foo.bar[baz="path with spaces"].buzz)` —> `IIQ.tools.MapUtil.get(foo, "bar[baz=\"path with spaces\"].buzz")`

Base path = ‘foo’

- `$(foo.bar)` —> `IIQ.tools.MapUtil.get(foo, "bar")` (assuming basePath is set to 'foo'. This respects the basePath and does not try to find a "foo" attribute within the "foo" map)
- `$(bar)` —> `IIQ.tools.MapUtil.get(foo, "bar")` (assuming basePath is set to 'foo')

Base path = ‘foo.bar[AD]’

- `$(baz)` —> `IIQ.tools.MapUtil.get(foo, "bar[AD].baz")` (assuming basePath is set to 'foo.bar[AD]')
- `$(/baz)` —> `IIQ.tools.MapUtil.get(foo, "baz")` (assuming basePath is set to 'foo.bar[AD]')