EVIDEN

Identity and Access Management

# DirX Identity

## Customization Guide

Version 8.10.14, Edition March 2026

# Table of Contents

# Preface

This manual provides information how to customize your DirX Identity installation. It consists of the following sections:

- Chapter 1 introduces DirX Identity objects.
- Chapter 2 describes how to customize objects in the Provisioning and Connectivity configuration.
- Chapter 3 describes hoe to customize policies.
- Chapter 4 describes how to customize object references.
- Chapter 5 describes how to customize parameters.
- Chapter 6 describes how to customize the Provisioning tree view.
- Chapter 7 describes how to customize wizards.
- Chapter 8 describes how to customize Provisioning target systems.
- Chapter 9 describes how to customize auditing.
- Chapter 10 describes how to customize DirX Identity program logic.
- Chapter 11 describes how to customize the password manager.
- Chapter 12 describes how to customize certification campaigns.
- Appendix A describes deprecated features that are obsolete and will not be supported in the future.

# DirX Identity Documentation Set

*Version 8.10.14 | Build 1858 | Date 2026-03-26 *

The DirX Identity document set consists of the following manuals:

- *DirX Identity Introduction*. Use this book to obtain a description of DirX Identity architecture and components.
- *DirX Identity Release Notes*. Use this book to understand the features and limitations of the current release. This document is shipped with the DirX Identity installation as the file **release-notes.pdf**.
- *DirX Identity History of Changes*. Use this book to understand the features of previous releases. This document is shipped with the DirX Identity installation as the file **history-of-changes.pdf**.
- *DirX Identity Tutorial*. Use this book to get familiar quickly with your DirX Identity installation.
- *DirX Identity Provisioning Administration Guide*. Use this book to obtain a description of DirX Identity provisioning architecture and components and to understand the basic tasks of DirX Identity provisioning administration using DirX Identity Manager.
- *DirX Identity Connectivity Administration Guide*. Use this book to obtain a description of DirX Identity connectivity architecture and components and to understand the basic tasks of DirX Identity connectivity administration using DirX Identity Manager.
- *DirX Identity User Interfaces Guide*. Use this book to obtain a description of the user interfaces provided with DirX Identity.
- *DirX Identity Application Development Guide*. Use this book to obtain information how to extend DirX Identity and to use the default applications.
- *DirX Identity Customization Guide*. Use this book to customize your DirX Identity environment.
- *DirX Identity Integration Framework*. Use this book to understand the DirX Identity framework and to obtain a description how to extend DirX Identity.
- *DirX Identity Web Center Reference*. Use this book to obtain reference information about the DirX Identity Web Center.
- *DirX Identity Web Center Customization Guide*. Use this book to obtain information how to customize the DirX Identity Web Center.
- *DirX Identity Meta Controller Reference*. Use this book to obtain reference information about the DirX Identity meta controller and its associated command-line programs and files.
- *DirX Identity Connectivity Reference*. Use this book to obtain reference information about the DirX Identity agent programs, scripts, and files.
- *DirX Identity Troubleshooting Guide*. Use this book to track down and solve problems in your DirX Identity installation.
- *DirX Identity Installation Guide*. Use this book to install DirX Identity.

- *DirX Identity Migration Guide*. Use this book to migrate from previous versions.

# Notation Conventions

**Boldface type**
In command syntax, bold words and characters represent commands or keywords that must be entered exactly as shown.

In examples, bold words and characters represent user input.

*Italic type*
In command syntax, italic words and characters represent placeholders for information that you must supply.

[ ]
In command syntax, square braces enclose optional items.

{ }
In command syntax, braces enclose a list from which you must choose one item.

In Tcl syntax, you must actually type in the braces, which will appear in boldface type.

|
In command syntax, the vertical bar separates items in a list of choices.

...
In command syntax, ellipses indicate that the previous item can be repeated.

*userID_home_directory*
The exact name of the home directory. The default home directory is the home directory of the specified UNIX user, who is logged in on UNIX systems. In this manual, the home pathname is represented by the notation *userID_home_directory*.

*install_path*
The exact name of the root of the directory where DirX Identity programs and files are installed. The default installation directory is *userID_home_directory*/**DirX Identity** on UNIX systems and **C:\Program Files\DirX\Identity** on Windows systems. During installation the installation directory can be specified. In this manual, the installation-specific portion of pathnames is represented by the notation *install_path*.

*dirx_install_path*
The exact name of the root of the directory where DirX programs and files are installed. The default installation directory is *userID_home_directory*/**DirX** on UNIX systems and **C:\Program Files\DirX** on Windows systems. During installation the installation directory can be specified. In this manual, the installation-specific portion of pathname is represented by the notation *dirx_install_path*.

*dxi_java_home*
The exact name of the root directory of the Java environment for DirX Identity. This location is specified while installing the product. For details see the sections "Installation" and "The Java for DirX Identity".

*tmp_path*

The exact name of the tmp directory. The default tmp directory is /tmp on UNIX systems. In this manual, the tmp pathname is represented by the notation *tmp_path*.

*tomcat_install_path*
The exact name of the root of the directory where Apache Tomcat programs and files are installed. This location is defined during product installation.

*mount_point*
The mount point for DVD device (for example, **/cdrom/cdrom0**).

# 1. Customizing DirX Identity

DirX Identity is designed to be highly configurable in its function and in the display of its objects in DirX Identity Manager. To achieve this goal, DirX Identity allows you to customize:

- Objects in the provisioning and connectivity configuration. You can use object descriptions and component descriptions to customize existing objects or to define your own objects.
- Policies. You can define access control for objects that are not yet protected and for your own custom objects.
- Workflow definitions. You can use object references to define how object attributes fill dynamically for DirX Identity workflow definitions.
- Proposal lists, permission parameters, and role parameters. You can use role parameter and proposal parameter objects to customize existing list definitions or to create your own.
- The Provisioning object tree view. You can adapt the tree view to your company's requirements.
- Wizards. The Provisioning and Connectivity wizards are used to hide parts of the complexity of DirX Identity
- Provisioning target systems. You can define target systems and how accounts are created including unique naming and inheritance rules
- Auditing tools, such as query folders and status reports. You can use query folders for data maintenance and reports to check compliance.
- DirX Identity program logic. You can use JavaScript and Java programming to extend the system where simple configuration is not sufficient.
- Password manager. You can customize the password checker and the password generator of the DirX Identity password management component. If you are using the DirX Directory Server as the Identity Store, you can also enable account locking.
- Certification campaigns. You can customize certification campaigns with user hooks.

# 2. Customizing Objects

DirX Identity Provisioning and Connectivity objects are described by text scripts called **object descriptions** that are formatted in Extensible Markup Language (XML). Each object description contains a set of elements that describe:

- The object itself, by listing its properties and their values
- The object mapping, to identify an instance from a given directory entry
- A list of context menu actions that become available by right-mouse-button clicks on the object's tree item
- A list of property pages that present the object's attributes in DirX Identity Manager
- A set of description extensions, currently just the set of attributes to be loaded when the object data are retrieved for the first time
- The object's attributes and their definitions

The sections in this chapter describe:

- The XML elements and XML attributes that comprise an object description (see the section "Object Description Format")
- Tasks specific to customizing Provisioning object descriptions (see the section "Customizing Provisioning Objects")
- Tasks specific to customizing Connectivity object descriptions (see the section "Customizing Connectivity Objects")
- How to create component descriptions to be used in object descriptions (see the section "Creating Component Descriptions")

## 2.1. Object Description Format

An object description consists of the XML elements shown in the following example:

```
<object
     name="dxrUser" ... other object attributes ... >
     <!-- import another object description to inherit its
specifications -->
<import file="storage://DirXmetaRole/DN_of_object_description_entry"
/>

     <!--How to map an object read from the database to an object
description: -->
     <mapping>
          <attribute objectclass="{any}dxrUser"/>
     </mapping>
```

```
     <!--Actions for the context menu -->
<action
...
</action>

     <!--Property page definitions -->
     <propertysheet>
          <propertypage ... />
     </propertysheet>

     <extension>
          <loadAttributes>
               ...
          </loadAttributes>
     </extension>

     <!--Property definitions -->
     <properties>
          <property name="objectclass" ... />
          ...
     </properties>

</object>
```

The remainder of this section provides more information on these elements and how to use them in an object description.

## 2.2. Object Element

The **object** element describes a DirX Identity object like a user or a role. The following table lists the XML attributes that can or must be specified within an **object** element.

*Table 1. Object Element Attributes*

| Attribute | Mandatory or Optional | Description |
|---|---|---|
| name | Mandatory | The unique name of the object description; often referenced as "odName". |
| class | Mandatory | The Java class name that represents this object in DirX Identity Manager and Agent. Do not change this value. |
| namingattribute | Optional | The property (LDAP attribute) that defines the object name in the DirX Identity store (the RDN). |

| Attribute | Mandatory or Optional | Description |
|---|---|---|
| displayattribute | Optional | The property (LDAP attribute) that is used when the object is displayed in the Manager tree. Default: cn. |
| label | Optional | The string that is used as the display name for the object class. |
| icon | Mandatory | The name of the icon to be displayed with the object in the tree. |
| candelete | Optional | Whether the object can be deleted from the store. Default: true. |
| cancopy | Optional | Whether the object can be copied to another container in the store. Default: true. |
| canrename | Optional | Whether the object can be renamed. Default: true. |
| canmove | Optional | Whether the object can be moved to another container in the store.<br><br>Default: true, if cancopy = true and candelete = true. |
| haschildren | Optional | Whether the object is a folder below which entries can be created. Default: false. |
| parents | Mandatory | The objects (names of object descriptions = odNames) below which this object can be created. DirX Identity evaluates this attribute when you select to create a new object. It scans all object descriptions for parents attributes that reference its own object description names. Only these objects can be created. |
| helpcontext | Optional | The unique reference to the corresponding online help page. |
| accesscontrol | Optional | Set this flag to 'true' if you want to enable access control for this object type. If the flag is to "false" or is not available, access control is not enabled.<br><br>Note: if you enable access control, you must set up the corresponding access policies. |

## 2.3. Body Element

The **body** element is used in imported object descriptions as the root element instead of the **object** element. It can contain the same sub-elements as the **object** element, but it does not contain its XML attributes, which must be specified in the **definition** element.

## 2.4. Definition Element

The **definition** element is contained in a **body** element. The **definition** element allows the

same XML attributes as the **object** element, but does not contain any sub-elements. See the **object** element for a description of XML attributes.

## 2.5. Mapping Element

The **mapping** element is a mandatory inner element of an object description. DirX Identity uses this element to map an object read from the DirX Identity store to an object description. In general, a mapping element uses the object class as the identifier, as shown in this sample for the user object:

```
<mapping>
     <attribute objectclass="{any}dxrContainer"/>
     <attribute dxrType="dxrTSContainer"/>
</mapping>
```

The **{any}** element as the **objectclass** attribute's value indicates that the attribute can take any value, but the explicit **dxrContainer** value indicates that the object's object class must take also this value.However, some objects with the same object class must be handled differently; for example, folders in the roles view vs. folders in the target systems views.Consequently, the mapping element uses the **dxrType** attribute to distinguish between the different types of object, as shown in the example.

You can combine the **{any}** element with the **~** (not) and the **{none}** element.Every relevant object has a type properties set.In the following example you want to define a special object description for all objects with type internal.All other objects should use a general ObjectDescription:

The mapping element part for internal is the following:

```
<mapping>
    <attribute objectclass="{any}myObjectClass"/>
    <attribute dxrType="internal"/>
</mapping>
```

The mapping element part for general is the following:

```
<mapping>
    <attribute objectclass="{any}myObjectClass"/>
    <attribute dxrType="~internal"/>
</mapping>
```

In the following example all objects with a filled type property **not equal** (the **~** sign) to internal map to the general object description. Objects with an **empty** type property do not

match with both mappings. The **{none}** element defines that it should also match for empty type properties:

```
     <mapping>
         <attribute objectclass="{any}myObjectClass"/>
         <attribute dxrType="{none},~internal"/>
    </mapping>
```

Different target systems usually need different naming rules for their accounts and groups: both attributes and default values will differ. As a result, target system-specific object descriptions are necessary. These object descriptions are held below the corresponding target system folder in the DirX Identity store. When DirX Identity needs to associate an object description with an account or group (either when reading an existing one or when creating a new one) it uses the distinguished name (DN) of the object to select the correct description from the list of account or group object descriptions.

Consequently, the mapping element of an account or group object description must contain the DN of the corresponding target system folder, as follows:

- In the mapping element for the object description, you supply an XML description for the DN that contains the tag ***,$(./../../..)**. For example:

```
<mapping>
     <attribute dn="*, $(./../../..)"/>
     <attribute objectclass="{any} dxrTargetSystemAccount" />
</mapping>
```

The object description is placed below the target system folder:

*TS_folder*/Configuration/Object Descriptions*/description_object*

- When DirX Identity reads this object description, it replaces the variable **$(./../../..)** with the DN of the target system folder (the entry three layers above it). If the DN of the selected account or group object contains this DN string, DirX Identity uses it as the object description.

The same substitution mechanism is used for the property page and other descriptions.

## 2.6. Action Element

The **action** element specifies the actions that are possible in the context menu when an object is selected in a DirX Identity Manager tree view. The action is identified by the **class** attribute, which specifies a DirX Identity-supplied Java class. For example:

```
<action
```

```
     class="siemens.DirXjdiscover.api.actions.NewAction"
     class="siemens.DirXjdiscover.api.actions.DeleteAction"
/>
```

The following table lists a subset of the actions that you can add or remove from an object description:

*Table 2. DirX Identity Actions*

| Action | Class | Description |
|---|---|---|
| New | siemens.DirXjdiscover.api.actions.NewAction | Allows the user to enter a new object below the object. This action is only applicable to folders. The permissible object classes are calculated using the "parents" attribute of all object descriptions. |
| Delete | siemens.DirXjdiscover.api.actions.DeleteAction | Allows the user to delete the object or folder. |
| Enable / disable | siemens.dxr.manager.actions.ActionEnableDisable | Allows the user to enable and disable the object; this action applies only to accounts, and must not be changed. |
| Rename | siemens.DirXjdiscover.api.actions.RenameAction | Allows the user to change the naming attribute for the object (modifyDN). This action is permitted for all objects except accounts and groups. |
| Report | siemens.dxr.manager.actions.ActionReport | Allows the user to produce a report. |

## 2.7. Properties and Property Elements

The **properties** element in an object description specifies the attributes of an entry in the

Identity store that DirX Identity is to read or write. Each LDAP attribute of the entry is specified by a **property** element within the **properties** element. For each LDAP attribute of the entry, you must define its LDAP attribute type, its data type (String or other), whether or not it is a multi-valued attribute and its default values.

The following table describes the XML attributes that you can or must specify for each **property** element.

*Table 3. Property Element Attributes*

| Attribute | Mandatory / optional | Description |
|---|---|---|
| name | Mandatory | The property's LDAP attribute name. For example, name="givenName". This value must be consistent with the LDAP attribute name in the directory schema.<br><br>In the Connectivity configuration, you can set up specific attributes within propertyPage elements. In this case, you can use the shortcut "_SP" for "dxmSpecificAttributes". Note that it is not possible to use this shortcut here in the name definition. |
| label | Mandatory | The display name of the property when viewed with Manager. For example, label="Given Name". |
| type | Mandatory | The Java type that specifies how the property is controlled and displayed. For a description of the available types and editors see the chapter "Types and Editors". |
| mandatory | Mandatory | Whether or not the property must have a value. For example, mandatory="true". You can specify a value with the **defaultvalue** attribute. |
| readonly | Optional | Whether or not the value of the property is only displayed and cannot be changed. For example, readonly="true". |
| multivalue | Optional | Whether or not the property has multiple values. For example, multivalue="true". Several editors are available to handle this type of property: siemens.dxr.manager.controls.MetaRoleJnbComboBox |
| visible | Optional | Whether or not the property is visible in Manager. For example: visible="false" (the default is "true"). An invisible property exists in the data model and is set from internal routines but is not displayed in Manager. |
| defaultvalue | Optional | The default value to be used for the property when the object is created. For example, defaultvalue="false" or defaultvalue="\{dxrUser,inetOrgPerson,person,top}" |

| Attribute | Mandatory / optional | Description |
|---|---|---|
| subsetdelimiter | Optional | The subset delimiter to use for all **specificAttributes** values. This attribute defines the character that separates the name and the value. For example, subsetdelimiter="/" assumes the values to be in the structure:<br><br>name1/value1<br>name2/value2<br><br>**Warning**: DirX Identity requires this value to be a space character in most values. Do not change it! |
| tags | Optional | A list of valid entries for the property. When of type String, this property permits the definition of a list of valid entries. Only the listed values are available; there is no direct edit. For example, tags="On,Off". You can use the value "(None)" to allow selection of no value (this results in deletion of this attribute). Example: tags="(None),On,Off". In this case, the user can choose between On and Off and no value. Note that fields with tags are marked with a red border if they are not filled with a value (the attribute does not exist). The red border is suppressed when the "(None)" tag value is used in the list. You can also use proposal lists to define the possible values for a property; see the topic "Specifying Proposal Lists in Property Descriptions" for details. |
| master | Optional | For Provisioning objects only, the property's master attribute. The master attribute is similar, but different from the naming rule. A naming rule specifies default values that can subsequently be changed at any time. A mastered property always has the same value as the same attribute in the master entry. See the topic "Using the Master Attribute" for an example. |
| dependsOn | Optional | For Provisioning objects only, one or more properties of the entry on which this property depends. Each time one of the specified properties changes, the rule for generating this property value is re-evaluated. When using this attribute, take care to avoid circular dependencies. |

| Attribute | Mandatory / optional | Description |
|---|---|---|
| uniqueIn | Optional | Usable for Provisioning objects only (accounts and groups). It guarantees that within a specific subtree object identifiers are unique. Use this XML attribute for properties like login names or identifications that must be unique within a domain. The value is interpreted as the root entry of the subtree in which the property value must be unique. Be sure to supply a naming rule (XML or Java script) that allows for alternatives (XML naming rule) or loops (Java script) in case the first proposed value already exists in the namespace. See the template Java scripts and XML naming rules that are used for accounts for examples. |

### 2.7.1. Specifying Default Values for a Property

DirX Identity offers a number of ways to specify default values for a property. Default values are set only when DirX Identity creates the object. The simplest way to specify one or more default values for a property is to use the **defaultvalue** attribute, as shown in this example:

```
<property
     name="l"
     ...
     defaultvalue="München"/>
```

For a multi value property, you can specify, for example:

```
<property name="objectclass" defaultvalue="{dxrAccessPolicy,top}"/>
```

Sometimes this mechanism is not sufficient, especially for the automatic generation of e-mail addresses, account names or even default passwords. For these cases, you can use naming rules to specify default values. You can specify them with XML elements and attributes or as Java scripts.

### 2.7.2. Using Specific Attributes in Property Descriptions

DirX Identity uses the **specificAttributes** construct for group attributes that are used as permission parameters in permission matching rules. (See the *DirX Identity Connectivity Administration Guide* for details) These attributes are stored in the multi-valued group attribute **dxrRPValues** for Provisioning objects and **dxmSpecificAttributes** for Connectivity objects. The format is: *attribute name=attribute value*, for example, l=München. The attribute name must be an attribute that is specified as a permission parameter. Specific attributes are specified as follows:

```
<property name="dxrRPValues" subsetdelimiter="="/>
<property name="dxrRPValues(l)" multivalue="true"
type="java.lang.String"
editor="siemens.dxr.manager.controls.MetaRoleJnbComboBox"/>
```

First, the property **dxrRPValues** is defined with a subsetdelimiter of **=**. Then the property **l** is defined as specific attribute with the term **dxrRPValues(l)**.

### 2.7.3. Specifying Proposal Lists in Property Descriptions

DirX Identity offers several ways to specify a proposal list, which is a list of possible values for an attribute. The simplest way is to use the **tags** attribute of the **property** element to specify the possible values. Here is an example that specifies that the values **De** and **En** are allowed for the **dxrLanguage** property:

```
<property
     name="dxrLanguage"
     label="Language"
     type="java.lang.String"
     tags="De{German / Deutsch},En{English}"
     defaultvalue="De"
/>
```

Use the **tagprovider** element when you want to store the proposal list in the DirX Identity store. The following example directs DirX Identity to take the proposal list for the **l** property from the attribute **dxrProposedValues** of the DirX Identity object with the DN "cn=Location,cn=Proposal Lists,cn=Configuration,cn=PQR-metaRole":

```
<property
     name="l"
     label="Location"
     multivalue="true"
     type="java.lang.String"
     editor="siemens.dxr.manager.controls.MetaRoleJnbComboBox">
     <tagprovider
          class="siemens.dxr.manager.Proposal"
          dn="cn=Location,cn=Proposal
Lists,cn=Configuration,cn=$(rootDN)"
          proposals="dxrProposedValues"/>
</property>
```

Proposal list objects are usually located below the domain configuration subtree in the folder "cn=Proposal Lists,cn=Configuration,cn=*domainRoot*". In this case, you can use the **cn** attribute in the **tagprovider** element instead of the **dn** attribute, as shown in this example:

```
<property
     name="l"
     ...
     <tagprovider
          class="siemens.dxr.manager.Proposal"
          cn="cn=LocationValues"
          proposals="dxrProposedValues"/>
</property>
```

The **cn** attribute specifies the unique common name of the object underneath the domain's configuration folder. DirX Identity searches for this object of object class **dxrProposal** within the domain.

There are no restrictions on where to place proposal lists: below domain configuration (recommended) or below a target system. The proposed values are stored in the attribute identified by the **proposals** tag. In the previous example, the proposed values are stored in the object with the cn "cn=LocationValues" in the multi-valued attribute **dxrProposedValues**.

## 2.8. Property Sheet and Property Page Elements

DirX Identity Manager presents each object using property sheets that consist of one or more tabs called property pages. These elements must be configured in the object description and the property page descriptions.

The **propertysheet** element specifies the entire presentation and consists of one or more **propertypage** elements. The **propertysheet** element has no attributes. Each **propertypage** element defines a visual tab for the object. The following table lists the XML attributes that can or must be specified for a **propertypage** element.

*Table 4. PropertyPage Element Attributes*

| Attribute | Mandatory or Optional? | Description |
|---|---|---|
| name | Mandatory | The internal name of this element (not the display name). For example, name="General". |
| title | Mandatory | The display name of the tab in DirX Identity Manager. For example, title="JobProperties". |
| helpcontext | Optional | The help context that should be shown in the Java Help viewer. For example, helpcontext="Job". |

| Attribute | Mandatory or Optional? | Description |
|---|---|---|
| class | Optional | The Java class that handles this property page. The default is the class **siemens.dxm.gui.components.PropertypageGeneric** ("generic page"), which displays all properties in an easily configurable table view.<br><br>In previous releases of DirX Identity, the class **siemens.DirXjdiscover.api.nodes.customizer.BMLNodePropertyPage** ("BML page") was used. It allowed fine grained layout configuration (see the layout attribute). This feature is deprecated due to the fact that the generic property page mechanism has been enhanced to allow individual layout configuration. |
| layout | Optional | Controls the layout of the page. Two possibilities exist:<br><br>1. For generic pages, you can define a table of properties to be displayed on that page (default: all properties are displayed as a simple list).<br>2. For generic pages, you can also define the sequence of the properties, the grouping into frames with text headers and the list of properties to be displayed in one row. The layout specification consists of:<br><br>• the initial term (mandatory):<br>**properties**:<br>• a list of properties separated with commas:<br>*attribute1*,*attribute2*,...<br>• row collection specifications:<br>{*attribute1*,*attribute2*,...}<br>• group definitions:<br>[**groupname**;_attribute1_,*attribute2*, ...]<br><br>Do not use BML page layout anymore. This feature is deprecated and will not be supported in the future. |

Here is the **propertysheet** definition in the user object description:

```
<propertypage name="UserGeneral"
      helpcontext="usergeneral"
class="siemens.dxr.manager.nodes.customizer.GenericPropertyPage"
      title="General"
```

```
      layout="properties:[General;cn,{givenName,initials},
      {dxrSalutation,title,sn},
      {gender,dayOfBirth},{dxmOprMaster,employeeNumber,dxmGUID},
      description],
      [Categories;employeeType,businessCategory]"/>
```

The following figure shows the resulting user property sheet in the General tab in DirX Identity Manager:

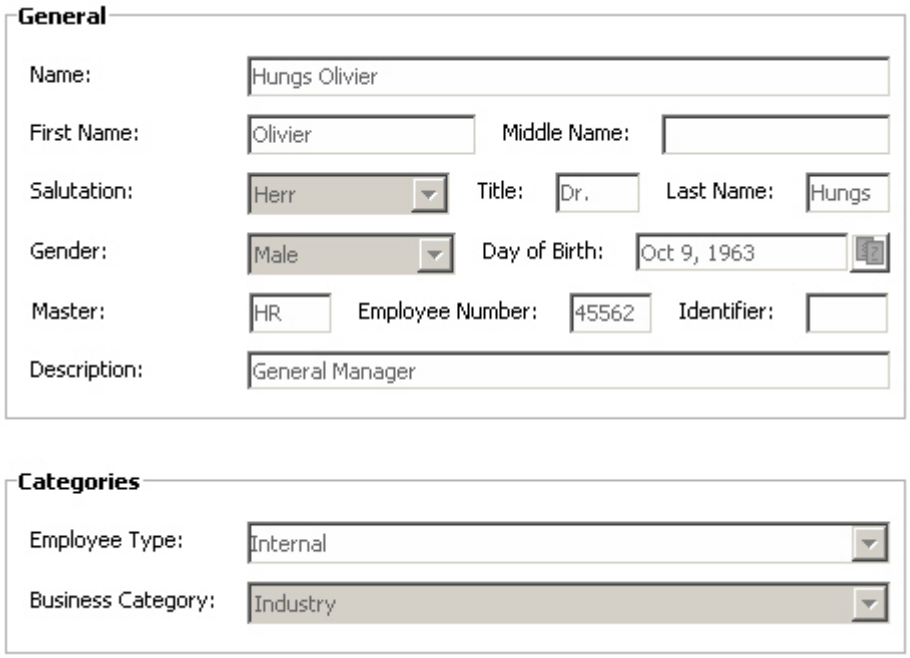



*Figure 1. User Property Sheet*

The first line displays the Name attribute, while the next line shows the First Name and Middle Name attributes in the same line. The third line is an example for three attributes in a row. All of these attributes including Description are grouped and marked with the header General.

Another group (Categories) combines the attributes Employee Type and Business Category.

## 2.9. Import Element

You can use the **import** element to inherit common configurations stored in another object description. DirX Identity uses the **import** element to separate customer extensions from system configuration.

In the extension you can define new properties or property pages or overwrite property descriptions that were already specified in the imported entry. DirX Identity concatenates the imported entry with the current one by adding the current elements to the imported ones. If you specify an element with the same name in both the extension and the imported entry, the extension definition wins, since it is read after the other.

There is one essential difference between "surrounding" and imported object descriptions. The XML parser gets them as one XML document. According to the XML standard, a well-formed XML document may only contain one root element. For object descriptions, this is the object element. Since the surrounding description "contains" the imported entry, only the surrounding, i.e. importing entry may contain the **object** element. Those object descriptions that are imported, use the body element instead of the **object** element. The **body** element may contain the same sub elements as the **object** does. But it does not contain any attributes! They have to be specified in the **definition** element, which does not contain any sub elements.

This is the basic structure of an imported and a surrounding object description:

First the imported description, which is in the entry "cn=TSAccount.xml,cn=Object Descriptions,cn=Windows 2000,cn=TargetSystems,cn=Configuration,cn=HHB-Domain":

```
<body>
<definition class="..." label="..." candelete="..." ... />
<mapping>...</mapping>
<!-- and the other elements, especially propertysheet, properties and
property -->
</body>
```

The following entry imports the previous description:

```
<object name="SvcTSAccount>
<import file="storage://DirXmetaRole/cn=TSAccount.xml,cn=Object
Descriptions,cn=Windows
2000,cn=TargetSystems,cn=Configuration,$(rootDN)?content=dxrObjDesc"
/>
<mapping>...</mapping>
<!-- and other elements, especially properties and property -->
</object>
```

## 2.10. Load Attributes Element

Folder object descriptions can contain the **loadAttributes** element, for example, the **org.xml** description:

```
     <extension>
          <loadAttributes>
               <children name="{ou,sn,givenName,dxrState}" />
          </loadAttributes>
```

```
    </extension>
```

In this example, the **name** attribute in the **children** inner element of the **loadAttributes** element tells DirX Identity which properties to read when objects are loaded from the LDAP directory for the first time. The properties that are specified are the important properties used for displaying an object in a tree or list view or identifying its state. This mechanism helps to improve performance, because DirX Identity does not need to re-read the entries due to missing attributes.

A **children** inner element refers to all child objects of a folder object.

In addition to the **children** inner element, all - real and virtual - property names that refer to a directory entry referenced by the current entry are allowed as inner elements of **loadAttributes**; for example, roles, senior roles, groups, accounts, and so on. Please do not change these constructs.

# 2.11. Types for Properties

You can specify types for properties. This section describes

- Types which are common to both Connectivity and Provisioning configuration.
- Types which are available only for Connectivity.
- Types which are available only for Provisioning.

DirX Identity uses types to handle attributes correctly.

## 2.11.1. Common Types

The following common types are available:

- **java.io.File** - a file name property (includes the path).
  Default editor: siemens.dxm.storage.beans.JnbFile
- **java.lang.Boolean** - a boolean property with values True or False.
  Default editor: siemens.dxm.storage.beans.JnbBoolean
- **java.lang.Integer** - the property is a field that can contain only numbers.
  Default editor: siemens.dxm.storage.beans.JnbIntegerField
- **java.lang.String** - the property is a simple text field.
  Default editor: "siemens.dxm.storage.beans.JnbTextField"
- **siemens.dxm.storage.StorageObject** - a storage object.
  Default editor: siemens.dxm.storage.beans.JnbReference
- **siemens.dxm.util.BooleanInteger** - a boolean property with values 1 or 0.
  Default editor: siemens.dxm.storage.beans.JnbBooleanInteger
- **siemens.dxm.util.GeneralizedTime** - a property that contains both time and date.
  Default editor: siemens.dxm.storage.beans.JnbGeneralizedTime
- **siemens.dxm.util.TextAreaString** - a property that contains text.

- **[B** - a property that contains binary values like pictures or certificates.

### 2.11.2. Connectivity Types

Connectivity types include:

- **siemens.dxm.util.TCLCodeString** - a string that contains Tcl-Code.
  Default editor: siemens.dxm.beans.JnbTCLEditor

### 2.11.3. Provisioning Types

Provisioning types include:

- **siemens.dxm.util.GeneralizedDate** - a date property.
  Default editor: siemens.DirXjdiscover.api.ldap.beans.JnbLdapGeneralizedDate
- **siemens.dxm.util.TimeInterval** - a time interval property.
  Default editor: siemens.dxm.storage.beans.JnbTimeInterval

## 2.12. Editors

You can specify types and editors for properties to handle attribute types correctly. The editor specifies display and handling at the user interface. This section provides information about editors provided with DirX Identity. In addition to the editors described in the following sections there are a lot of internal used editors.

Usually one editor can handle just one value. To handle multi-value attributes there is a specific multi-value editor or you must use the general multi-value editor. (See section "General Multi-Value Editor" for details.)

The following sections describe editors for

- simple property types like integer or boolean values.
- strings.
- date and time.
- references like references to files or distinguished names.
- filters.
- specific components and types like pictures and IP addresses.
- code.

For each editor the types are provided that the editor can handle. For a property the editor is specified in the following format:

**type=**"*data_type*"
[**multivalue=**"{**true**|**false**}"]
[**editor=**"*single-value_editor*"]
[**editorparams**="*editor_parameters*"]
[**multivalueeditor**="*multi-value_editor*"]

where

***type*="data_type"**

Specifies the property type. *data_type* is the name of the type, for example **java.lang.Integer**. (See "Types for Properties" for valid *data_type* names.)

***multivalue="{true|false}"***

Specifies whether the property type is a multi-value (**true**) or a single-value (**false**). This statement is optional. The default value is **false**.

***editor*="single-value_editor"**

Specifies the editor. This statement is optional. It can be omitted if a default editor is specified for the property type. (See "Default Editors" for details.) *single-value_editor* is the name of the single-value editor, for example **siemens.dxm.storage.beans.JnbIntegerField**. (See the following sections for valid *single_value_editor* names.)

***editorparams*="editor_parameters"**

Specifies the editor parameters. This statement is optional and can be omitted if the editor is used with default parameters or without parameters. *editor_parameters* are editor specific. Separate multiple editor parameters with **;**. The syntax for *editor_parameters* is as follows:

*parameter_name_1*=***parameter_value_1***[;*parameter_name_2*=*parameter_value_2* ...]

where

*parameter_name* is the name of the parameter, for example **visibleparts** and

*parameter_value* is the parameter value, for example **dhms**.

Here is an example for *editor_parameters*:

```
visibleparts=dhms;partsperrow=4
```

***multivalueeditor*="multi-value_editor"**

Specifies the multi-value editor. This statement is optional and can be omitted for single-value property types (**multivalue="false"**). *multi_value_editor* is either the name of the general multi-value editor (see "General Mult-Value Editor" for details) if no specific multi-value editor is available or the name of the specific multi-value editor (see the following sections for details).

## 2.12.1. Default Editors

If an editor is associated to a property type as the default editor it is not necessary to specify this editor. Nevertheless you can use another editor for a type for special handling.

For Provisioning, you can find the default editor assignment for a type in the file **Domain Configuration** → **Object Descriptions** → **Editors.xml**.

For Connectivity, you can find the default editor assignment for a type in the **Data View**. Browse to the entry *
Connectivity dxmC=DirXmetahub* → **dxmC=Configuration** → **dxmC=GUI** → **dxmC=system** → **dxmC=core** → **dxmC=editors.xml**.
Click the **All Attributes** tab and press the **Export the binary value** button of the dxmContent attribute. Specify the file name **editors.xml** and open the exported file in the Internet Explorer.

### 2.12.2. Using Variables

It is possible to use variables in editor parameters. Specify variables in the format:

**$(***variable_name***)**

The variables are replaced at run-time.

Example:

**$(rootDN)**

At run-time the variable is replaced with the current value of **$(rootDN)**, for example **cn=My-Company**.

### 2.12.3. Using Expressions

It is possible to use expressions in editors. Specify expressions in the format:

**${***expression***}**

Expressions are replaced at run-time. (See "Using Variable Substitution" in the *DirX Identity Application Development Guide* for details.)

Example:

```
${workflow.subject.sn}
```

At run-time the variable is replaced with the current value of **${workflow.subject.sn}**, for example **Smith**.

### 2.12.4. Nationalization Support in Editors

Some of the editors described in the following sections support nationalization. The context-sensitive menu of these editors provides the following additional operations:

- **Insert a Message...** - inserts a message with an absolute path, for example **#{Common Text.DoNotAnswer}**.
- **Insert a Message relative...** - inserts a message with a relative path to the selected object, for example **#{../../_Nationalization.UserCreationFailed_body}**.
- **Resolve a Message ...** - displays the resolved message in the default language of the domain.

Insert message operations can be performed in edit-mode only. The insert operation opens a **Select** dialog:

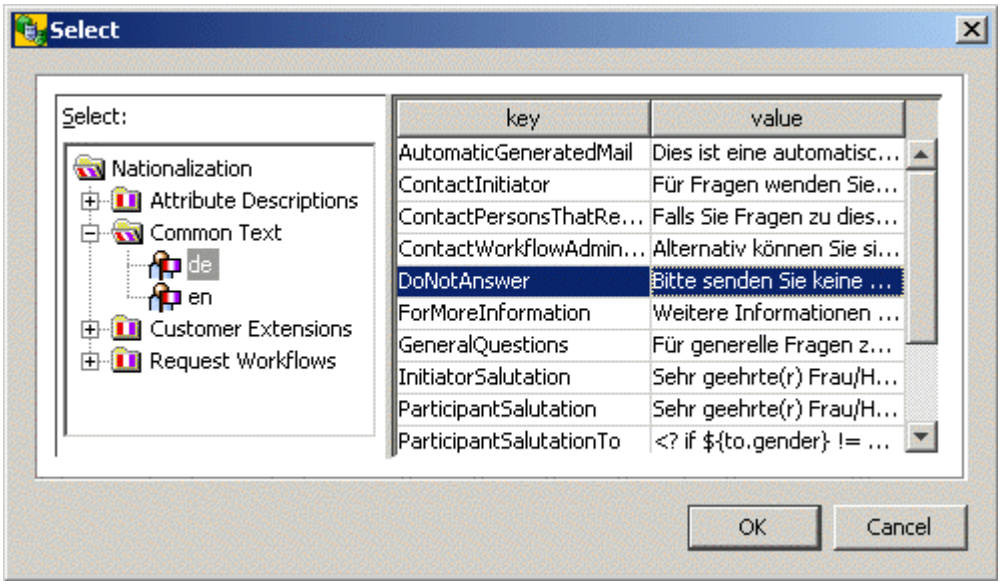



*Figure 2. Select Message*

Browse to the message you want to insert. Select the message and press **OK** to insert the message placeholder, for example **#{Common Text.DoNotAnswer}**.

The resolve a message operation can be performed in edit and display-mode. The following message:

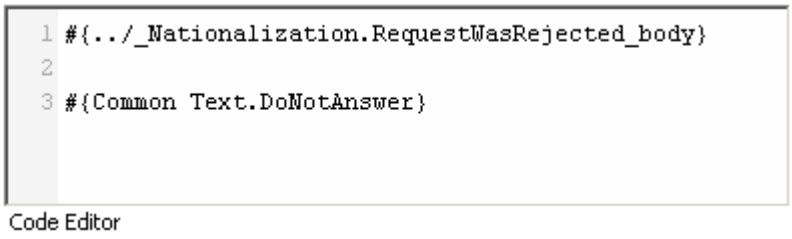



*Figure 3. Unresolved Message*

is resolved with default language English:

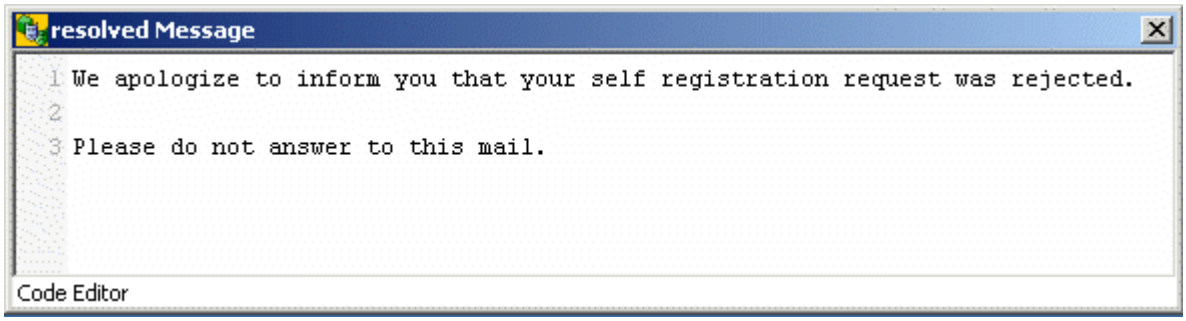



*Figure 4. Resolved Message*

For more information about nationalization support, see "Nationalizing Request Workflows" in the *DirX Identity Application Development Guide*.

## 2.12.5. General Multi-Value Editor

If there is no specific multi-value editor you use the general multi-value editor to handle multi-value attributes.

### Name

siemens.dxr.manager.controls.MetaRoleJnbMultivalue

### Description

This editor handles multi-value attributes. First you specify the corresponding single-value editor followed by the general multi-value editor specification:

**type="***data_type*"
**multivalue="true"**
[**editor="***single-value_editor*"]
[**editorparams="***editor_parameters*"]
**multivalueeditor="siemens.dxr.manager.controls.MetaRoleJnbMultivalue"**

(For a detailed description of the statements, see "Editors".)

The multi-value editor displays the list of values and buttons to add a value  or delete a selected value  .

### Type

multi-value

### Editor parameters

None.

### Data Types

All data types.

### Applicable

General.

### Output Format

Multiple values of the single-value editor specified.

### Example

In the following example, additional locations can be specified for a user:

**Editor specification:**

```
<property name="dxrSecLocationLink"
   type="siemens.dxm.storage.StorageObject"
   label="More Locations"
   multivalue="true"

editorparams="choosefilter=dxrLocation;chooserootdn=cn=Countries,cn=B
usinessObjects,$(rootDN)"
   editor="siemens.dxm.storage.beans.JnbReference"

multivalueeditor="siemens.dxr.manager.controls.MetaRoleJnbMultiValue"
/>
```

**Screenshot:**


Click:

 to add a value.

 to delete a selected value.

## 2.12.6. Editors for Simple Types

This section provides information about editors for simple property types like integer or boolean.

### 2.12.6.1. Editor for Boolean

Use this editor for the handling of Boolean types.

#### Name

siemens.dxm.storage.beans.JnbBoolean

#### Description

This editor handles boolean property types.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.Boolean

#### Applicable

General.

#### Output Format

If the box is checked the editor returns **Boolean.TRUE** else it returns **Boolean.FALSE**.

#### Example

**Editor specification:**

```
type="java.lang.Boolean"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbBoolean"
```

**Screenshot:**

☑
Returns the value **Boolean.TRUE**.

#### 2.12.6.2. Editor for Boolean Integer

Use this editor for the handling of Boolean integer.

#### Name

siemens.dxm.storage.beans.JnbBooleanInteger

#### Description

This editor handles Boolean integer property types. Instead of returning true or false it returns the integer values **1** and **0**.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.BooleanInteger

#### Applicable

General.

#### Output Format

If the box is checked the editor returns **1** else it returns **0**.

#### Example

**Editor specification:**

```
type="java.lang.BooleanInteger"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbBooleanInteger"
```

**Screenshot:**

Returns the value **1**.

### 2.12.6.3. Editor for Boolean Inverse

Use this editor for the handling of Boolean types.

#### Name

siemens.dxm.storage.beans.JnbBooleanInverseDefault

#### Description

This editor handles Boolean property types. The box is checked if the value is set to **True** or if the value does not exist. If the value is **False** the box is not checked.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.Boolean

#### Applicable

General.

#### Output Format

If the box is checked the editor returns **Boolean.TRUE** else it returns **Boolean.FALSE**.

#### Example

**Editor specification:**

```
type="java.lang.Boolean"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbBooleanInverseDefault"
```

**Screenshot:**


Returns the value **Boolean.TRUE**.

#### 2.12.6.4. Editor for Integer

Use this editor for the handling of integer fields.

#### Name

siemens.dxm.storage.beans.JnbIntegerField
siemens.DirXjdiscover.api.beans.JnbIntegerField

#### Description

This editor handles integer property types. It prevents the input of non-digit characters.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.Integer

#### Applicable

siemens.dxm.storage.beans.JnbIntegerField: General
siemens.DirXjdiscover.api.beans.JnbIntegerField: Connectivity

#### Output Format

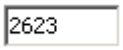
The editor returns the specified integer value.

#### Example

**Editor specification:**

```
type="java.lang.Integer"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbIntegerField"
```

**Screenshot:**




The editor returns the integer value **2623**.

### 2.12.7. Editors for Strings

This section provides information about editors for property types in string format.

#### 2.12.7.1. Editor for Integer Strings

Use this editor for the handling of integer in string format.

#### Name

siemens.dxm.storage.beans.JnbIntegerString

#### Description

This editor handles integers in string format. It prevents the input of non-digit characters.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.String

#### Applicable

General.

#### Output Format

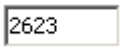
The editor returns the specified integer value in string format.

#### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbIntegerString"
```

**Screenshot:**




The editor returns the integer value **2623** in string format.

#### 2.12.7.2. Editor for Simple One-Line String

Use this editor for the handling of simple one-line strings.

#### Name

siemens.dxm.storage.beans.JnbTextField

#### Description

This editor handles property types in simple one-line string format.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.String

#### Applicable

General.

#### Output Format

The editor returns the specified value in string format.

#### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbTextField"
```

**Screenshot:**


The editor returns the string **Gabriela**.

#### 2.12.7.3. Editor for Simple One Line String with Limited Character Set

Use this editor for the handling of simple one-line strings with a limited character set.

#### Name

siemens.dxm.storage.beans.JnbSimpleTextField

#### Description

This editor handles property types in simple one-line string format. It prohibits specifying the characters **/** (slash), **** (backslash), **.** (dot), **=** (equal-sign), and **:** (colon) to avoid conflicts with nationalization support. No escaping is possible.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.String

#### Applicable

General.

#### Output Format

The editor returns the specified value in string format.

#### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbSimpleTextField"
```

**Screenshot:**


The editor returns the string **Gabriela**.

#### 2.12.7.4. Editor for Simple One-Line String with Nationalization Support

Use this editor for the handling of language-specific simple one-line strings.

#### Name

siemens.dxm.storage.beans.JnbTextFieldNat

#### Description

This editor handles property types in one-line string format. It supports nationalization.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.String

#### Applicable

General.

#### Output Format

The editor returns the specified value in string format.

**Example**

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbTextFieldNat"
```

**Screenshot:**


At run.time the software replaces the placeholder **#{../_Nationalization.ApprovalByPrivilegeManagers_title}** and returns the associated string, for example

> Genehmigung durch den Berechtigungsverantwortlichen

for German or

> Approval by Privilege Managers

for English.

#### 2.12.7.5. Editor for Multiple-Line String

Use this editor for the handling of multiple-line strings.

**Name**

siemens.dxm.beans.JnbBorderTextArea

**Description**

This editor handles property types in multiple-line string format. After pressing the return-key the editor adds a new line.

**Type**

single-value

**Editor parameters**

None.

#### Data Types

java.lang.String

#### Applicable

General.

#### Output Format

The editor returns the specified value in string format.

#### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxm.beans.JnbBorderTextArea"
```

**Screenshot:**


The editor returns the string **Assistant**.

### 2.12.7.6. Editor for Multiple-Line String with Nationalization Support

Use this editor for the handling of language-specific multiple-line strings.

#### Name

siemens.dxm.beans.JnbBorderTextAreaNat

#### Description

This editor handles property types in multiple-line string format. It supports nationalization. After pressing the return-key the editor adds a new line.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.String

#### Applicable

General.

#### Output Format

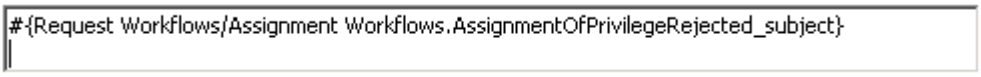
The editor returns the specified value in string format.

#### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxm.beans.JnbBorderTextAreaNat"
```

**Screenshot:**




At run-time the software replaces the placeholder **#\{Request Workflows/Assignment Workflows.AssignmentOfPrivilegeRejected_subject}** and returns the associated string, for example

> Ablehnung der Berechtigung ${workflow.resources[0].dxrassignto@cn} für den Benutzer ${workflow.subject.cn}

for German or

> Assignment of privilege ${workflow.resources[0].dxrassignto@cn} to user ${workflow.subject.cn} was rejected

for English.

At run-time the software replaces the expressions **${workflow.resources[0].dxrassignto@cn}** and **${workflow.subject.cn}** by the current value.

#### 2.12.7.7. Editor for Postal Address

Use this editor for the handling of a postal address.

#### Name

siemens.DirXjdiscover.api.ldap.beans.JnbLdapPostalAddress

#### Description

This editor handles a postal address.

#### Type

single-value

#### Editor parameters

**mandatory=***boolean_value*
If **true**, a postal address must be present (default: **false**).

#### Data Types

java.lang.String

#### Applicable

General

#### Output Format

The editor returns the postal address in DirX syntax.

#### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapPostalAddress"
```

**Screenshot:**


### 2.12.7.8. Editors for String with Specific Values

Use this editor for the handling of string with a specific range of values.

### Name

siemens.dxm.storage.beans.JnbRandomStringTag
siemens.dxm.storage.beans.JnbStringTag
siemens.dxr.manager.controls.MetaRoleJnbComboBox

### Description

This editor handles strings with a specific range of values. It provides a drop-down list containing all valid values. If using the siemens.dxm.storage.beans.JnbRandomStringTag editor you can edit the field. If the field is edited there is no check whether the edited value is contained in the list of valid values. Specify the valid values in a **tag** statement. Separate the values by a comma-sign **,**:

**tags="***value1***,***value2*...**"**

for example:

**tags="IMPORTED,IGNORE,DELETED"**

Alternatively you can store the valid values for a property in proposal lists in the Identity store. (See section "Specifying Proposal Lists in Property Descriptions" for details.)

### Type

single-value

### Editor parameters (only siemens.dxr.manager.controls.MetaRoleJnbComboBox editor)

***editable*=boolean**

Specifies whether the field is editable (**true**) or not (**false**) The default value is **false**. If the field is edited there is no check whether the edited value is contained in the list of valid values.

### Data Types

java.lang.String

### Applicable

siemens.dxm.storage.beans.JnbRandomStringTag: General
siemens.dxm.storage.beans.JnbStringTag: Genaral
siemens.dxr.manager.controls.MetaRoleJnbComboBoxGeneral: Provisioning

### Output Format

The editor returns the selected value in string format.

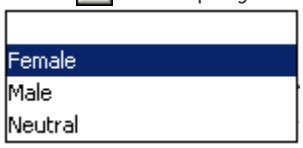
### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
tags="Female, Male,Neutral"
editor="siemens.dxm.storage.beans.JnbRandomStringTag"
```

**Screenshot:**


Click to display the drop-down to select one of the values:




## 2.12.8. Editors for Date and Time

This section provides information about editors for property types in date and / or time format.

### 2.12.8.1. Editor for Simple Time Period

Use this editor for the handling of simple time periods.

#### Name

siemens.DirXjdiscover.api.beans.JnbTimePeriod

#### Description

This editor handles the input of simple time intervals. It displays a text-box to specify an integer string and a combo-box (seconds, minutes, hours, days) to select the time unit for the interval. Specifying non-digit characters in the text-box is prohibited. The editor converts the specified value into seconds.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.Integer

#### Applicable

General

#### Output Format

The editor returns the time period value in integer format in seconds.

#### Example

**Editor specification:**

```
type="java.lang.Integer"
multivalue="false"
editor="siemens.DirXjdiscover.api.beans.JnbTimePeriod"
```

**Screenshot:**


Specify an integer value and select the time unit you want to specify from the drop-down list:


#### 2.12.8.2. Editor for Time Period (Hour-Format)

Use this editor for the handling of simple time periods.

#### Name

siemens.dxm.storage.beans.JnbTimeInterval

#### Description

This editor handles the input of simple time intervals. Only digits and colons can be specified. Specify the value in the format:

*hh:mm:ss*

where

*hh* specifies the number of hours starting from **00**,

*mm* specifies the minutes from **00** through **59**,

*ss* specifies the seconds from **00** through **59**.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

siemens.dxm.util.TimeInterval

#### Applicable

General

#### Output Format

The editor returns the specified time interval value in the format described under "Description" above.

#### Example

**Editor specification:**

```
type="siemens.dxm.util.TimeInterval"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbTimeInterval"
```

**Screenshot:**


### 2.12.8.3. Editor for Time Period

Use this editor for the handling of time periods.

#### Name

siemens.dxr.manager.controls.requestworkflow.JnbTimeout

#### Description

This editor handles the input of time intervals as used within request and real-time workflow definitions.

### Type

single-value

### Editor parameters

***visibleparts*=value**

Specifies the time-period units in:

**s** specifies seconds

**m** specifies minutes

**h** specifies hours

**d** specifies days

**M** specifies moths

**Y** specifies years

***partsperrow*=integer_value**

Specifies the number of fields / parts of the time period per row.

### Data Types

java.lang.String

### Applicable

General

### Output Format

The editor returns the time period value in the format **P***years***Y***months***M***days***DT***hours* **H***minutes***M***seconds***S**.

### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editorparams="visibleparts=dhms;partsperrow=4"
editor="siemens.dxr.manager.controls.requestworkflow.JnbTimeout"
```

**Screenshot:**

#### 2.12.8.4. Editor for Date

Use this editor for the handling of dates.

**Name**

siemens.DirXjdiscover.api.ldap.beans.JnbLdapGeneralizedDate

**Description**

This editor handles dates. Click  to display the calendar widget and select the date or specify the date in the field provided in the following format:

*Mmm dd, yyyy*

where

*Mmm* is the three letter abbreviation for the month, for example Jan, Feb, and so on.

*dd* specifies the day (**01** .. **31**).

*yyyy* specifies the year, for example 2011.

**Type**

single-value

**Editor parameters**

None.

**Data Types**

siemens.dxm.util.GeneralizedDate

**Applicable**

Provisioning

**Output Format**

The editor returns the generalized date in the GeneralizedDate format.

**Example**

**Editor specification:**

```
type="siemens.dxm.util.GeneralizedDate"
multivalue="false"
editor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapGeneralizedDate"
```

**Screenshot:**


Click to display the calendar widget:


### 2.12.8.5. Editor for Generalized Time

Use this editor for the handling of time values.

#### Name

siemens.dxm.storage.beans.JnbGeneralizedTime
siemens.DirXjdiscover.api.ldap.beans.JnbLdapGeneralizedTime

#### Description

This editor handles generalized time objects. Click to display the calendar widget and select the date and time or specify the date in the field provided in the following format:

*dd/MM/yyyy hh:mm:ss* {**AM**|**PM**}

where

*dd* specifies the day (**01** .. **31**).

*MM* specifies the month (**01** .. **12**).

*yyyy* specifies the year, for example 2011.

*hh* specifies the hour (**00** .. **12**).

*mm* specifies the minute (**00** .. **59**).

*ss* specifies the second (**00** .. **59**).

#### Type

single-value

#### Editor parameters

None.

#### Data Types

siemens.dxm.util.GeneralizedTime

#### Applicable

General

#### Output Format

The editor returns the generalized time in GeneralizedTime format.

#### Example

**Editor specification:**

```
type="siemens.dxm.util.GeneralizedTime"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbGeneralizedTime"
```

**Screenshot:**


Click  to display the calendar widget:


### 2.12.8.6. Editor for Local Time

Use this editor for the handling of dates.

#### Name

siemens.dxr.manager.controls.JMetaRoleFilterCalendar

### Description

This editor handles dates. Click  to display the calendar widget and select the date or specify the date in the field provided in the following format:

*Mmm dd, yyyy*

where

*Mmm* is the three letter abbreviation for the month, for example Jan, Feb, and so on.

*dd* specifies the day (**01** .. **31**).

*yyyy* specifies the year, for example 2011.

Click  and select the format from the item list displayed:

- **Calendar** - for displaying the calendar widget and selecting the date from the widget.
- **Local time** - for specifying the value **$(localtime)**. The editor returns the current local date at run-time.
- **GMT time** - for specifying the value **$(gmttime)**. The editor returns the current GMT time at run-time.
- **$(...)** - for specifying a time expression like **$(gmttime+14)**.

### Type

single-value

### Editor parameters

None.

### Data Types

java.lang.String

### Applicable

Provisioning

### Output Format

The editor returns the specified value.

### Example

**Editor specification:**

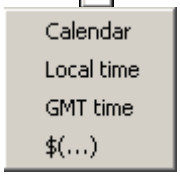
```
type="java.lang.String"
multivalue="false"
```

```
editor="siemens.dxr.manager.controls.JMetaRoleFilterCalendar"
```

**Screenshot:**


Click  to display the calendar widget and select the date:


Click  and select the format from the item list displayed:




## 2.12.9. Editors for References

This section provides information about editors for property types containing references like references to files or distinguished names.

### 2.12.9.1. Editor for File References

Use this editor for references to files.

#### Name

siemens.dxm.storage.beans.JnbFile

#### Description

This editor handles filename properties. It displays the full qualified filename (including the path) and provides buttons to display the file content  or browse to the file  .

#### Type

single-value

#### Editor parameters

***showpath***

If defined, the path to the file is displayed instead of just displaying the name of the file.

*win_unc*

If defined, the path to the file located on a mapped network drive is displayed and stored in UNC format. This parameter has effect only on Windows platform. (Hint: If you are turning this parameter on/off with a already selected file, you should clear the editor and select the file again to make sure that the notation change is also propagated to LDAP).

#### Data Types

java.io.File

#### Applicable

General

#### Output Format

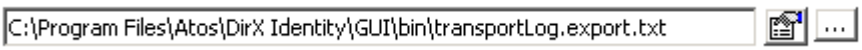
The editor returns the full qualified filename in string format.

#### Example

**Editor specification:**

```
type="java.io.File"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbFile"
```

**Screenshot:**




Click
to display the file content and edit the file.
... to browse to the file.

### 2.12.9.2. Editor for Object References

Use this editor for the handling of references.

#### Name

siemens.dxm.storage.beans.JnbReference

#### Description

This editor handles references to objects. It allows selection of storage objects from a tree. Alternatively the displayed value can be an attribute of the selected object instead of the DN. Click to display the details of the referenced object.

### Type

single-value

### Editor parameters

***nobrowsebutton***

Suppresses the browse button  .

***noremovebutton***

Suppresses the remove button  .

***showpath***

Displays the tree path with display names.

***showabsolutepath***

Shows the path including the root path.

***showdn***

The DN of the object is displayed.

***chooserootdn*=value**

Specifies the root node where to start the object browser.The root node can be the dn of a target system, for example **${tsDN}**, or the dn of a cluster, for example **${tsClusterDN}**.If there is no cluster the target system dn is used.The current object must be a child of a target system or a cluster.

***choosefilter*=object_name_1[,object_name_2 ...]**

Specifies the selectable object names of the object descriptions. Use a comma as the delimiter.

***lastselection=true***

Specifies to remember the last selection. If the editor is invoked again, it tries to show the last selected entry in the tree.

### Data Types

siemens.dxm.storage.StorageObject

### Applicable

General

### Output Format

The editor returns the referenced object.

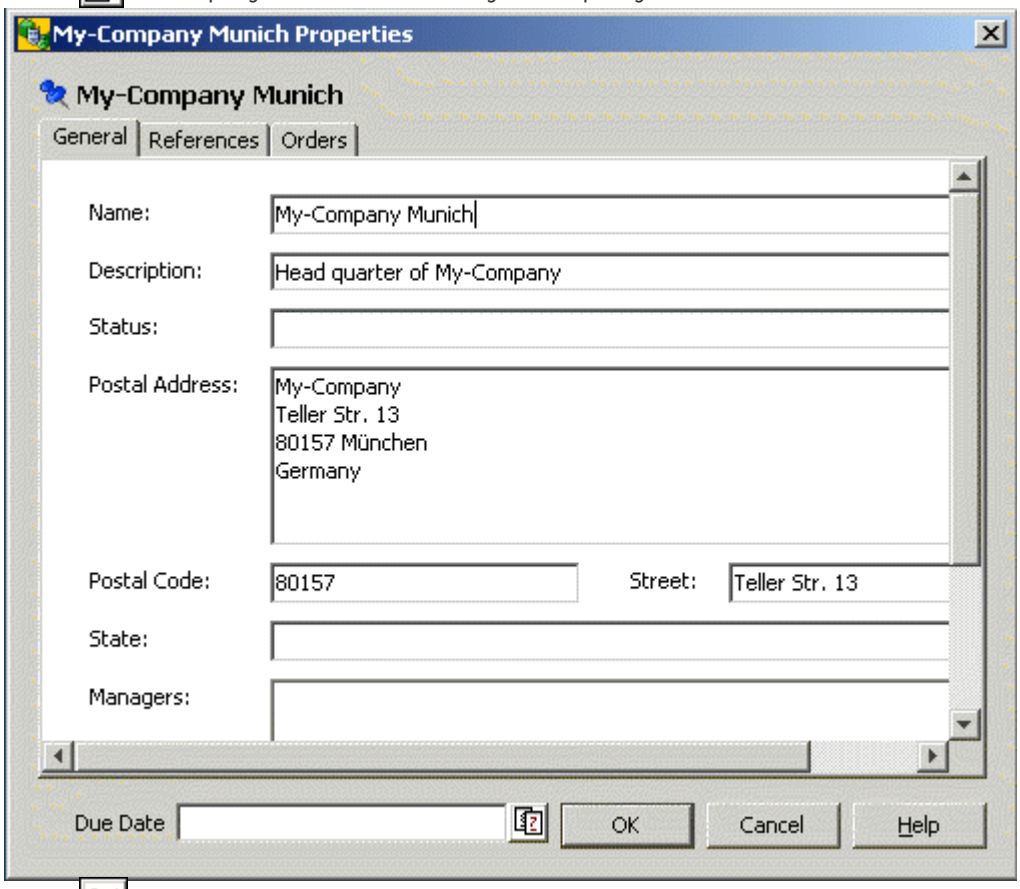
### Example

**Editor specification:**

```
type="siemens.dxm.storage.StorageObject"
multivalue="false"
editorparams="choosefilter=dxrLocation;chooserootdn=cn=Countries,cn=B
usinessObjects,$(rootDN)"
editor="siemens.dxm.storage.beans.JnbReference"
```

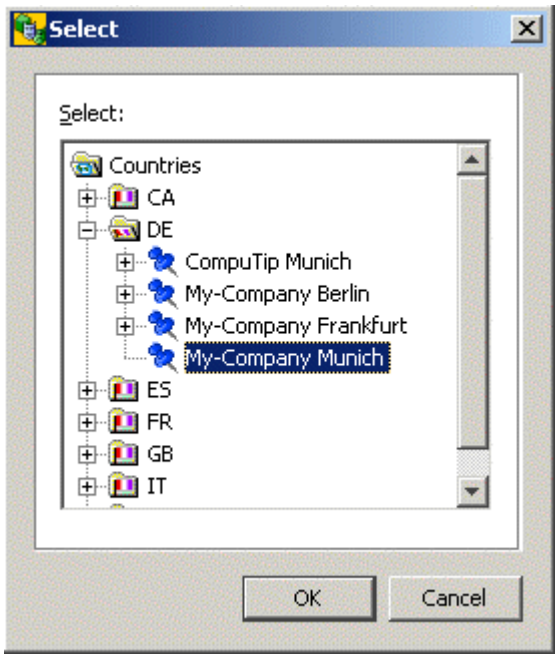
**Screenshot:**


Click  to display the details of My-Company Munich:




Click  to delete the value **My-Company Munich**.

Click  to display the object browser and select the referenced object:




### 2.12.9.3. Editor for Object References in String Format

Use this editor for the handling of references in string format.

#### Name

siemens.dxr.manager.controls.MetaRoleJnbLdapDN

#### Description

This editor handles references to objects in string format. It allows selection of storage objects from a tree. Alternatively you can edit the distinguished name of the referenced object. Click

the button to display the details of the referenced object.
the ... to display the object browser and select the referenced object.

#### Type

single-value

#### Editor parameters

***chooserdn*=value**

Specifies the root node where to start the object browser.The root node can be the dn of a target system, for example **${tsDN}**, or the dn of a cluster, for example **${tsClusterDN}**.If there is no cluster the target system dn is used.The current object must be a child of a target system or a cluster.

***choosefilter*=object_name_1[,object_name_2 ...]**

Specofies the selectable object names of the object descriptions.Use a comma as the delimiter.

#### Data Types

java.lang.String

#### Applicable

Provisioning

#### Output Format

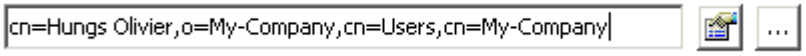
The editor returns the referenced object in string format.

#### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editorparams="choosefilter=dxrUser;chooserdn=cn=Users"
editor="siemens.dxr.manager.controls.MetaRoleJnbLdapDN"
```

**Screenshot:**




## 2.12.10. Editors for Search Filters

This section provides information about editors for the handling of search filters. For more information about search filters, see section "Search Filters" in chapter "String Representation for LDAP Binds" in the *DirX Identity Meta Controller Reference*.

### 2.12.10.1. Common Features of Filter Editors

You can edit the correct filter value in the field provided or press the ... button to open a window with two tabs:

- The **Advanced** tab provides a form that guides the user through building the filter value. Drop-down lists are available for providing all valid values for attributes and operators. Mandatory fields are displayed in red.
  Click

to select one of the valid values from the drop-down list provided.

to provide the context sensitive menu for adding or deleting filter items:

- **New row** - creates a new filter item row.
- **Delete row** - deletes the current filter item row.
- **New group** - creates a new filter item group.

- **Delete group** - deletes the current filer item group.
- **Help** - provides help information.

 to redo last filter input action.

 to undo last filter input action.

- In the **LDAP** tab the user can edit the filter in LDAP format. He must know the correct format, valid attribute names and operators.

#### 2.12.10.2. Editor for Filters with References

Use this editor for the handling of search filters.

**Name**

siemens.dxm.beans.JnbMetaLdapFilterPreview

**Description**

This editor handles the specification of **Search Filters** in Connectivity. This editor can handle references in filters.

**Type**

single-value

**Editor parameters**

***tclinput***

If this parameter is present, the filter value is enclosed in curly brackets **{}**. The default behavior is not to enclose the value.

**Data Types**

java.lang.String

**Applicable**

Connectivity

**Output Format**

The editor returns the filter value in string format.

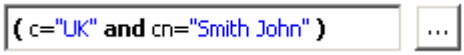
**Example**

**Editor specification:**

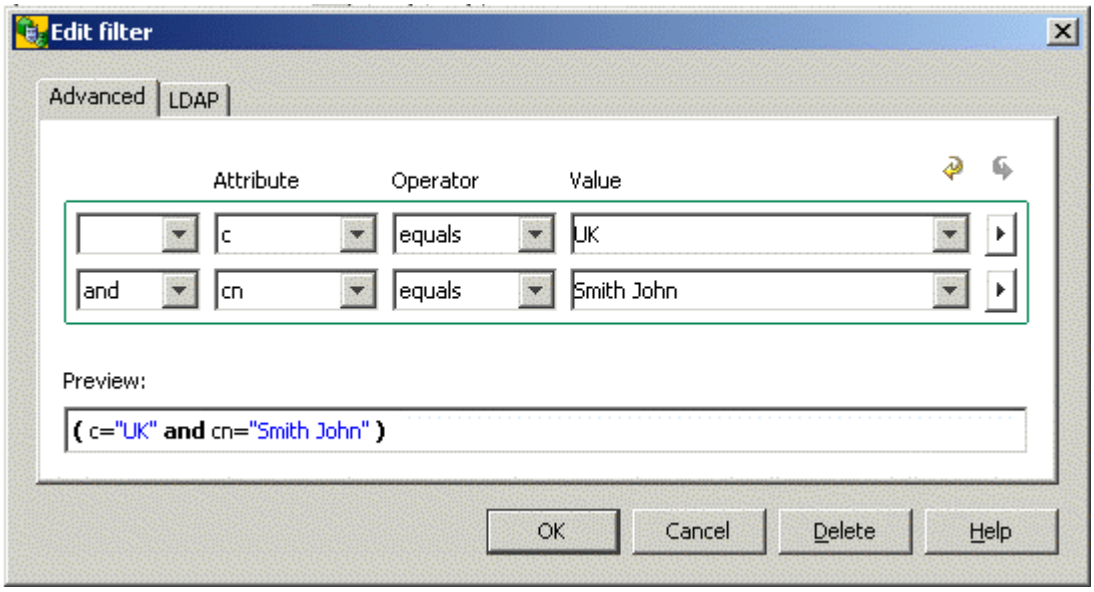
```
type="java.lang.String"
```

```
multivalue="false"
editor="siemens.dxm.beans.JnbMetaLdapFilterPreview"
```

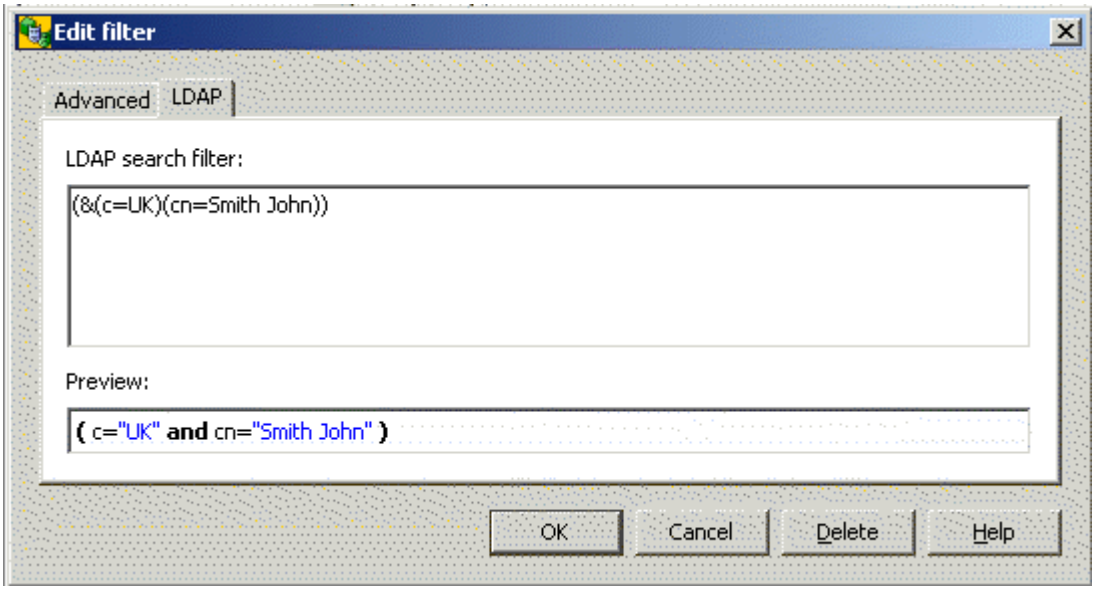
**Screenshot:**




Click ... to edit the filter:*
Advanced* tab:




**LDAP** tab




### 2.12.10.3. Editor for Filters with Variable $Now

Use this editor for the handling of search filters.

**Name**

siemens.dxm.beans.JnbQueryLdapFilterPreview

### Description

This editor handles the specification of **Search Filters**. It can handle the value **$NOW** for attributes of type GENERALIZED_TIME in filters. For this purpose it uses the editor **siemens.dxm.storage.beans.JnbGeneralizedTime** for GENERALIZED_TIME attributes. (See "Editor for Generalized Time" for details.)

### Type

single-value

### Editor parameters

None.

### Data Types

java.lang.String

### Applicable

General

### Output Format

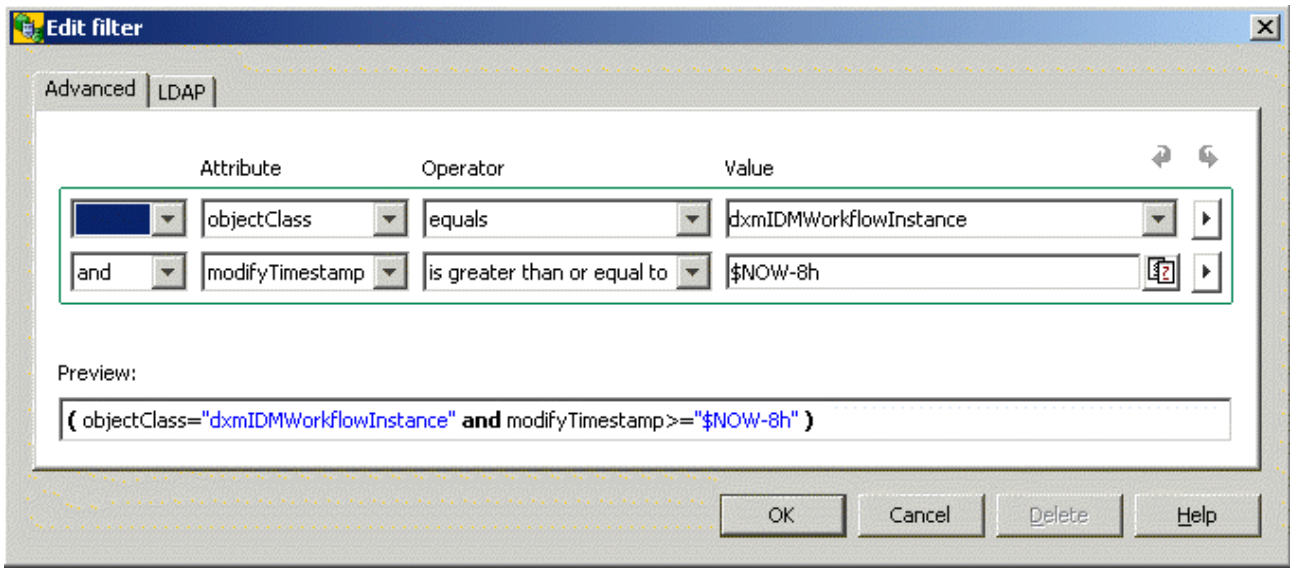
The editor returns the filter value in string format.

### Example

**Editor specification:**

```
type="jav.lang.String"
multivalue="false"
editor="siemens.dxm.beans.JnbQueryLdapFilterPreview"
```

**Screenshot:**




#### 2.12.10.4. Editor for Filter Returning Time Classes

Use this editor for the handling of search filters.

**Name**

siemens.dxr.manager.controls.JnbMetaRoleQueryLdapFilterPreview

**Description**

This editor handles the specification of **Search Filters**. It can handle the value **$NOW** for attributes of type GENERALIZED_TIME in filters and returns the time classes and the time editor class in the Provisioning context. For this purpose it uses the editor **siemens.dxr.manager.controls.JMetaRoleFilterCalendar** for GENERALIZED_TIME attributes. (See "Editor for Local Time" for details.)

**Type**

single-value

**Editor parameters**

None.

**Data Types**

java.lang.String

**Applicable**

Provisioning

**Output Format**

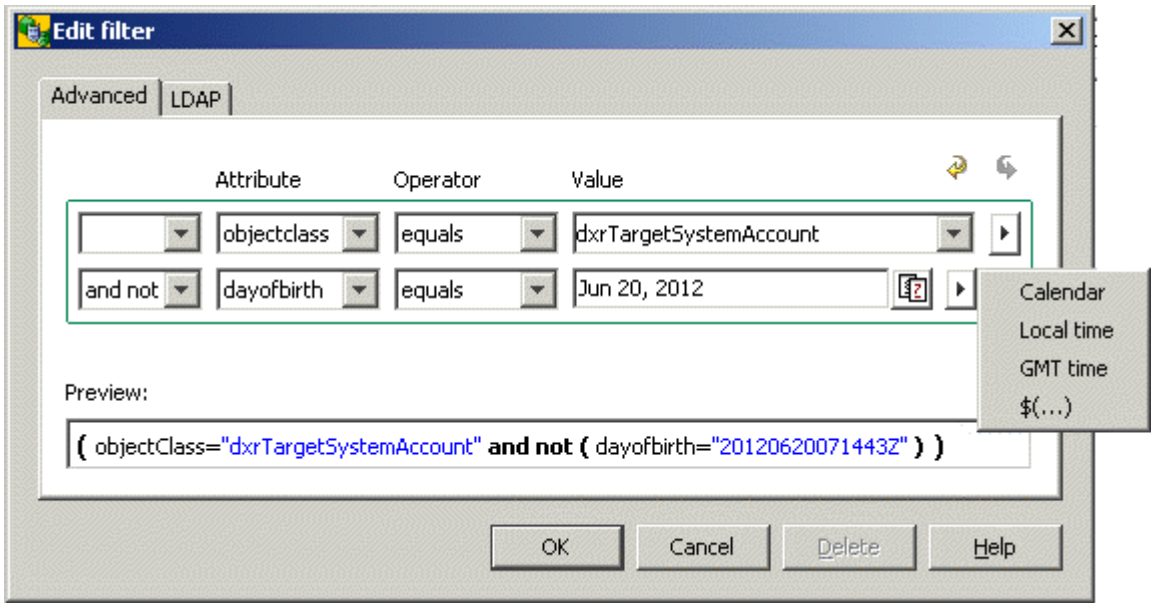
The editor returns the filter value in string format.

#### Example

#### Editor specification:

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxr.manager.controls.JnbMetaRoleQueryLdapFilterPrevie
w"
```

#### Screenshot:




## 2.12.11. Editors for Specific Components and Types

This section provides information about editors for specific property types and components like pictures or filters.

### 2.12.11.1. Change Password Button

Use this editor to handle a password change.

#### Name

siemens.DirXjdiscover.api.beans.JnbPasswordButton

#### Description

This editor handles the modification of passwords. It displays a change password button. After clicking the button a form to change the password opens. The editor checks whether the user specified the correct old password value. If there is a password policy it also checks whether the new password is conformant to the password policy. The editor transfers all passwords in scrambled or encrypted format.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.String

#### Applicable

General

#### Output Format

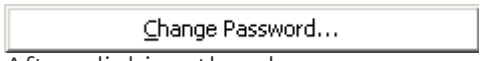
The editor returns the new password value in scrambled or encrypted format.

#### Example

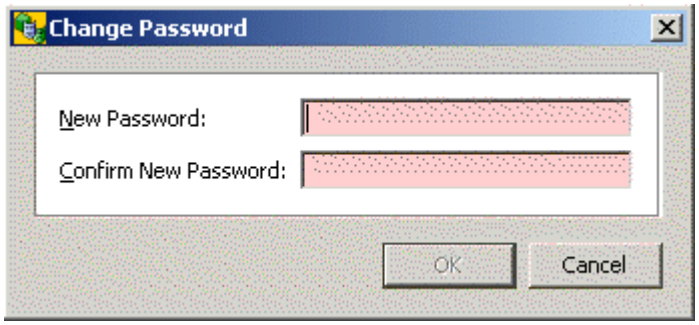
**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.DirXjdiscover.api.beans.JnbPasswordButton"
```

**Screenshot:**




After clicking the change password button the following window is displayed:




### 2.12.11.2. Editor for Passwords (Connectivity)

Use this editor to handle passwords.

#### Name

siemens.dxm.beans.JnbDXMPassword

### Description

This editor handles passwords. Click
the  button to delete the displayed password value.
the  button to open a password dialog that allows handling DirX Identity compatible passwords.

The editor checks whether the user specified the correct old password value if there is one. If there is a password policy it also checks whether the new password is conformant to the password policy. The editor transfers all passwords in scrambled or encrypted format.

### Type

single-value

### Editor parameters

None.

### Data Types

java.lang.String

### Applicable

Connectivity

### Output Format

The editor returns the new password value in scrambled or encrypted format.

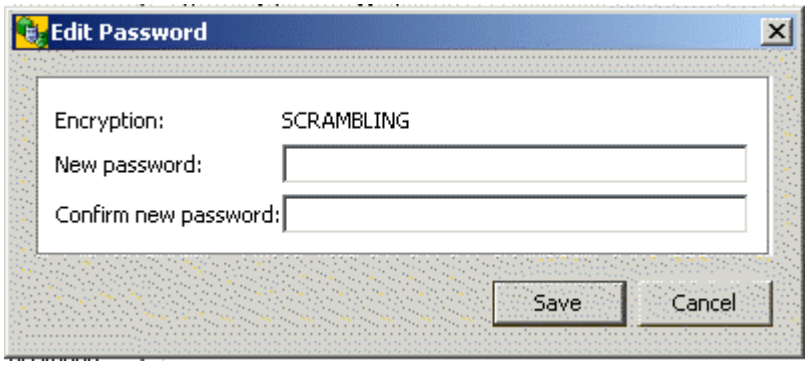
### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxm.beans.JnbDXMPassword"
```

**Screenshot:**


Click  button to delete the displayed password value.
Click  button to open the following window and specify the password value:




#### 2.12.11.3. Editor for Binary

Use this editor for the handling of binary types.

**Name**

siemens.DirXjdiscover.api.ldap.beans.JnbLdapBinary

**Description**

This editor handles binary property types. It provides buttons for import  , export  and display  of the binary value.

**Type**

single-value

**Editor parameters**

None.

**Data Types**

[B

**Applicable**

General

**Output Format**

The editor returns the binary value.

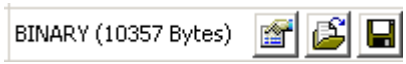
**Example**

**Editor specification:**

```
type="[B"
multivalue="false"
```

```
editor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapBinary"
```

**Screenshot:**




Click
to display the binary data
to import a binary value
to export the binary value.

#### 2.12.11.4. Editor for Binary Data Stored as Base64-encoded String

Use this editor for the handling of p12 files. The binary content of those files is stored as base64-encoded string in the LDAP attribute. It can be used for importing and exporting binary data from a file to an LDAP attribute where the binary data is stored as base64 representation of the binary data.

**Name**

siemens.dxm.storage.beans.JnbLdapBinary2B64

**Description**

This editor handles binary data. The binary data is internally converted to a base64-encoded string.

**Type**

single-value

**Editor parameters**

***noshowbutton***
If defined, the Details button is not visible (default: not defined (visible)).

***noimportbutton***
If defined, the Import button is not visible (default: not defined (visible)).

***noexportbutton***
If defined, the Export button is not visible (default: not defined (visible)).

***nodeletebutton***
If defined, the Delete button is not visible (default: not defined (visible)).

**Data Types**

Java.lang.String

#### Applicable

General

#### Output Format

The editor returns a base64-encoded string.

#### Example

#### Editor specification:

```
type="java.lang.String"
editor="siemens.dxm.storage.beans.JnbLdapBinary2B64"
```

#### Screenshot:


Click
 to display the Base64 String
 to import binary data from a file
 to export binary data to a file
 to delete the content.

### 2.12.11.5. Editor for Certificate

Use this editor for the handling of one certificate.

#### Name

siemens.DirXjdiscover.api.ldap.beans.JnbLdapCertificate

#### Description

This editor handles one certificate.

#### Type

single-value

#### Editor parameters

***mandatory*=boolean_value**
If **true**, a certificate must be present (default: **false**).

***showButtons*=boolean_value**
If **true**, all buttons are visible (default: **true**).

***showDetailsButton*=boolean_value**

If **true**, the Details button is visible (default: **true**).

***showImportButton*=boolean_value**

If **true**, the Import button is visible (default: **true**).

***showExportButton*=boolean_value**

If **true**, the Export button is visible (default: **true**).

***showDeleteButton*=boolean_value**

If **true**, the Delete button is visible (default: **true**).

### Data Types

[B

### Applicable

General

### Output Format

The editor returns the certificate value.

### Example

**Editor specification:**

```
type="[B"
multivalue="false"
editor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapCertificate"
```

**Screenshot:**


Click
to display the details of the certificate
to import a certificate
to export the certificate
to delete the certificate.

#### 2.12.11.6. Editor for Certificates

Use this editor for the handling of multiple certificates.

**Name**

siemens.DirXjdiscover.api.ldap.beans.JnbLdapCertificates

**Description**

This editor handles multiple certificates. (See "General Multi-Value Editor" for general information about the use of multi value editors.)

**Type**

multi-value

**Editor parameters**

None

**Data Types**

[B

**Applicable**

General

**Output Format**

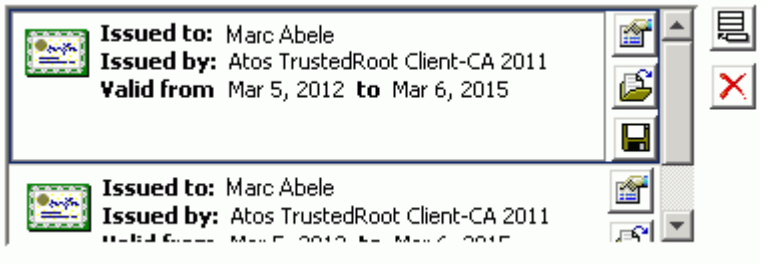
The editor returns the multiple certificate values.

**Example**

**Editor specification:**

```
type="[B"
multivalue="true"
editor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapCertificate"
multivalueeditor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapCertifi
cates"
```

**Screenshot:**




Click
to display the details of the certificate
to import a certificate
to export the certificate
to add a new certificate
to delete the selected certificate.

#### 2.12.11.7. Editor for IP Address

Use this editor for the handling of IP addresses.

**Name**

siemens.dxm.storage.beans.JnbIPAddressField

**Description**

This editor handles IP addresses. Specify either the DNS name or the TCP/IP address in dotted notation.

**Type**

single-value

**Editor parameters**

None.

**Data Types**

siemens.dxm.util.IPAddress

**Applicable**

General

**Output Format**

The editor returns the IP address.

**Example**

**Editor specification:**

```
type="siemens.dxm.util.IPAddress"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbIPAddressField"
```

**Screenshot:**

DNS name: kelle20-vm

TCP/IP address: 123.45.67.890

#### 2.12.11.8. Editor for Picture

Use this editor for the handling of pictures.

**Name**

siemens.DirXjdiscover.api.ldap.beans.JnbLdapJPEG

**Description**

This editor handles pictures. It displays buttons to view  , import  , export  , and delete  a picture.

**Type**

single-value

**Editor parameters**

***showButtons*=boolean_value**
: If **true**, all buttons are visible (default: **true**).

***showViewButton*=boolean_value**
: If **true**, the View button  is visible (default: **true**).

***showImportButton*=boolean_value**
: If **true**, the Import button  is visible (default: **true**).

***showExportButton*=boolean_value**
: If **true**, the Export button  is visible (default: **true**).

***showDeleteButton*=boolean_value**
: If **true**, the Delete button  is visible (default: **true**).

**Data Types**

[B

#### Applicable

General

#### Output Format

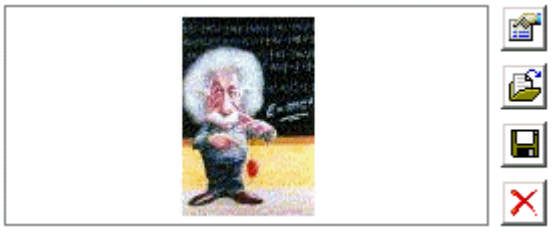
The editor returns the binary value.

#### Example

**Editor specification:**

```
type="[B"
multivalue="false"
editor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapJPEG"
```

**Screenshot:**



Click
to view the photo
to import a photo
to export the photo
to delete the photo.

## 2.12.12. Editors for Code

This section provides information about editors for code like Tcl or JavaScript.

#### 2.12.12.1. Common Features of Code Editors

This section provides information about common features for all code editors. Note that some code editors may not provide all features.

#### Cursor Location

The code editors display row and column of the cursor position below the code area, for example Row: 57, Column: 17.

#### Context-Sensitive Menu

When clicking the context-sensitive mouse-button anywhere in the code a context-sensitive menu opens providing the following operations:

- **New Window** - opens the code in a new window.
- **Undo** - undo last action. You can undo one action.
- **Redo** - redo last action. You can redo one action.
- **Cut** - cuts the selected text.
- **Copy** - copies the selected text to clipboard.
- **Paste** - pastes the text from clipboard to the cursor location.
- **Find...** - opens a find dialog for searching specific strings.
- **Replace...** - opens a find and replace dialog for replacing specific strings.
- **Go to insertion point** - locates the cursor at the next location for inserting new code.
- **Find other block end** - locates the cursor at the next block end, for example at the end of an if statement.
- **Select all** - selects the whole code.
- **Line numbers** - if checked displays the line number at the beginning of the line.
- **Close** - closes the new window and returns to the old window.

#### Common Buttons

The code editors provide the following buttons:

- **External Edit...** - opens an external editor, for example Notepad. Specify the external editor in the **dxi.cfg** file. (See "Customizing the Property File (dxi.cfg)" in the *DirX Identity User Interfaces Guide* for details.)
- **Import ...** - opens a file browse dialog to import code from an external file.
- **Export ...** - opens a file browse dialog to export the code to an external file.

### 2.12.12.2. Editor for JavaScript

Use this editor for the handling of JavaScript.

#### Name

siemens.dxm.storage.beans.JnbJavaScriptTextArea

#### Description

This editor handles JavaScript code. It provides buttons to import and export JavaScript code, and a button to open an external editor.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

siemens.dxm.util.TextAreaString

#### Applicable

General

#### Output Format

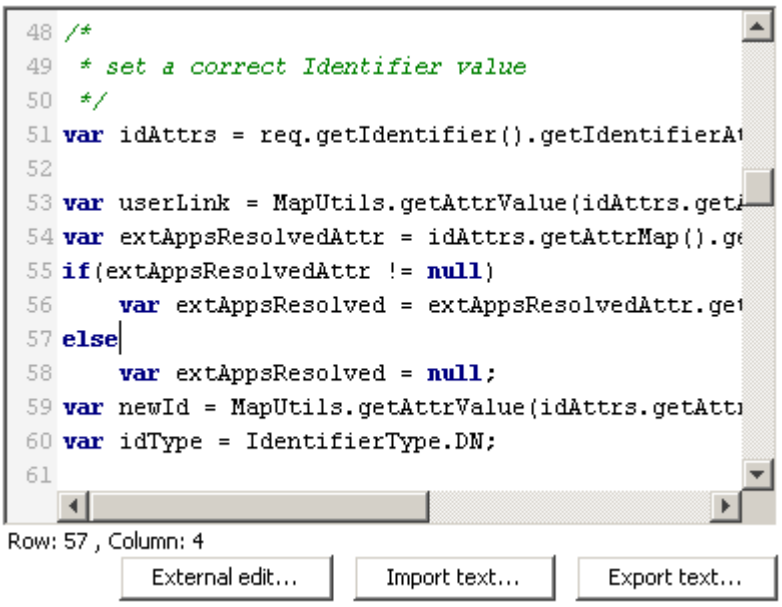
The editor returns the JavaScript code in string format.

#### Example

**Editor specification:**

```
type="siemens.dxm.util.TextAreaString"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbJavaScriptTextArea"
```

**Screenshot:**




### 2.12.12.3. Editor for Tcl

Use this editor for the handling of Tcl.

#### Name

siemens.dxm.beans.JnbTCLEditor

### Description

This editor handles Tcl code. It provides buttons to import and export Tcl code, and a drop-down list to go to named procedures.

### Type

single-value

### Editor parameters

None.

### Data Types

siemens.dxm.util.TCLCodeString

### Applicable

Connectivity

### Output Format

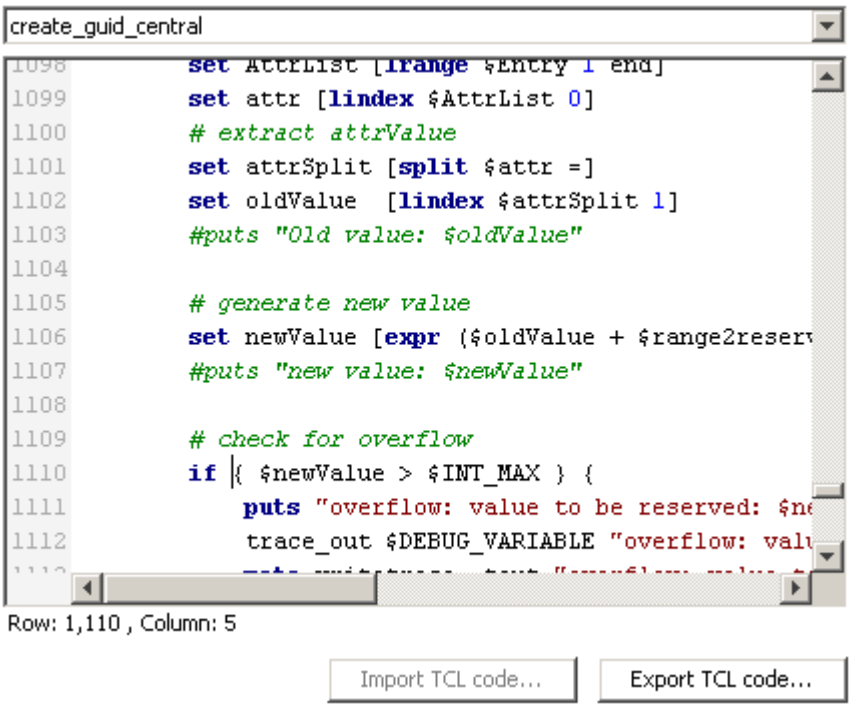
The editor returns the Tcl code in string format.

### Example

**Editor specification:**

```
type="siemens.dxm.util.TCLCodeString"
multivalue="false"
editor="siemens.dxm.beans.JnbTCLEditor"
```

**Screenshot:**




#### 2.12.12.4. Editor for Text

Use this editor for the handling of texts.

#### Name

siemens.dxm.storage.beans.JnbTextArea

#### Description

This editor handles texts. When pressing **Enter** the editor adds a new line.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.String

#### Applicable

General.

#### Output Format

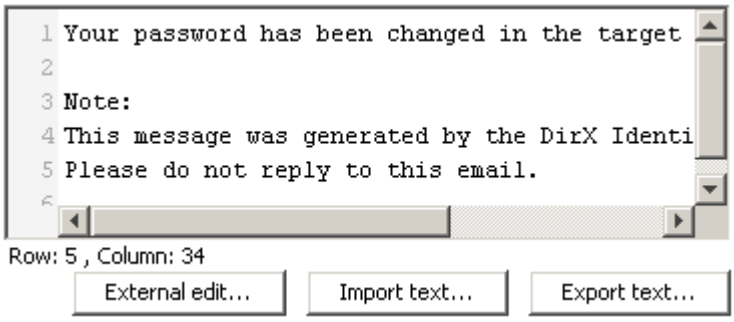
The editor returns the specified text in string format.

#### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbTextArea"
```

**Screenshot:**




### 2.12.12.5. Editor for Text with Nationalization Support

Use this editor for the handling of language-specific texts.

#### Name

siemens.dxm.storage.beans.JnbTextAreaNat

#### Description

This editor handles language specific texts. It supports nationalization. When pressing **Enter** the editor adds a new line.

#### Type

single-value

#### Editor parameters

None.

#### Data Types

java.lang.String

#### Applicable

General.

#### Output Format

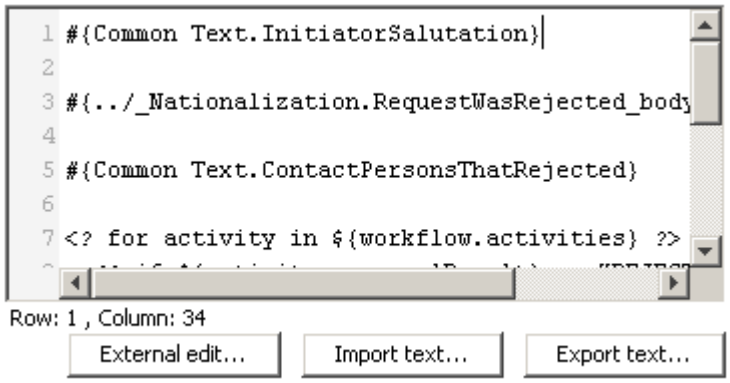
The editor returns the specified text in string format.

#### Example

**Editor specification:**

```
type="java.lang.String"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbTextAreaNat"
```

**Screenshot:**




The editor replaces the placeholder **#{Common Text.IniatorSalutation}** with

> Sehr geehrte(r) Frau oder Herr ${workflow.initiatorEntry.sn},

for German or

> Dear Mrs./Mr. ${workflow.initiatorEntry.sn},

for English.

At run-time the software replaces the expression **${workflow.initiatorEntry.sn}** by the current value.

#### 2.12.12.6. Editor for XML

Use this editor for the handling of XML.

#### Name

siemens.dxm.storage.beans.JnbXMLTextArea

### Description

This editor handles XML code. It provides buttons to import and export XML code, and a button to open an external editor.

### Type

single-value

### Editor parameters

None.

### Data Types

siemens.dxm.util.TextAreaString

### Applicable

General

### Output Format

The editor returns the XML code in string format.

### Example

**Editor specification:**

```
type="siemens.dxm.util.TextAreaString"
multivalue="false"
editor="siemens.dxm.storage.beans.JnbXMLTextArea"
```

**Screenshot:**

# 2.13. Customizing Provisioning Objects

This section describes how to:

- Customize provisioning object property descriptions
- Add a new user attribute
- Add a new role attribute

## 2.13.1. About Provisioning Object Descriptions

DirX Identity Provisioning object descriptions are located in the domain configuration subtree:

**cn=Configuration,cn=***DomainName*

where *DomainName* is the name of your domain.

Most of the object descriptions are system-specific and may be overwritten without notice on each DirX Identity update installation. The object descriptions for users, roles, permissions, accounts, groups, and for entries that represent the target system itself are intended to be changed by the DirX Identity customer. Each of these object descriptions is divided into a customizable part and a system-specific part. The customizable part imports the system-specific part and may extend it with additional properties or overwrite default descriptions of the system part.

The object descriptions in the following folders are system-specific:

**cn=ObjectDescriptions,cn=Configuration,cn=***DomainName* - contains the basic descriptions and common parts of target system-specific or customer-modifiable object descriptions.

**cn=TargetSystems,cn=Configuration,cn=***DomainName* - contains the configuration data that is target system- specific. A subfolder exists for each type of target system that DirX Identity supports. The object descriptions in each subfolder are for accounts, groups and for the entry that represents the target system itself.

The object descriptions in the following folders will typically be customized and will not be overwritten during an upgrade installation:

**cn=Object Descriptions,cn=Customer Extensions,cn=Configuration,cn=***DomainName* - These descriptions are for all objects associated with real directory entries, especially for the user, the role and the permission entries.

if you want to add completely new object types and the corresponding object descriptions, enter all of these object description paths into the file **main.xml**. The referenced files are read at the end of the configuration phase.

**cn=Object Descriptions,cn=Configuration,cn=***TargetSystemType, ...* - These descriptions are for target system-specific objects. The object descriptions here contain templates for

the target system entry and its accounts and groups. The default templates simply import the object description of the target system type, which allows you to configure them differently for each target system instance.

## 2.13.2. Provisioning Object Naming Conventions

In the Provisioning view group, objects are clearly separated into views. Within these views naming depends on customer requirements and is not restricted.

The following tips help working with DirX Identity. View also the example objects of the DirX Identity Sample Domains to see how names could be built.

#### 2.13.2.1. Privileges

If a group is only assigned to one permission and one role it makes sense to use the same name for the group, permission and role.

#### 2.13.2.2. Policies

DirX Identity comes with this default naming schema:

*subject operation resource*

where operation denotes the action like **read** or **modify**. Use **handle** if read and modify are used both.

For example:

```
UserAdmins create accounts

Users handle passwords of their accounts
```

#### 2.13.2.3. Request Workflows

DirX Identity comes with this default naming schema:

*operation subject modifier*

For example:

```
Create User with Approval

Modify User

Approve Customer Self Services
```

#### 2.13.2.4. Target Systems

A good default naming schema is:

*type name area*

where *type* stands for the target system type. Alternatively, you can use a target system or cluster folder to group target systems. For *name*, we recommend using the real and well-known names for target systems. These are, for example, machine names or project names. In many cases, it makes sense to specify additionally the *area* for which this target system is responsible.

For example:

```
Windows Domain XYZ Europe

SAP R/3 System ABC
```

### 2.13.3. Invoking JavaScript Programs from Provisioning Objects

You can use the **RunJavaScriptByURL** action in a Provisioning object description to add a context menu entry that invokes a specific JavaScript procedure to operate on one or more objects selected in DirX Identity Manager.

For example, the DirX Identity workflow instance object description (**WorkflowInstance.xml**) includes the **RunJavaScriptByURL** action to call a JavaScript procedure that restarts request workflow instances. Here is the definition:

```
<action class="siemens.dxr.manager.actions.ActionRunJavaScriptByURL"
multiselection="true"
parameter="restartWF@storage://DirXmetaRole/cn=restartReqWF.js,cn=Req
uestWorkflows,cn=JavaScripts,cn=Configuration,$(rootDN)?content=dxrOb
jDesc" />
```

As shown in this example:

- The part of the parameter definition that occurs before the "at" sign (@) defines the string that is presented in the object's context menu. The string is composed of the fixed string **RunJavaScript** followed by a specific string (**restartWF** in the example).
- The remainder of the parameter definition gives the URL of the JavaScript procedure to be run on the object (a workflow instance, in this example) when the user selects the context menu entry for the action (**RunJavaScript restartWF** in the example)..
- The action class parameter **multiselection="true"** specifies that the action is displayed in the context menu when multiple entries are selected so that the JavaScript procedure is run on all of the selected objects sequentially (scriptContext.getObject() returns the current entry). Using this method could be time-consuming for larger scripts. By default, **multiselection** is set to false so that the menu selection is not visible on multiple entry selections.

### 2.13.4. Customizing Provisioning Object Property Descriptions

This section describes how to customize property descriptions for provisioning objects. It includes information about:

- Using the master attribute in a property element
- Specifying naming rules in XML and JavaScript
- Handling property dependencies

#### 2.13.4.1. Using the Master Attribute in a Property Element

Here is an example of a **master** attribute in a **property** element:

```
<property
     name="sn"
     label="surname"
     type="java.lang.String"
     mandatory="true"
     master="dxrUserLink"
/>
<property
     name="dxrUserLink"
     label="user DN"
     type="java.lang.String"
/>
```

The **master="dxrUserLink"** instruction directs DirX Identity to get the attribute value of the **sn** property from the object referenced by the link in the attribute **dxrUserLink** during each save operation. In this example, the link is the corresponding user object. Furthermore, if DirX Identity creates the object, it takes the value automatically from the user object. Note that the property must have the same name in both objects.

If working with hierarchical object descriptions and you intend to fill the property with another method (for example a JavaScript) while a higher object description defined a master relationship simply set **master=""** to break this higher rule.

The **mandatory="true"** instruction indicates to DirX Identity that it needs a value for this attribute when storing it.

The **master** attribute can handle attributes of type binary (byte arrays, indicated by the type **[B** in the object description). You can use this mechanism, for example, to transport all certificates of a user to a privileged account. Describe the userCertificate property as shown below:

```
<property name="userCertificate"
 type="[B"
 label="Certificate"
 clearOnMasterRemoval="true"
```

```
 multivalue="true"
 master="dxrUsedBy"
 editor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapCertificate"
/>
```

Note that the **clearOnMasterRemoval="true"** instruction ensures removal of the last certificate when the last user is unassigned. Normally, the synchronization of master properties does not clear attributes if the master is null.

You can also master an attribute with a different attribute name. Use the following syntax:

```
master=<linkAttributeName>.<attributeNameInLink>
```

Here is an example of mastering the **manager** user attribute to the **accountManager** account attribute:

```
<property
    name="accountManager"
    l    master="dxrUserLink.manager"
/>
```

To specify case-sensitive mastering, use the **,cs** syntax. You can combine this syntax with the mastering of different attribute names. Case-sensitive mastering means that a change from **abC** to **aBC** will be mastered, so the account's value is **aBC**. Use the following syntax:

```
master=<linkAttributeName>,cs[.<attributeNameInLink>]
```

Here is an example of case-sensitive mastering for the **emloyeeNumber** attribute:

```
<property
   name="employeeNumber"
       master="dxrUserLink,cs"
/>
```

#### 2.13.4.2. Specifying Naming Rules in XML

The **namingRule** element is specified within the definition of the property. It is embedded in an **extension** XML element and has the following format:

```
<namingRule>
```

```
    <reference
        baseObject="identification of the object from which you take
the value"
        address="[optional] attribute with a reference (DN) to the
associated object"
        attribute="attribute type"
    />
</namingRule>
```

A naming rule can be used for account and user objects. They are most commonly used when DirX Identity is to create accounts.

You can specify a number of naming rules for one property. DirX Identity evaluates them beginning with the first one until it is able to generate a non-null value. This feature allows you to define alternatives when you are not sure that a necessary attribute value is available. For example: take the mail address from the user if he has one specified; otherwise generate a default one.

If an object description hierarchy has naming rules at all hierarchy levels, the naming rules are combined and evaluated in sequence.

You can use the following naming rule to stop evaluating naming rules:

```
<namingRule>
    <break/>
</namingRule>
```

If this naming rule is reached, the evaluation stops and the property is assumed to have no value.

If a value cannot be generated at the end of the evaluation sequence, an error message is issued.

If you define a JavaScript and naming rules exist at a lower level, you can exclude the namingRule evaluation at the lower levels if you define an empty naming rule in your object description:

```
<extension>
    <namingRule/>
</extension>
```

When DirX Identity creates an account, its property values can be taken from a number of objects associated with this account. The object is identified by the reference element within the XML naming rule. If you take a property

- from its associated user, use "SvcUser"
- from another property of the same account, use "SvcTSAccount"
- from a property of an associated account, create a reference to "SvcTSAccount". See the Exchange 5.5 mailbox example later in this discussion for instructions on how to specify the reference
- from a default value you stored at the target system object, use "SvcTS"

Here is an example that takes the surname from the associated user object:

```
<property name="sn" type="java.lang.String" mandatory="true">
    <extension>
       <namingRule>
           <reference baseObject="SvcUser" attribute="sn"/>
        </namingRule>
    </extension>
</property>
```

Within the **namingRule** element, the **reference** element's **baseObject** attribute directs DirX Identity to take the property value from the **SvcUser** object, which is the associated user for this account.

The following table describes the XML attributes for the **reference** element.

*Table 5. Reference Element Attributes*

| Attribute | Description |
|---|---|
| baseObject | Identifies the object by its class name. Only the symbols "**SvcUser**" (the associated user of an account) and "**SvcTSAccount**" (the account itself) are allowed. |
| attribute | Identifies the property of the base object. |
| range | Identifies a portion of the property value from *index0* through *index1*. An index value of 0 denotes the first character. The "$" character denotes the end of the string: "$-3:$" mean the last 3 characters of the string. The rule fails if the property value is too short for the range. |
| address | Specifies the attribute within the base object that contains the property value. |
| case | Specifies the case for the reference element value. The default is **case="lower"**. |
| type | Specifies the types of values allowed for the property. Possible values are string (default) and integer. If **type = "integer"**, the rule fails if a character within the range is outside of 0-9. |

Sometimes it is necessary to get values from another associated object, as is the case with Exchange 5.5: the Exchange mailbox needs an associated account in some Windows

domain. To reference this account and take some attribute values from it, use the **address** attribute of the reference element. For example:

```
<reference
baseObject="SvcTSAccount"
address="dxrToPeer"
attribute="cn"
/>
```

The **address** attribute specifies the attribute within the account that holds a reference to the base object. In the example just shown, the mailbox attribute **dxrToPeer** contains the distinguished name (DN) of the associated account, the base object.

The previous example to reference an attribute of the associated user can be re-written in the following form and returns the same result:

```
<property name="sn" type="java.lang.String" mandatory="true">
     <extension>
          <namingRule>
               <reference baseObject="SvcUser" address="dxrUserLink"
attribute="sn"/>
          </namingRule>
     </extension>
</property>
```

The account attribute **dxrUserLink** holds the DN of the user and it is taken as the reference from the account to its associated user.

The same and much more can be achieved by the **master** attribute. The following example results in the same default value as the previous example:

```
<property name="sn" type="java.lang.String" mandatory="true"
master="dxrUserLink" />
```

This instruction directs DirX Identity to take the value for the **sn** property from the object referenced by the property **dxrUserLink**, and this property is the associated user of the account. With the **master** attribute, properties designated as mastered by another object are always updated when the property in the master is updated. This functionality is currently only available for the account properties and the associated user.

The **namingRule** element allows more complex constructs to compose a property value from fixed parts, other property values and random numbers. Consider the following example, which composes the common name of an account from the first letters of the

surname and givenName properties, the last five digits of the telephone number of the account itself, a fixed value, a counter and a random number (this example is not a typical naming rule, but it includes all components):

```
<property name="cn" mandatory="true" readonly="true"
type="java.lang.String" >
     <extension>
          <namingRule>
               <reference
                   baseObject="SvcTSAccount"
                   attribute="sn"
                   range="0:0"
                    case="lower"
               />
               <fixedValue
                    value="0"
               />
          <reference
                    baseObject="SvcTSAccount"
                    attribute="givenName"
                    range="0:0"
                     case="lower"
               />
               <counter
                   min="0"
                   max="9"
               />
               <reference
                   baseObject="SvcTSAccount"
                   attribute="telephoneNumber"
               range="$-3:$"
                   type="integer"
               />
               <random
                   min="700"
                   max="901"
               />
          </namingRule>
      </extension>
</property>
```

The result for the user **John Doe** with telephone number **++49-89-636-12345** is **d0j0345789**.

The **address** attribute of the naming rule's **reference** inner element allows you to specify an indirect reference to obtain values in attributes that can be determined at runtime. Candidates for this feature are attributes stored in base objects. Since it is not clear which explicit base object will really be used for obtaining the value, we simply reference the attribute at which to find the requested value. For example:

```
<reference baseObject=baseObjectName address=addressName
attribute=attributeName/>
```

where *baseObjectName* specifies the object at which to look up a particular attribute, *addressName* specifies the address of this object in the actual object, and *attributeName* specifies the name of the respective attribute.

The **fixedValue** inner element of **namingRule** allows you to specify a fixed prefix or suffix.

The **counter** inner element of **namingRule** directs DirX Identity to select a value beginning from "min" up to "max", until the resulting property value is unique. This element can be used to create unique names or email addresses. For example, suppose there are two users "John Doe" for which an email address must be created. The **counter** element allows DirX Identity to calculate the addresses **john.doe0@company.com** and **john.doe1@company.com**.

The **random** inner element of **namingRule** permits you to create random numbers within the minimum-to-maximum range.

These inner elements can be specified in an arbitrary sequence.

Note that the sequence of property specification in the XML object description is critical. The default calculation is performed according the sequence in the object description. This is the reason why the above example works: calculation of the **cn** property depends on the properties **sn**, **givenname** and **telephonenumber**, which are specified prior to **cn**.

#### 2.13.4.3. Specifying Naming Rules with JavaScript

You can use JavaScript programs to specify complex naming rules. JavaScript programs can be specified as separate objects next to the object and property page descriptions. DirX Identity currently supports only target system-specific JavaScript programs, which are located below the target system folders.

The following example shows how JavaScript programs can be referenced from the property definition in the object description to create the **dxrPassword** property for an account:

```
<property name="dxrPassword" type="java.lang.String"
    defaultvalue="$(script:createPassword)" >
    <script name="createPassword" return="password"
```

```
        reference="storage://DirXmetaRole/cn=Password.js,
        cn=JavaScripts,$(./../..)?content=dxrObjDesc"/>
</property>
```

The **script** element specifies the name of the script and the script variable that contains the return value. The **reference** attribute specifies the DN and the property of the object that contains the script code. The DN reference contains a variable part "$(./../..)", which specifies a relative path starting from the object description, as in file systems. In this example, the resulting object DN is:

**cn=Password.js, cn=JavaScripts, cn=Configuration,***target_system_dn*

The source code itself is in the property **dxrObjDesc**.

DirX Identity installs sample scripts for creating account names, passwords and descriptions in the system and demo domains.

If you write your own scripts, note that you get the current object (here: the account) from the method scriptContext.getObject(). To read a property, use the method getProperty().

DirX Identity does not currently support using Java scripts for plausibility checks of properties.

##### 2.13.4.3.1. Example

The following example shows the calculation of the dxrName attribute of an account with the script dxrNameForAccounts. It creates in conjunction with the uniqueIn functionality a unique 8 char long string from the first letter of givenname, the first 5 letters of surname, and the last 2 digits of the telephone number in reverse order.

The snippet of the object description definition in TSAccount.xml is:

```
<property name="$UniqueCounterSH" type="java.lang.Integer" />
<property name="dxrName" type="java.lang.String"
defaultvalue="$(script:dxrNameForAccounts)"
 uniqueIn="$(./../../..)" dependsOn="$UniqueCounterSH">
 <script name="dxrNameForAccounts"
     return="dxrName"

reference="storage://DirXmetaRole/cn=dxrNameForAccounts.js,cn=javascr
ipts,cn=configuration,$(./../../..)?content=dxrObjDesc"/>
</property>
```

The script code is:

```
importPackage(java.lang);
var obj=scriptContext.getObject();
var gn=obj.getProperty("givenName");
var sn=obj.getProperty("sn");
var triesProperty = obj.getProperty("$UniqueCounterSH");
var tries = 0;
if (triesProperty != null) {
 tries = triesProperty.intValue();
}
obj.logger.log("DBG", "dxrNameForAccount: tries="+tries);
// try 8 times to create a unique dxrName attribute value for the
account
if (tries > 8) {
// JavaScript.Error stands for "no success"
dxrName="JavaScript.Error";
}
//--- 2 characters of givenName and sn
// UID: 2 characters of givenName and sn.
// if present more characters from givenName
if (gn != null && gn.length() > tries + 1) {
 dxrName = gn.substring(0, 2 + tries)+ sn;
} else {
if ( tries > 0) {
   dxrName="JavaScript.Error";
} else if (gn != null) {
   dxrName = gn + sn;
} else {
   dxrName = sn;
}
}
dxrName = dxrName + "";
```

`obj.logger.log("DBG", "dxrNameForAccount: tries="+tries);`

is a debug message that shows the progress of the script in the log files.

#### 2.13.4.4. Handling Property Dependencies

Properties with naming rules or JavaScript entries may depend on the values of other attributes. You must ensure that their values are

- Initially calculated after the properties on which they depend
- Recalculated each time the properties on which they depend have changed

Use the dependsOn attribute to accomplish this task. The syntax is:

```
<property name=propertyName dependsOn=propertyNameList .../>
```

where *propertyName* is the name of the dependent property and *propertyNameList* is a comma-separated list of local properties like *propertyName1, ..., propertyNameN*.

It is your responsibility to:

- Specify the property definitions in an optimized series
- Avoid circular dependencies, as DirX Identity does not check for them.

**Example:**

The example defines the dxrPrimaryKey attribute for a group.

```
<property name="dxrPrimaryKey"
 label="PrimaryKey (DN in TS)"
 type="java.lang.String"
 mandatory="true"
 multivalue="false"
 dependsOn="cn">
 <extension>
     <namingRule>
         <fixedValue value="cn="/>
         <reference baseObject="SvcGroup" attribute="cn" />
         <fixedValue value=",OU=Sales,"/>
         <reference baseObject="SvcGroup"
attribute="targetsystem.dxrOptions(GroupRootInTS)"/>
     </namingRule>
 </extension>
</property>
```

The dxrPrimaryKey attribute is recalculated when the cn of the group changes (dependsOn statement). If the cn has the value 'NewGroup' and GroupRootInTS has the value 'cn=groups' the primary key gets the value: 'cn=NewGroup,OU=Sales,cn=groups'.

### 2.13.5. Adding a New User Attribute

To add a new user attribute:

- Check to see if the attribute is supported by the directory server schema.
- Extend the schema if necessary.
- Extend the user object description.
- Define a default value or a naming rule (optional).
- Define a list of possible values either by the XML "tags" attribute or by supplying a proposal list (optional).
- Extend the property page description of the tab that should display the attribute.

### 2.13.6. Adding a New Role Attribute

To add a new role attribute:

- Check to see if the attribute is supported by the directory server schema.
- Extend the schema if necessary.
- Extend the role object description.
- Define a default value or a naming rule (optional).
- Define a list of possible values either by the XML "tags" attribute or by supplying a proposal list (optional).
- Extend the property page description of the General tab that should display the attribute.

## 2.14. Customizing Connectivity Objects

This section discusses how to:

- Use the propertypage to extend existing Connectivity object types with additional properties or to create completely new types.
- Create property page hierarchies
- Use "design mode" to hide properties at the object instance level that are not needed for this instance of the object
- Use connectivity default application object naming conventions

### 2.14.1. About Connectivity Object Descriptions

As previously discussed, DirX Identity Manager is almost completely configurable by XML descriptions. The XML descriptions define all of the specific parameters that are necessary for connected directory types or for agent types. DirX Identity's standard configuration uses these mechanisms extensively. These mechanisms can also be used for customer-specific connectivity objects. Some of the connectivity objects provided with the connectivity configuration can therefore be virtually extended without actually changing the connectivity configuration's LDAP data schema. You can extend the following connectivity objects:

- Jobs

- Connected directories
- Channels
- Bind profiles

When viewed with DirX Identity Manager, these objects consist of tabs and properties. These tabs and properties are defined as **propertypages** and **properties** at the XML level. By changing the **propertypage** and **properties** descriptions, you can control:

- The additional attributes to be displayed for this object type or the attributes to be hidden.
- The way in which these attributes are displayed (for example, as a text field or as a selection box).
- The tab location at which these attributes are displayed.
- The visual grouping of attributes.

You can also hide properties that are not relevant to a given object instance; see the section "Using Design Mode" for details.

Because the behavior of DirX Identity can be influenced enormously, you are only allowed to change parts of these XML descriptions. The following sections provide more details about customizing property page and property descriptions.

### 2.14.2. Customizing Connectivity Object Property Page Descriptions

Connectivity object property page descriptions follow the format described in "PropertySheet and PropertyPage Elements". Note that in the layout formats, you can use the shortcut "_SP" for "dxmSpecificAttributes" to make the definitions easier to read. Let's look at the LDAP connected directory configuration object to view its definition:

In the **Expert View**, open **Connectivity Configuration Data** → **Configuration** → **Connected Directory Types** → **LDAP** → **Object Descriptions** → **LDAP-ConnDir.xml** and click the **Content** tab. Select **New Window** from the context menu in the editor window.

Search for <propertysheet>. The first property page section looks like this:

```
<propertypage name="general"
helpcontext="default-conndir"
class="siemens.dxm.gui.components.PropertyPageGeneric"
title="Connected Directory"
layout="properties:dxmDisplayName,description,dxmVersion,
  dxmService-DN,dxmDirectoryType-DN,dxmDirectorySubtype,
  dxmAttrConfigFile-DN,dxmOpenCommand,dxmWizard-DN"
/>
```

The name of the property page is "general", and then a helpcontext is defined. Next, the

Java class that performs the display action is defined. The display title "Connected Directory" for this tab is defined with the title tag. The layout tag defines the sequence of properties, first dxmDisplayName is shown, then the description field and so on.

Verify this while looking at the MetaStore connected directory's first tab.

Another example in the same object shows group definitions:

```
<propertypage   name="script"
helpcontext="operattr"
class="siemens.dxm.gui.components.PropertyPageGeneric"
title="Operational Attributes"
layout="properties:dxmOprOperationalAttributes,
  dxmMasterName,[Base Nodes;_SP(base_obj),dxmTombstoneBase],
  [GUID Attributes;dxmGuidID,dxmLocalGUIDAttr,dxmGUIDAttr],
  dxmObjectClasses,dxmPhysicalDeletion"
/>
```

In this case, after two normal properties (dxmOprOperationalAttributes and dxmMasterName) two groups are defined. A group "Base Nodes" that consists of dxmSpecificAttributes(base_node) and dxmTombstoneBase and a group "GUID Attributes" that consists of dxmGuidID, dxmLocalGUIDAttr and dxmGUIDAttr. After the second group two ungrouped properties are displayed: dxmObjectClasses and dxmPhysicalDeletion.

Verify this while looking at the MetaStore connected directory's **Operational Attributes** tab.

### 2.14.3. Creating a Property Hierarchy

You can construct a configuration object hierarchy in which lower-level objects can inherit properties from higher-level objects. For example:

Generic Job → metaCP job

In this case, the metaCP job by default inherits all properties from the Generic Job, but you can redefine and hide properties at the lower level. The **superior** property in the **<object ...** section defines the object hierarchy.

### 2.14.4. Using Design Mode

Connectivity objects are defined via XML object descriptions, which can be extended via virtual object extensions. Sometimes it might not be necessary to use all of these defined attributes.

DirX Identity provides a **Design Mode** that helps you to adjust objects at the object instance level. This can be done at two levels:

- **Expert View** - Here you can hide all properties that are not relevant to this specific

object instance. These properties are then no longer visible at the Expert View and the Global View.

- **Global View** - Properties that make sense at the Expert View level can be hidden at the Global View level.

This two-level mechanism allows you to adjust your user interface in a very fine granular way:

- First, you define a new object type with an XML description.
- Second, you create a specific object instance and hide some properties that do not make sense at this specific object instance.
- Third, you use a tab of this object in a wizard. There you decide additionally that another property that is perhaps critical shall not be visible at the wizard level.
- Tabs where all fields are hidden are not displayed when design mode is switched off. This allows you to hide complete tabs for some object instances.

#### 2.14.4.1. Working with Design Mode in the Expert View

When you start DirX Identity Manager, **Design Mode** is switched off per default. We will make the Tombstone Base property visible again in the ADS connected directory to show how the mechanism works.

- Select the object **Connected Directories** → **Default** → **ADS** in the expert view and click the **Operational Attributes** tab. You will see about 4 attributes displayed.
- Now click the icon after the delete icon in the task bar to switch on the design mode. You will see more fields now, each one prefixed by a check box. The fields that were visible before are checked, all others are not.
- You can toggle the **Design Mode** switch several times to watch the effect.
- Now click **Edit**. Click the check box before the Tombstone Base field (it is checked now).
- Switch off design mode and you will see that the Tombstone Base field is now visible again.

#### 2.14.4.2. Working with the Design Mode in Global View

Here we will hide the Tombstone Base field at the global view level because we decide that this level of detail does not make sense at this level.

- Select global view. Switch off **Design Mode**.
- Click the **ADS** connected directory. Select **Configure** from the context menu. In the Operational Attributes tab, you will see the Tombstone Base field we made visible in the previous exercise. We decide to hide it at the global view level.
- To switch on design mode, the wizard must be closed. Click **Cancel** to close the wizard and switch on **Design Mode**.
- Open the wizard again and click the **Operational Attributes** tab. You will see some fields with grayed-out check boxes. These fields are defined in the expert view to be invisible. You cannot change the status here.

- Click the check box before the Tombstone Base field (it is not checked now). Click **Finish** to close the wizard.
- Switch off design mode. Open the wizard again. The field is no longer visible at the global view level.

## 2.14.5. Default Application Object Naming Conventions

The DirX Identity installation procedure installs a lot of configuration objects into the configuration database in the identity store.

All objects that belong to the default applications are located in **Default** folders. The next set of topics describes the naming rules for the following default application configuration objects:

- Connected directories
- Workflows
- Schedules
- Activities
- Jobs
- Channels

Although you are not restricted to using these naming conventions, we recommend that you follow them.

### 2.14.5.1. Connected Directory Naming Convention

A default application connected directory is located in the folder **Connected Directories** → **Default** and has the following naming convention:

*cd-type*

Example: ADS.

Intermediate connected directories always reside underneath a job folder. The naming convention is:

**Data**

Example: Data

When several intermediate connected directories are necessary under a job, other names can be freely defined (for example, SpecialData).

### 2.14.5.2. Workflow Naming Convention

A default application workflow is located in the **Workflows** → **Default** subfolder and has the following naming convention:

*cd-source***2***cd-target*_**mode**

Example: ODBC2Ident

When additional information for the mode must be defined, it should be added as one word and be separated with an underline character (for example ODBC2Ident_Sync).

The *mode* parameter is optional.

#### 2.14.5.3. Schedule Naming Convention

A default application schedule is located in the **Schedules** → **Default** sub folder and has the following naming convention:

*cd-source***2***cd-target*_**mode_time** (= *workflow*_**time**)

Example: ODBC2Meta_Noon

*time* can be any keyword that indicates a special date such as Noon, Midnight, Sunday, and so on.

The *mode* parameter is optional.

#### 2.14.5.4. Activity Naming Convention

A Default Application activity is always located under the corresponding workflow object and has the following naming convention (the same naming convention as for jobs):

*cd-source***2***cd-target*_**mode_agent** (= *workflow*_**agent**)

Example: ODBC2Ident_MetaCP or ODBC2Ident_ODBCAgent)

The *mode* parameter is optional.

#### 2.14.5.5. Job Naming Convention

A default application job is located in the **Jobs** → **Default** sub folder and has the following naming convention (the same naming convention as for activities):

*cd-source***2***cd-target*_**mode_agent** (=*activity*)

Example: ODBC2Ident_MetaCP or ODBC2Ident_ODBCAgent)

The job includes the configuration (ini- and tcl-files), deltainfo, data channels and mapping information.

These items are named:

**Tracefile** - for trace files

**Reportfile** - for report files

**Export INI file** - for export INI files

**Import INI file** - for import INI files

**Notify** - for a notify object

When several files of a specific type are needed you can choose an appropriate name freely (for example NotifyNotOK or NotifyData).

#### 2.14.5.6. Channel Naming Convention

A default application channel has the following naming convention:

*workflow*

Example: ODBC2Ident

For intermediate connected directories, the channels are named:

- **InData** for channels which read from the connected directory
- **OutData** for channels which write to the connected directory

When more channels are needed you can choose other names freely.

## 2.15. Creating Component Descriptions

DirX Identity workflows consist of a rather limited set of activity types.But each activity and job configuration has a lot of diversity: optional components, different implementations (for example, a large set of connectors), and other differences.Here are some illustrations:

- In order to add activities to realtime workflows, both the Global and Expert Views of DirX Identity Manager's Connectivity view need information about the activity type; for example, the number and type of ports and the functionality of the activity.
- Ports of an activity must be connected to ports of other activities or especially to connected directories.This determines the connector type.Although the connector configurations have some options in common, they differ in others.
- Filters may intercept the job controller and each connector.We have different types of filters (delta, crypto), each with different configuration properties.
- Error and notification handling may also differ: in most cases, it is to send an email, sometimes to send a SOAP Request.

Using the object description approach to manage these issues would lead to an explosion of object descriptions for activities.Thus we need an alternative approach: meta data (component descriptions).

The sections in this chapter describe:

- How component descriptions work
- The general format of a component description
- Elements and attributes for structure and data-handling
- Elements for presentation handling

### 2.15.1. How Component Descriptions Work

Component descriptions describe the configuration of components such as connectors or activity types. Component descriptions must be associated with an object description, either directly or indirectly. At runtime, the component descriptions are merged with their parent object descriptions and thus form a new object description.

Component descriptions can:

- Extend a parent object description
- Extend another component description
- Implicitly extend a parent or component description. This type of component is included into a set of other object or component descriptions, but it does not know them a-priori and hence cannot reference them explicitly.

Here is a typical usage scenario:

An administrator wants to add a notify activity to a workflow definition. DirX Identity supports a few notification implementations such as email. The administrator either selects an existing template from a palette (that is, a template for an email notification) or first selects an abstract notification and selects an implementation type for the job.

The presentation of the activity must change depending on the selected job type. Typically, each type needs its own specific options and default values. At least one of these is the implementation class name, which is fixed for each type. When the administrator selects the job type, he actually selects the appropriate component description. The DirX Identity Manager presentation and the underlying XML configuration must then be changed on the fly. If another type has already been selected, the data for the old one must be removed and replaced with the default values of the new type. As a part of this procedure, the component description of the selected component is merged into the object description of the parent object. New properties must be added and old ones removed; the property pages also must be updated accordingly.

Here is an example for a real-time activity:

The typical Provisioning activity needs a connector to the Identity domain, one to the target system and a set of standard connections for input events and responses (not in scheduled activities), error, audit and notifications. While the configuration for the standard ones is generally stable, the connector for the target system must change with the target system type. Therefore, the configuration needs a property where the connector type is selected and stored. The DirX Identity Manager must present the connector options on some tab sheet (that is, property page) and store them at the appropriate location within the XML configuration document. Some of the connector options can be derived from the selected connected directory or bind profile; for example, address and port can be taken from the connected directory service as soon as the connected directory is known. Other options are independent and must be entered explicitly by the administrator. In this case, the component description must specify that address and port values can be derived from the connected directory and needs a formula for getting and calculating the value. It does not store the calculated values in the document. Instead, it evaluates the formula when the workflow definition is downloaded to the Identity server.

The administrator may want to insert one or more filters between the join engine and the connector. The Manager supports him by offering the list of filters and the administrator stores them in a multi-value property. In most cases, this cannot be an LDAP attribute, and must instead be stored as part of the XML document. When the reference to the new filter is stored, the Manager needs to extend the object description with the filter's component description and allow the administrator to enter the options that the filter requires. Most often this will be presented as a new dialogue that just shows the filter options.

To achieve this, the Manager needs to know that there is a multi-value property for filters, where to store the filter configuration within the XML (that is, parent node and sequence), in which XML format to store the options; for example, XML attributes of the filter root element, simple elements named as the option name, name/value property elements or alike. All of this information must be specified in the component description.

### 2.15.2. Component Description Format

The root element of a component description is <componentDescription>. It is basically separated in a set of <element> sections followed by a <presentationDescription> section. Very similar to an XML schema, the <element>s describe the structure of the component's XML configuration. You can next <element>s, usually by references such as the following:

```
<element name="connector">
…
          <element ref="connection" minOccurs="1" maxOccurs="1"/>
</element>
<element name="connection" …/>
```

In this example, the configuration element <connector> contains exactly one <connection>.

As with XML schemata, the attributes 'minOccurs' and 'maxOccurs' specify the minimum and maximum number of instances. Default values are "1".

The <element>s contain a list of <property>s, which describe the data issues of properties, such as their data type, single- or multi-value, mandatory, default values and their description.

The <presentationDescription> element and its sub-elements deal with the presentation of the properties defined in the component description. Their sub-elements specify their distribution to property pages, their labels, editors and other presentation-related information.

### 2.15.3. The Property Element

The <property> element inherits the attributes from the object description that deal with non-presentation aspects: name, type, mandatory, multivalue.

The new attribute xmlNotation specifies the XML format of the property in the XML document underneath its parent element. If it is missing, we assume the property is an

LDAP attribute (specifying a defaultvalue means initializing the LDAP attribute with this value). The following values are allowed:

- xmlAttribute: The property name is taken as the name of an XML attribute like this: propertyname="value"

  this value can only be applied to single values.

- SimpleElementProperty: The property name is taken as the XML element name. The value is the content of the element. The additional XML attribute "elementname" changes the name stored in XML. By default the property name is used. Elements with the same name may occur several times. For example:

```
<prop>value1</prop>
<param>value1</param>
```

- NestedElementProperty: The property name is again a XML element. The values are also the content of XML sub-elements. Note: this allows multi-values. The additional XML attribute "elementname" changes the name stored in XML. By default, the property name is used as elementname. If you specify an element name then the name is interpreted as xml attribute name(search criteria for reading) .The additional XML attribute "valueElement" changes the value tag stored in XML. The default is “value”.

Here is a default sample for a property "param":

```
<param><value>this is a value</value></param>
     name="to" xmlNotation="NestedElementProperty"
elementname=”property”:
<property name=”to”>
    <value>admin@example.test</value>
    <value>sample@example.test </value>
</property>
     name="to" xmlNotation="NestedElementProperty"
elementname=”property" valueElement =”data”:
<property name=”body”>
    <data>admin@example.test</data>
</property>
     name="activitiesToBeApproved "
xmlNotation="NestedElementProperty":
<activitiesToBeApproved>
    <value>enterAttributes</value>
</activitiesToBeApproved>
     name="parentNodes" xmlNotation="NestedElementProperty"
```

```
valueElement="parentNode">:
<parentNodes>
    <parentNode>cn=Users,cn=My-Company</parentNode>
</parentNodes>
```

- NameValueProperty: The property name and value are stored as XML attributes "name" and "value" with an XML element <property>. You can change the element name by specifying the additional attribute 'elementname="OtherElementName"'. By default “property” is used. This is only appropriate for single-valued properties.

```
name="propname" xmlNotation="NameValueProperty"  :
<property name="propname" value="value1"/>
name="dxrRole" xmlNotation="NameValueProperty"
elementname="param":
<param name="dxrRole" value="false"/>
```

- DsmlProperty: denotes the DSMLv2 style of attribute value notation.

```
<attr name="prop">
<value>value1</value>
<value>value2</value>
</attr>
```

- component: This defines that this is a generic property. It is described by another component description. To specify this component description use the attribute component. Usually the type is SvcGenericCompDescObject. You need to define a special editor to handle these generic objects.

```
<property name="attributes"
type="siemens.dxm.storage.component.SvcGenericCompDescObject"
xmlNotation="component" component="attribute" multivalue="true">
</property>
…
<propertyPresentations>
<property name="job/attributes/attributes" readonly="false"
multivalueeditor="siemens.dxr.manager.controls.requestworkflow.Jnb
GenericCompDesc"
editorparams="compDescName=attribute;
properties=name,description,mandatory">
    <label>Attributes</label>
```

```
</property>
</propertyPresentations>
```

#### 2.15.3.1. Default Values and Tag Providers

Use value elements to define a default value or a list of tags that represents the valid values for the property. If exactly one value element is specified, this is the default value for this property. If you want to define a list of possible values specify one value element per possible value. DirX Identity Manager displays a combo-box with all specified values if it finds this property in a propertyPresentation. Specify the default value by adding the attribute default=”true”. This is equivalent to the tags construct in an objectDescription.

Here is an example for a property (**sign**) with boolean syntax and the default value **false**:

```
<property name_"sign" xmlNotification="SimpleElementProperty"
        type="java.lang.Boolean"
        elementname="sign">
    <value>false</value>
</property>
```

In the following example the property **execution** is specified with the default value **parallel** and a second possible value **sequential**:

```
<property name="execution" xmlNotation="SimpleElementProperty"
        elementname="execution">
    <value default="true">parallel</value>
    <value>sequential</value>
<property>
```

### 2.15.4. The CDATA Attribute

The SimpleElementProperty and NestedElementProperty support an additional "cdata" attribute. Specifying cdata="true" means the value is stored as a CDATA element. This attribute is only supported for SimpleElementProperty and NestedElementProperty.

```
name="code" xmlNotation="SimpleElementProperty" cdata="true":
<code><![CDATA[/*** ***/]]></code>
```

### 2.15.5. Parent-Child Elements and Attributes

Component descriptions extend other component or object descriptions. Consequently, a component description must specify where the child component is to be inserted into the

parent. Properties of the parent influence the selection of the child type and even the properties of the child. On the other hand, a child component can influence properties of the parent. The <property>, <parentproperty>, <component>, <reference> and <mapto> elements and some XML attributes are used to define these relationships.

#### 2.15.5.1. Inserting Child Components

The <component> may be inserted underneath an <element>. It references a <property>, which itself holds the link to a component description. By setting this property, the customer selects a child component that is inserted into the parent. The following example defines a connector component to be inserted underneath a <port> element:

```
<element name="port">
 <component refProperty="connectorTS" minOccurs="1" maxOccurs="1"/>
 <properties>
      <property name="connectorTS"
           mandatory="true" xmlNotation="xmlAttribute"
           insertbelow=".">
 </property>
```

The <component> refProperty attribute references a property by name. The referenced property holds the reference to a component description (the name of the component description as string) and has to be stored underneath the parent element in the format that the attribute xmlNotation prescribes.

The attributes minOccurs and maxOccurs limit the number of occurrences of the reference. In the above example, only one <connectorTS> is allowed. The default value for each of them is "1". The value "unbounded" for maxOccurs allows an unlimited number of values.

The new attribute insertbelow of the <property> specifies the XML element of the parent, underneath the component's configuration must be inserted.

You can specify a list of components of the same type (maxOccurs="unbounded") and to specify several components of different type as in the following sample:

```
<component refProperty="connectorDesc" minOccurs="1" maxOccurs="1"/>
<component refProperty="filterDescription" minOccurs="0"
maxOccurs="unbounded"/>
<properties>
 <property name="connectorDesc" type="SvcComponentReference"
      mandatory="true" xmlNotation="xmlAttribute"
insertbelow="./port">
 </property>
 <property name="filterDescription" type="SvcComponentReference"
```

```
      mandatory="true" xmlNotation="SimpleElementProperty"
insertbelow="./port">
 </property>
```

Here we have a component "connectorDesc" and another component " filterDescription ". Note that a list of filterDescription's can be inserted underneath a <port> element.

#### 2.15.5.2. Setting Parent Properties from a Child Component

In some cases, the inserted child component governs the setting of a parent property. For example, in a notification activity, the class name of the job implementation must be stored outside the component configuration as a sub-element of the <job>, but is determined by the selected component:

```
<job>
 <component>mailNotification</component>
 <class>com.siemens.idm.rqjobs.NotifyClient</class>
 <notification>
      <data>
           <from>admin@example.test</from>
```

This relationship must be reflected in the parent and child component descriptions. The parent description describes the property as "abstract" and the child defines a <parentproperty> and supplies a default value.

Here is the parent snippet:

```
<element name="job">
 <component refProperty="component" minOccurs="1" maxOccurs="1"/>
 <properties>
 <property name="component"  …/>
 <property name="class" abstract="true" …/>
```

Here is the child snippet:

```
<parentproperties>
 <parentproperty name="class">
      <value>XXXcom.siemens.idm.rqjobs.NotifyClientXXX</value>
 </parentproperty>
```

Note that the property in the child definition has to reflect the relative path in the parent beginning with the parent's root node(here the job element) !

In such a scenario, the child must make assumptions about the parent. In this case, it must know that the parent has a property named "class".

#### 2.15.5.3. Reading Parent Properties from a Child Component

Often the child will re-use properties of the parent for value calculation of its own properties. In this case, the child needs a reference to the parent property as in the following snippet:

```
<parentproperties>
 <reference name="connectedDirectory"/>
</parentproperties>
```

Such a property is typically stored as a resolution variable. It can now be used to calculate own property values, as shown in these snippets:

```
${DN4ID(connectedDirectory)@dxmService-DN@dxmAddress}
${ssl} == 'true'
```

If a parent contains two children of the same type, which both carry references to a parent property, we need a mapping of property names. Suppose a <job> has two <port>s: one for the Identity domain, one for the target system. The <job> will have a property for the connected directory of each of the <port>s. It may name them "connectedDirectoryTS" and "connectedDirectoryID". Each connector component will find the value for its address property by navigating from the connected directory to the service (or system) entry. Thus it needs access to the parent property, which is from its point of view the connected directory.

We must have a way to reference the corresponding connected directory from the child. This is done with the <mapto> element. The name mapping is done in the parent, which is the only one that knows the different children. The <property> element gets a new child element <mapto>, which specifies the child component and the property name in the child to which it is associated:

```
<property name="IdentityDomain" type="DN" …>
 <mapto
property="port[@name='IdentityDomain']/connector/connectedDirectory"/
>
</property>
<property name="connectedDirectoryTS" type="DN"… >
 <mapto property="port[@name='TS']/ connector/connectedDirectory"/>
</property>
```

The "property" attribute takes the name of the property in the child component. Note that the <mapto> element must select one child element from within a list with the same element name! It identifies the corresponding child by using the child's "name" attribute.

### 2.15.6. Presentation Description Element

A component description specifies its presentation issues in the element <presentationDescription>. It can be compared with the <propertysheet> of an object description. The sub-elements of the <presentationDescription> are merged into the <propertysheet> of its parent.

A <presentationDescription> can contain the following sub-elements: <propertypage>, <propertyPageExtension> and <propertyPresentations>.

### 2.15.7. Property Page Element

A <propertypage> has exactly the same attributes as the corresponding element in the object description: name, title, class, layout and insertafter. It is merged with the parent propertysheet the same way as already realized within the object description.

### 2.15.8. Property Page Extension Element

A <propertyPageExtension> element extends a parent property page.

The attribute "extendsPage" names the parent page to be extended. This is optional. See below how to insert it into an unknown parent.

The attribute "insertafterattribute" specifies the parent attribute after which the child properties have to be inserted. It is also optional to allow for unknown parents.

The "layout" attribute is mandatory and lists the properties to be displayed in this page. Note that a leading "properties:" string is ignored, since it depends on the parent specification, if the properties are shown in the denoted sequence or not.

How to display properties of several child elements with the same name? Here we use again (see the <mapto> element in "Reading Parent Properties from a Child Component") the XPath notation as shown in the following example:

```
layout="port[@name='TS']/connectedDirectory, ..."
```

### 2.15.9. Property Presentation Element

The element <propertyPresentations> contains a list of <property> sub-elements. They correspond to the <property> elements of the data section and describe the presentation aspects. This <property> element contains the well-known attributes "name" (the reference to the corresponding data description), "readonly", "editor", multivalueeditor", and "editorparams". "label" and "tagprovider" are specified as separate elements. Here is an example:

```
<property name="connector"
        editor="siemens.dxr.manager.controls.MetaRoleJnbComboBox">
    <label>Connector</label>
    <tagprovider
class="siemens.dxr.service.tags.ComponentDescriptions"
            dxmComponentType="connector"
    />
</property>
```

### 2.15.10. Initial Content Element

Although the activity and job schemata allow a wide range of definitions, most of our activity definitions assume a certain structure. As an example, a provision activity assumes a list of channels (in, response, error, ...), a framework job configuration with the corresponding connectors for the activity channels plus ports for the Identity domain and the target system. It makes life easier for our GUI and for customers, if we deliver initial content for our activity definitions. They set up the expected structure and give default (or fixed) values for some of the properties. It's up to the object and component descriptions to allow only changes within the given framework structure.

Thus a component description can contain an <initialContent> element that represents the default configuration. Look at the templates for the password synchronization workflows, which can be treated as initial content.

Initial content is defined at the element level. During initialization, all initial content elements are evaluated from the root element down to the leaves. The initialization of the properties with their default values is performed after the evaluation of initial content. So properties default values may override initial content.

Here is the component description:

```
<element name="activity" abstract="true">
<initialContent>
    <addElement1 attr="avalue"/>
    <addElement2>
         <property name="name" value="aname"/>
    </addElement2>
</initialContent>….
```

Here is the created XML fragment:

```
<activity basicType="${DN4ID(THIS)@dxmType}"
name="${DN4ID(THIS)@dxmDisplayName}"
```

```
subType="${DN4ID(THIS)@dxmActivityType}">
<addElement1 attr="avalue"/>
<addElement2><property name="name" value="aname"/></addElement2>
<retrylimit>0</retrylimit>…
```

In the previous example, the initialContentElement creates the elements addElement1 and addElement2. The other parts like retrylimit are specified in another component description extending the activity element.

# 3. Customizing Policies

This chapter describes how to customize policies:

- Customizing access policies

## 3.1. Customizing Access Policies

DirX Identity allows protecting objects via access policies.Per default access control and access policies are defined for a set of standard objects.You can define access control for

- Objects in DirX Identity that are already defined but are not yet protected via access policies.
- Your own defined custom objects.

To enable access control for an object type, perform these steps:

- Adapt the object description for the object type.
- Create the necessary access policies.

### 3.1.1. Adapting Object Descriptions

To enable an object type for access control, add this attribute definition to the object element:

```
accesscontrol="true"
```

This enables access control for this object type. It is checked by the service layer and the corresponding access policies.

### 3.1.2. Creating Custom Access Policies

If an object type is enabled for access control, you need to set up the corresponding access policies. Otherwise access to this object type is no longer possible through Web Center, Business User Interface or Manager.

Due to the fact, that the list of object types that is supported by default does not contain all available object types (for example dxrAuditPolicy is not part of the list), use this procedure to set up additional policies:

- Create a new access policy and configure it for example for object type User.
- Click this new access policy in the tree view and perform **Goto DataView**.
- Edit the object and change the dxrObjectType attribute to the required value (for example dxrAuditPolicy or MyPrivateObject).
- Save the object.

Whenever you change the operation of the access policy you must setup the object type value again.

# 4. Customizing Object References

This chapter describes how to customize:

- References for Java-based workflows
- References for Tcl-based workflows

## 4.1. References for Java-based Workflows

You can use references in the XML content definitions of all Java-based workflows.Java-based workflows consist of request workflows (Provisioning view group) and real-time workflows (Connectivity view group).References for Java-based workflows can occur anywhere in the XML content.To avoid constant repetition of definitions and also to be able to use variables from higher level sections in lower level sections you can use resolution variables.Resolution variables are stored in the section "resolutionVariables".Only one "resolutionVariables" section is permitted per XML node.

### 4.1.1. Reference Definition

A reference is a variable that is resolved and replaced during runtime by its value.

The syntax of a reference is:

**${DN4ID(***variable***)@***attribute***@***attribute***@...@***attribute***}**

where *variable* can be:

- **THIS** - the actual object. Don't define a resolution variable named THIS. It will replace the predefined THIS variable.
- **SUPERIOR** - the superior object. Don't define a resolution variable named SUPERIOR. It will replace the predefined SUPERIOR variable. The following example shows how Superior Domain Node is referenced from the Java-based Server object:

  ${DN4ID(SUPERIOR)@dxmSsl}

- *resVar* - the name of a resolution variable.
- superior-DN - the superior-DN meta attribute. Use this as an alternative to the predefined SUPERIOR resolution variable. The following example shows how to navigate two steps up:

  grandparentOfChanelfolder="${DN4ID(THIS)@dxmSpecificAttributes(channelparent)@superior-DN@superior-DN@dxmDisplayName}

### 4.1.2. Using Simple References

Here are two examples for references:

The example:

${DN4ID(THIS)@dxmSpecificAttributes(ssl)}

Inserts the value of the specific attribute ssl of the actual object variable.

The example:

${DN4ID(mappingscript)@dxmContent}

Inserts the value of the dxmContent attribute of the mapping script resolution variable.

### 4.1.3. Access to Multi-value Attributes

Specifying $\{DN4ID(THIS)@description} returns always the first value of the description attribute. For multi-value attributes you can get all the values as comma separated list.

Specify the following constructs:

```
<description>${DN4ID(THIS)@description,mv}</description>
<resourcefamily>${DN4ID(THIS)@dxmSpecificAttributes(resourceFamily),m
v}</resourcefamily>
```

The output may look like:

```
<description>desc2,Error Activity</description>
<resourcefamily>LDAP,ABC</resourcefamily>
```

If mv is specified and only a single value is present the result is:

```
<description>Error Activity</description>
<resourcefamily>LDAP</resourcefamily>
```

### 4.1.4. Resolution Variables

Resolution variables are defined in the section <resolutionVariables> with this syntax:

```
<resolutionVariables>
<resolutionVariable name="..." value="..." .../>
<resolutionVariable ...
</resolutionVariables>
```

Each resolution variable represents exactly one LDAP entry (the LDAP entry is specified by an "identifying attribute")

Attributes of a resolution variable include:

**name** - the variable name. Use this name to reference the variable.

**identAttr** - the identifying attribute. This attribute defaults to **cn**. Use **dn** to identify the variable via the DN.(internally a baseobject search is performed).

**value** - the value of the identifying attribute that identifies the entry.

**baseVar** - the name of a resolution variable that represents the root. Under this root, the object must be unique. This attribute defaults to **dxmc=DirXMetahub**.

**objectclass** - the object class of the LDAP entry. This attribute defaults to ***.

Technically, these attributes result in an LDAP search to get the corresponding entry.

Here are some examples of resolution variable that are used in the workflow element of a real-time workflow:

```
<resolutionVariables>
<resolutionVariable
name="topicSet"
objectclass="dxmTopic"
value="TOPIC_PROVISION_TO_TS"
identAttr="dxmTopicName"
/>
<resolutionVariable
name="activities"
baseVar="THIS"
objectclass="dxmIDMActivityDefinition"
multivalue="true"
/>
<resolutionVariable
name="workflow"
equalsVar="THIS"
/>
</resolutionVariables>
```

The variable topicSet gets its value from the attribute dxmTopicName. The relevant object is searched under the node dxmC=DirXmetahub with the filter objectclass="dxmTopic" and value="TOPIC_PROVISION_TO_TS".

The activities variable is defined as a multivalue variable. It searches under the actual object (THIS) all objects with objectclass="dxmIDMActivityDefinition". This list can be used to load later on the XML content of all these objects to include it into the current XML definition.

#### 4.1.4.1. Multi-value Resolution Variables

If a resolution variable may represent more than one LDAP entry (an array of entries) specifies the attribute "multivalue". This feature is used for the **foreach** processing instructions.

In the following example the resolution variable **activities** represents all LDAP entries with the objectclass **dxmIDMActivityDefinition** under the base object **THIS**:

<resolutionVariable name="activities" baseVar="THIS"
objectclass="dxmIDMActivityDefinition" multivalue="true"/>

To access the entries use the **[]** operator specifying the index of the array. The array index starts with 0. The following example accesses the second entry of the resolution variable **activities** and returns the value of the dxmContent attribute:

${DN4ID(activities)[1]@dxmContent}

If you want to sort the result use the attribute

**sortAttribute="***attribute***"**

where *attribute* specifies the attribute that is used to sort the result. Only one attribute can be specified. Sorting is always performed in ascending order. This feature is used with framework filters. To keep the sequence constant a sequence number is specified for every filter. In the following example the dxmSequenceNumber attribute is used to sort the result:

sortAttribute="dxmSequenceNumber"

### 4.1.5. Meta Tags

Meta tags are used to avoid definition-specific Java implementations. These tags are only evaluated when the XML fragments are expanded and exported into the workflow server, and instruct the configuration manager to include content from LDAP attributes.

Initial content meta tags are stored as XML processing instructions <?IDMCONF ... ?>. This notation indicates to the XML parser that these elements are to be evaluated by the application. "IDMCONF" is an identifier for the Identity Manager Configuration.

#### 4.1.5.1. Include LDAP Attribute Tag

With the "INCLUDELDAPATTRIBUTE" operation, the content of an LDAP attribute of the same or another LDAP entry can be included:

```
<?IDMCONF INCLUDELDAPATTRIBUTE REF="${DN4ID(channel)@dxmContent}"  ?>
```

The "REF" attribute expects a reference to the LDAP attribute. Note that the content of the included LDAP attribute has to be well-formed XML! It again has to be expanded the same way as the parent.

The attribute may be multivalued. In this case, the include operation is performed in a loop for each value.

You can use an extended notation:

```
<?IDMCONF INCLUDELDAPATTRIBUTE REF="${DN4ID(channel)@dxmContent}"
PATH="template" ?>
```

If PATH is not present the whole XML document is exported.

Specifying PATH means to export the given path and its children. In this example, the element "template" and it's children are exported.

Path is relative to the root element and path elements are separated by '/'.

Example:

```
<content contentType="ReplacementAttributeDefinition">
 <template mappingType="constant" match="description">
     <value>Test of the constant template</value>
 </template>
</content>
```

To export the value element specify PATH="template/value"

#### 4.1.5.2. Loop Tags

The operation "FOREACH" and "ENDFOREACH" allow for including values of several LDAP entries defined by a multi-value resolution variable in between a loop:

<?IDMCONF FOREACH channel IN ${channels} ?>

<this will be included for each value of variable ${channels} />

<?IDMCONF ENDFOREACH ?>

The content will be included for each value of the loop variable (here: ${channels}, which is assumed to be multi-value). This construct allows to follow multi-value LDAP references and include the content of a set of referenced LDAP attributes as part of an enveloping XML construct.

The included content is expected to be well-formed XML!

In the following example the multi-value resolution variable **activities** represents all LDAP entries with object class **dxmIDMActivityDefinition** under the base object **THIS**. In this example **THIS** represents a workflow object, for example a realtime workflow object with usually two activities: join and error.

For each LDAP entry (represented by the variable **activity**) in the list of all LDAP entries represented by the variable **activities**, for example returned by a search operation, the dxmContent attribute is expanded via the INCLUDELDAPATTRIBUTE tag, once for the join activity and once for the error activity.

```
    <resolutionVariables>
          <resolutionVariable name="activities" baseVar="THIS"
objectclass="dxmIDMActivityDefinition" multivalue="true"/>
    <activities>
…
 <!-- insert activities from LDAP entries here! -->
 <?IDMCONF FOREACH activity IN ${activities}
content=<content><?IDMCONF INCLUDELDAPATTRIBUTE
REF="${activity@dxmContent} ?&></content>  ?>
</activities>
```

#### 4.1.5.3. Conditional Include Tag

Use the **if** construct to insert a configuration section depending on an attribute of a resolution variable. The body of the **if** branch must be a syntactically correct XML structure (for example a complete XML section/tag).The **if** construct is like the XSLT **if** construct: there is no **else** branch.

The syntax is as follows:

**<?IDMCONF if-begin test="***resolutionVariable operator* **'***value***'"?>***
***xmlBopy***
<?IDMCONF if-end ?>**

where

- *resolutionVariable* is a resolution variable
- *operator* is one of the following operators:

  **==** string comparison for equality ignoring case

  **!=** string comparison for non equality ignoring case
- *value* is a constant text value
- *xmlBody* is the conditional XML body specification.

In the following example, the connection's support for SSL depends upon the specific attribute ssl:

```
<?IDMCONF if-begin test="${ssl} == 'true'"?>
${DN4ID(connectedDirectory)@dxmService-DN@dxmSecurePort}}
<?IDMCONF if-end ?>
```

```
<?IDMCONF if-begin test="${ssl} != 'true'"?>
${DN4ID(connectedDirectory@dxmService-DN@dxmDataPort}
<?IDMCONF if-end ?>
```

With the component descriptions, it can be used the first time for individual properties, not just for complete XML elements.

#### 4.1.5.4. Include Attribute Mapping Tag

Attribute mappings in data-synchronization workflows are formulated as <attrMapping>, <idMapping> or <postMapping> within a XML <mappingDefinition> element. Here is an example:

```
<attrMapping name="sAMAccountName" mappingType="direct">
<value>dxrName</value>
</attrMapping>
```

If the mapping is realized via Java code (mappingType="java"), the Java source code and the compiled unit are placed in extra LDAP entries. They lie underneath the channel entry identified by their attribute name. The compiled unit is found in the LDAP attribute "dxmCompiled" (see /1/). Exceptions: For <idMapping> a fixed Name “dxmIDMapping” and for <postMapping> a fixed name “dxmPostMapping” is used to identify the LDA entries ( because here is no attribute name available).

Such a construct makes the downloading more complex. Most of the XML can be taken as it is. Only if the mapping Type is "java", the corresponding AttributeMapping entry has to be read, and the content of the LDAP attribute "dxmCompiled" inserted as <code> sub-element. Here the resulting element:

```
<attrMapping name="sAMAccountName" mappingType="java">
<code>RMO8c3Nlb…GRvcmY=</code>
</attrMapping>
```

Virtually the same is needed for identification mapping <idMapping> ...

```
<idMapping type="urn:oasis:names:tc:SPML:1:0#DN"
mappingType="direct">
<value>dxrPrimaryKey</value>
</idMapping>
```

... and <postMapping>.

This transformation cannot be formulated with the above processing instructions. Therefore we introduce a new one named "INCLUDEATTRMAPP

```
<?IDMCONF INCLUDEATTRMAPPING REF="${DN4ID(THIS)@dxmAttrMapping}" ?>
```

The "REF" attribute expects a reference to the LDAP attribute, which holds one <mappingDefinition> element. The implementation has to replace the <value> element of all java-based mappings in <attrMapping>, <idMapping> and <postMapping> by the <code> element, which holds the compiled Java code.

The content of the included LDAP attribute has to be well-formed XML! It again has to be expanded the same way as the parent.

#### 4.1.5.5. Include Specific Attributes as Property List Tag

DirX Identity holds some attributes as specific attributes at various LDAP entries. Important entries for the real-time workflows are: root node, configuration entry "dxmC=Configuration,dxmC=DirXmetahub", each connected directory and each channel. They are stored in the format "attributeName value".

Mapping in the real-time workflows makes use of a list of environment attributes. They are collected from several entries (listed above) and stored in the following XML format:

```
<property name="…" value="… "/>
```

The following processing instruction reads these specific attributes and stores them in this XML format:

```
<?IDMCONF INCLUDESPECIFICATTRIBUTE
REF="$\{DN4ID(THIS)@dxmSpecificAttribute}" ?>
```

The "REF" attribute expands to an LDAP attribute in some entry, which contains the attributes in specific format.

#### 4.1.5.6. Insert Conditional Attributes Tag

The value of some attribute may depend on the value of another property. For example, the target system port usually depends on the SSL flag of the bind profile. The insertCondAtt operation allows you to include referenced content on a condition. Here is an example:

```
<connection pagedRead="false" pagedTimelimit="0" pagesize="100"
password="${DN4ID(bindprofileTS)@dxmPassword}"
server="${DN4ID(targetSystem)@dxmService-DN@dxmAddress}"
type="${DN4ID(targetSystem)@dxmDirectoryType-DN@dxmType}"
```

```
user="${DN4ID(bindprofileTS)@dxmUser}">
<?IDMCONF insertCondAtt attName="port"
test="${DN4ID(bindprofileTS)@dxmSSL} == 'true'"
true="${DN4ID(targetSystem)@dxmService-DN@dxmSecurePort}"
false="${DN4ID(targetSystem)@dxmService-DN@dxmDataPort}" ?>
<?IDMCONF insertCondAtt attName="ssl"
test="${DN4ID(bindprofileTS)@dxmSSL} == 'true'"
true="true"
false="false" ?>
</connection>
```

In the example shown here, some attributes of the connection are fixed (pagedRead, password, ...). The attributes port and ssl depend on the dxmSSL attribute of the bindprofile. The instruction inserts an attribute of the given name to the enclosing element (here connection). If the test expression matches then the attributevalue is set to the one specified after “true=” otherwise the value of false is taken. If the expression can’t be evaluated as the referenced attribute isn’t present, false is assumed.

The example shown here can be resolved to:

```
<connection pagedRead="false" pagedTimelimit="0"
pagesize="100"
password="{SCRAMBLED}QXNMJR50BlNzIXQsBCA3"
server="server11" type="ADS" user=”metatest@mh4.mycompany.de”
port="636" " ssl="true“/>
```

### 4.1.6. Reference Scope

References defined in a ResolutionVariables section of Node X have the scope "X and all children of X".

Variables defined in a ResolutionVariables section of Node Y under Node X are valid in Node Y and all of its children.

If variable "a" is defined in Node X and Node Z, where the structure looks like this:

```
<x>
(definition of resolution variable a)
<y>
<z>
  (another definition of resolution variable a)
</z>
</y>
```

```
</x>
```

In this case, the first variable a is used in x and y. The second definition is only valid in z. There is no way to reference the variable "a" of Node X in or under Node Z.

### 4.1.7. Reference Resolution Errors

In the next example the dxmUrlPrefix and dxmSpecificAttributes(systemid) attributes have to be resolved:

```
<connection
type="Soap"
url="http://${DN4ID(connectedDirectoryCN)@...@dxmUrlPrefix}
.${DN4ID(bindprofileCN)@...@dxmType}">
<property
name="systemID"
value="${DN4ID(connectedDirectoryCN)
@dxmSpecificAttributes(systemid)}"
/>
...
```

If a resolution expression cannot be resolved, a warning is generated and by default the attribute is removed completely.

```
<connection type="Soap">
<property name="systemID" />
```

In the DirX Identity Manager the attribute name is inserted and surrounded with '#'.For example:

```
<connection
type="Soap"
url="http://machineA:1111/###@dxmUrlPrefix###.Notes">
<property name="systemID"
 value="###@dxmSpecificAttributes(systemid)###"/>
```

## 4.2. References for Tcl-based Workflows

You can extend connectivity configuration objects that relate to Tcl-based workflows with additional attributes (called "properties" in the DirX Identity Manager usage documentation) so that they are easily accessible for editing using DirX Identity

Manager.These attributes and also the attributes from the standard DirX Identity schema can be referenced in command lines or text files (Ini files, Tcl script files).Using this generic mechanism guarantees consistency over the whole DirX Identity system.

The syntax for a reference description is XML-like:

```
<?_reference_/>
```

References can be simple or complicated constructs.If references retrieve values, then these values can contain references again.This allows you to set up a hierarchy of references.

Because references can be nested, DirX Identity does not allow more than 20 levels to avoid endless loops.

Reference description types include:

- Single-value references
- Multi-value references
- Reference variables

This section describes these topics as well as the following reference description features:

- Reference description abbreviation
- Relative references
- File name handling

## 4.2.1. Single-Value References

Single-value references include:

- Fixed references
- Variable references

#### 4.2.1.1. Fixed References

DirX Identity uses the following fixed-value references:

- **<?Date/>** - is replaced by the current date.
- **<?Time/>** - is replaced by the current time.
- **<?Gmtime/>** - is replaced by a timestamp representing the current date and time with respect to Greenwich Mean Time zone. The timestamp format is: 4 digits for the year, and 2 digits for month, day of month, hour, minutes and seconds. For example, October 12th, 2008 09:34:25, GMT is represented by the string 20081012093425Z.
- **<?Localtime/>** - is replaced by a timestamp representing the current date and time with respect to the time zone of the machine hosting the C++-based server. The format

is the same as for the reference **<?Gmtime/>**.

- **<?InstallPath/>** - is replaced by the installation path stored in the relevant C++-based server object.
- **<?WorkPath/>** - is substituted by the work path stored in the relevant C++-based server object.
- **<?DeltaInputData/>** - is replaced by the delta information that DirX Identity delivers to agents. This tag allows a customer-specific agent to use this information. See the "Delta Handling" sections for further details.

#### 4.2.1.2. Variable References

Another type of reference starts at a fixed object (set at runtime for the actual running workflow) and can follow any distinguished name an object contains to another object. This method produced relative references that are very reliable against structural changes in the configuration database.

References of this type have the following syntax:

<?*StartObject*@*DistName*@*DistName*@...@*ValueSpec*/>

The available *StartObjects* are:

- **Activity** - the actual running activity when a workflow is running.
- **Configuration -** a pointer to the configuration object in the configuration database.
- **Job -** the actual running job when a workflow is running.
- **loopVar -** the variable to handle multi value attributes in loops.
- **MappingFunctions** - a pointer to the mapping function folder in the configuration object.
- **Workflow -** the actual running workflow.
- **Root -** a pointer to the root object of the Connectivity configuration.

Some of these objects are not static; they are determined by the C++-based server at runtime (for example: the current activity that is running at the moment determines the activity start object). The loopVar construct allows for the handling of multi-value attributes (see the section "Loops" for further details).

*DistName* is a distinguished name in the directory with three possible representations:

- *SingleValueAttribute* - if it is a single value attribute. For example: ?Job@Agent-DN@...
- *MultiValueAttribute*[*DisplayName*] - if it is a multi value attribute and a specific value must be selected by using the display name. For example: <?Job@OutputChannel-DN[MyFirst]@...>
- *MultiValueAttribute*[*OtherAttribute*=*Value*] - if it is a multi value attribute and a specific value must be selected by using any other attribute besides the display name. For example:

<?Job@OutputChannel-DN[Anchor=Mapping]@...>

DirX Identity is tolerant if you define a selection with [OtherAttribute=Value]. If only one element is present in the target list that does not have this attribute specified, this element is taken even if it does not fulfill the condition. No error is reported in this case. If the single element contains another attribute value, an error is reported. Example:

<?Job@OutputChannel-DN[Anchor=Mapping]@...

If two channels exist and one of it contains dxmAnchor=Mapping this one is taken.

If two channels exist and none of if contains dxmAnchor=Mapping, an error is reported.

If one channel exists that does not contain the dxmAnchor attribute, this one is taken.

If one channel exists that contains dxmAnchor=MetaCP, an error is reported.

*ValueSpec* can be one of:

***SingleValueAttribute***

a property name. For example:
?Activity@Description/

***MultiValueAttribute(Name)***

allows for the extraction of values from a list of name / value pairs separated by a space character. For example:
<?Job@SpecificAttributes(Trace)/> which extracts the value from a name/value pair (for example, if the value is 'Trace On', it retrieves 'On' as the value).

***SubstringAttributeValue***

here we have several possibilities:

***Attribute[start:end]***

allows you to extract a part of the string. For example:
<?Activity@Description[0:9]/> with Description='This is my description' evaluates to 'This is my'. Note that the first character is accessed by the value 0. *Start* or *end* may be empty ([:5] or [5:]).

***Attribute*.$n or *Attribute*.name**

to extract specific parts of the string (parts are separated by blanks).
For example: <?Job@SpecificAttrib.../>

'... utes.name(Trace)' with 'Trace Enabled' evaluates to 'Trace'

'... utes(Trace).$0' with 'Trace Enabled' evaluates to 'Trace Enabled'

'... utes(Trace).$1' with 'Trace Enabled' evaluates to 'Trace'

'... utes(Trace).$2' with 'Trace Enabled' evaluates to 'Enabled'

'... utes(Trace).$2[1:2] ' with 'Trace Enabled' evaluates to 'En'

If the n-th field does not exist, an empty string is returned.

Additional specifiers for *ValueSpec* can be used to handle special situations:

**$default**

allows you to define a default value when the attribute or one of the defined *DistName* references could not be retrieved.

...*SingleValueAttribute*($default="*value*")

...*MultiValueAttribute*($default="*value*")

Examples:

...@FileName($default="data.ldif")

...@SpecificAttributes(Trace,$default="0")

...@OutputChannel-DN@FileName($default="data.ldif") → will succeed even if no output channel is present.

If DirX Identity encounters a broken DN reference or has problems reading a defined DN reference, the reference resolution will report an error.

## 4.2.2. Multi-Value References

MultiValueProperties can be handled with:

- A default mechanism
- Loops

### 4.2.2.1. Default Mechanism

If an attribute at the end of the reference chain contains multiple values, all values are concatenated using the comma as separator.

If an [index] range is defined for creating a substring of the attribute value, each value of the substring is built prior to concatenation of the values.

Example:

SpecificAttributes='a b;c d;e f'
<?Job@SpecificAttributes/> evaluates to 'a b,c d,e f'
<?Job@SpecificAttributes[1:1]/> evaluates to 'a,c,e'

### 4.2.2.2. Loops

Loop constructs provide a special handling mechanism. These definitions allow the system

to extract a variable **loopVar** which can then be used in a **cmdline** construct to define exactly how it looks.

<?loop loopVar=<?*object*...@*multivaluedAttribute*/>
cmdline="... <?loopVar@*object* ...@*attribute*($*specifier*,$*specifier*,...)/> ..."
/loop>

Note that no newline is written at the end of the cmdline. If a newline is required, it must be written as **\n**. In a cmdline expression, the following quoting applies:

\n → newline
\t → tab
\r → carriage return
\\ → \

In all other cases, \c yields c, the character itself. Note that a cmdline expression ending in \\\n will result in a backslash at the end of the line.

Possible definitions in the cmdline are:

<?loopVar/>
<?loopVar.*name*/>
<?loopVar.$n/>
<?loopVar[*start*:_end_]/>
<?loopVar.$n[*start*:_end_]/>

Additional *specifiers* for *ValueSpec* can be used to handle specific situations:

- **$default** - allows you to define a default value when the attribute could not be retrieved.
  ...*SingleValueAttribute*($default="*value*")
  ...*MultiValueAttribute*($default="*value*")
  Note: Using this feature extensively might lead to strange behavior because errors in the configuration (for example missing values) are no longer detected.
- $sorted - allows you to sort the result that is returned from the loopVar expression. This is especially useful to sort the SelectedAttributes for a CSV workflow. The SelectedAttributes list contains a leading number, which is used for sorting, and which is automatically removed before used in the cmdline expression.
  This option can be used in combination with other options (for example $optional). Then the options must be separated by a comma.
- $optional - allows you to define whether the result that is returned from the loopVar expression must contain 0 to n or 1 to n values. In the first case, no error is reported when the multi-valued attribute does not contain a value.
  This option can be used in combination with other options (for example $sorted). The options must then be separated by a comma.

*Example 1. Loops*

```
<?loop loopVar=?Job@dxmTCLFurtherScript_DN/>
cmdline="source \"<?loopVar@dxmFileName($optional)/>\""
/loop>
```

This loop creates source statements out of the TCLFurtherScripts objects at the job. Using $optional bewares of error messages if the job does not contain a TCLFurtherScript.

*Example 2. Loops*

If the attribute dxmSelectedAttributes contains the values:

SN surname
GN givenName
Tel phone

then the specification:

<?Job@InputChannel_DN@SelectedAttributes.name/>

returns the string

SN,GN,Tel

and

<?Job@InputChannel_DN@SelectedAttributes.$2/>

results in

surname,givenName,phone

In both cases error messages are generated when the SelectedAttributes are empty. Use $optional to avoid this situation.

*Example 3. Loops*

A loop over all the values of the previous attribute:

<?loop loopVar=<?Job@InputChannel_DN@SelectedAttributes/>
cmdline ="<?loopVar.name/>=<?loopVar.$2/>\n"
/loop>

can be used to create the result:

SN=surname
GN=givenName
Tel=phone

Also in this case an error is produced when the SelectedAttributes list is empty. You can avoid this situation in two ways:

<?loop loopVar=<?Job@InputChannel_DN@SelectedAttributes($optional)/>
cmdline ="<?loopVar.name/>=<?loopVar.$2/>\n"

/loop>

With $opional set no cmdline is generated.

<?loop loopVar=<?Job@InputChannel_DN@SelectedAttributes($default="sn surname")/>
cmdline ="<?loopVar.name/>=<?loopVar.$2/>\n"
/loop>

With $default set a fixed commandline is generated:

sn=surname

### 4.2.3. References in References

DirX Identity supports two mechanisms:

- Variables
- Hierarchical References

Note: Nested references are only supported to a depth of 20. Otherwise an error is reported. This prevents infinite loops.

#### 4.2.3.1. Variables

To enhance readability of scripts with references, variables can be defined. First, the variable must be defined at the beginning of the script:

<?set *var=any_string*/>

where *var* defines the name of the variable and *reference* can be any string or reference definition. Then the variable can be used:

<?$*var*/>

The set statement produces a resolved line as output. Therefore you should place it into a comment line of your configuration file. This is especially useful for debugging. Alternatively you can use the set_non_verbose statement that suppresses the output line completely.

<?set_nv *var=any_string*/>

The scope of the variable definition is limited to the current object being processed with the reference mechanism. Variables declared in INI file templates are neither known to Tcl scripts, nor to attribute configuration files.

Variables declared in Tcl scripts are accumulated. Since Tcl scripts are processed in the order "Tcl control script", "Tcl mapping script", "Tcl further script", and "Tcl profile script", variables defined in the control script are also known in the mapping script or in the profile script.

In case of recursively-defined tags, the variables are propagated top-down and bottom-up.

*Example 4. Variables*

Definition: <?set_nv src=rh_<?Job@InputChannel-DN@RoleName/>/>
Usage: ...<?$src/>

As shown in the example, variable definitions may contain tags.

*Example 5. Variables*

# <?set timestamp=<?date/>-<?time/>/>
# <?set workflow=<?Workflow@DisplayName/>/>
# <?set started=<?$workflow/> started at <?$timestamp/>/>
# <?$started/>

In this case, the tags are evaluated recursively, e.g.

# <?set timestamp=2002-05-23-15:14:39/>
# <?set workflow=ADS2Meta/>
# <?set started=ADS2Meta started at 2002-05-23-15:14:39/>
# ADS2Meta started at 2002-05-23-15:14:39

*Example 6. Variables*

# <?set timestamp=\<?date\/>-\<?time\/>/>
# <?set workflow=\<?Workflow@DisplayName\/>/>
# <?set started=\<?$workflow\/> started at \<?$timestamp\/>/>
# <?$started/>

This example shows that you can also define tags in the variable values that are evaluated when the variable is referenced, rather than during variable definition. For this purpose, the tags <? and /> have to be escaped using \ as escape character.

This sequence is evaluated to:

# <?set timestamp=<?date/>-<?time/>/>
# <?set workflow=<?Workflow@DisplayName/>/>
# <?set started=<?$workflow/> started at <?$timestamp/>/>
# ADS2Meta started at 2002-05-23-15:14:39

#### 4.2.3.2. Hierarchical References

Hierarchical references can be used to set up a flexible parameter concept.

Let's assume a Tcl script needs the parameter base_obj. The reference points to a specific attribute in the output channel of a job.

(1) set base_obj "<?job@OutputChannel-DN@SpecificAttributes(base_obj)/>"

The referenced attribute can contain again a reference to a central parameter located for

example in the central configuration object:

(2) base_obj <?Configuration@SpecificAttributes(base_obj,$default="o=pqr")/>

In this case the Configuration object should contain for example an entry "base_obj ou=people,o=pqr". Then this value is delivered to the previous construct (2) and in a second step provided to the Tcl line via (1).

If no entry is available in the Configuration object then the default value "o=pqr" is used from (2).

If the central value shall no longer be used then a fixed value can be entered into (2), for example base_obj c=DE.

### 4.2.4. Abbreviating Reference Descriptions

Since almost all the attributes in the Connectivity configuration subtree dxmc=DirXMetahub start with the prefix "dxm", this prefix can be dropped.

Example:

<?Workflow@dxmMetahubSyncServer_DN@dxmServer_DN@dxmAddress/>

can be written more elegantly and shorter:

<?Workflow@MetahubSyncServer_DN@Server_DN@Address/>

If an attribute *name* is requested from the DIT that does not have the prefix dxm, and this attribute does not exist in the search result, the value of attribute dxm*name* is taken. If the attribute dxm*name* is not found in the search result, it is reported to be missing (if no $default or $optional specifier is used).

### 4.2.5. Using Relative References

If strings are to be concatenated, an attribute can be referenced relative to the object that has been defined to get the previous attribute value. You can then define a relative reference (it starts with an @):

Example:

<?Workflow@dxmMetahubSyncServer_DN @dxmServer_DN@dxmAddress/>:
<?@dxmDataPort/>

### 4.2.6. File Name Handling

For the attribute "...@dxmFileName" or "...@FileName" the absolute path name of the referenced file is created automatically by the C++-based server. The working path of the activity and the installation path is included. Also the channel mapping is taken into account to calculate the correct path.

File names can also contain reference tags.

### 4.2.7. Complex Example

The following example demonstrates the power of the tag syntax. The use of variables and nested tags is demonstrated. To simplify the documentation, line numbers are added which must not be present in a real code example. Because some lines are very long, they are wrapped, which is indicated by a "/" character which is also not part of the real code. The example is used in the DirX Identity default applications to build the encrypted attributes section in configuration files.

```
[EncryptedAttributes]
1   <?loop
2
3 loopVar=<?Workflow@Activity-DN[IsStartActivity=TRUE]@RunObject-DN@
/
  OutputChannel-DN[Anchor=DATA]@SelectedAttributes($sorted)/>
4
5   cmdline=
6     "<?set_nv innerLoopCmd=\<?loop
7
8   Loopvar=\<?Job@InputChannel-DN@ConnDir-DN@AttrConfigFile-DN@ /

AttrConfigItem(Abbr:<?loopvar.name/>,$optional)&($7=Encryption:Y).$3[
8:$-2]\/>
9
10         Cmdline=\"\<?Loopvar.name\/>=1\"
11
12     \/loop>/><?$innerLoopCmd/>"
13
14  /loop>
```

The whole expression serves to create the encrypted attributes section of an ini file. In step one, all selected attributes of the output channel have to be determined. In the second step, the AttrConfigItems of the connected directory are evaluated.

Then, for each selected attribute obtained in step one, it is checked whether the AttrConfigItem of this attribute is encrypted. In this case, the name is extracted from a substring of the third field of the item. This name then is printed and followed by "=1", resulting for example in a statement like "password=1".

To get all selected attributes of the output channel, a loop is opened in line 1. The loopVar has the assigned attribute "selectedAttributes" of the output channel object, whose (multi-valued) entries are sorted (defined by option $sorted).

The command line is executed once for each value contained in "SelectedAttributes". It contains two actions: Setting of variable "innerLoopCmd" (line 6). This setting ends with the

closing tag in line 12.

The second action is evaluating this variable. Since the result of the evaluation is a loop statement, this (inner) loop then is executed. Note, that we have constructed a nested loop here that makes use of variables, what is normally not supported by the tag syntax.

The inner loop statement starts with \<?loop in line 6 and ends with \/loop> in line 12. The tags have to be escaped by \, otherwise it would interfere with the outer loop.

The loopVar of the inner loop has assigned the AttrConfigItem (multi-valued) attribute of the outputchannel object. From this multi-value attribute, the line assigned to the selected attribute contained in the loopVar defined in line 3 is extracted.

This is done by the statement ...@AttrConfigItem(Abbr:<?loopvar.name/>,$optional) ... in line 8. The <?loopvar.name/>-Tag in line 8 is nested into the <$set_nv tag starting in line 6. Since nested tags are evaluated from inside to outside, first the value contained in the outer loopVar (e.g. "password") is replaced in the expression.

Then the whole expression is assigned to the variable "innerLoopCmd". The AND-relation to the Encryption flag is done with "&($7=Encryption:Y)" in line 8. Thus, for SelectedAttributes contained in AttrConfigItem flagged with "Encryption:Y" the line is chosen and a substring of the third field $3 is extracted.

Defining the range $3[8:$-2], 8 characters are cut from the left and two characters are cut from the right, extracting exactly the name from the expression.

In the commandline of the inner loop (line 10), the expression <?LoopVar.name/>=1 is written. Loopvar is either empty (in case the expressions in line 8 do not match) or does contain exactly one value, the name of the attribute to be encrypted.

In case of an empty loopVar, no output is written. In case the loopVar contains the name, the expression <name>=1 is written.

Thus, the whole complex expression evaluates to an empty string or for example to something like "password=1".

# 5. Customizing Parameters

The topics in this section describe how to:

- Create a new proposal list
- Add new permission parameters
- Add new role parameters
- Create dependent proposal lists

## 5.1. Creating a New Proposal List

To create a new proposal list:

- In the **Domain Configuration** view, use DirX Identity Manager to create a new proposal list entry in the *Domain* → **Customer Extensions** → **Proposal Lists** folder.
- You can fill the proposal list in two ways: Define the values as a simple list or search for the list entries in the directory.
- Select **String** as **Type** and enter the possible values in the **Item List** tab of the new proposal list.
- Select **DN** as **Type** if you intend to search for the entries, and then define the search parameters.You can add more list items if you enter values into the list.
- You can choose between two formats:
  *storedValue*
  *storedValue*;*Description*

  The *storedValue* part is the value that is stored at an object when you select it from the list.If you add the *Description* part, then this description is displayed, but the stored value is stored in the directory.This makes your proposal lists more readable and allows adding comments for users.

  If you selected **DN** as **Type**, you must specify the **Display** and **Storage** attributes of which the list shall be composed.

  Example:
  **DE** - in this case 'DE' is displayed at the user interface and stored in the directory.
  **DE;Germany** - in this case 'Germany' is displayed at the user interface but 'DE' is stored in the directory.
- If the list is mastered in another database, consider defining a DirX Identity batch workflow that feeds the list regularly into the **dxrProposedValues** attribute of the proposal list entry.
- Enter the reference to this list in the object descriptions of all objects where you need it.Use the XML element "tagprovider" with XML attribute "dn=...".

If the proposal list is only valid for one target system, you can choose to place it in the target system subtree's Proposal Lists folder.

## 5.2. Adding a New Permission Parameter

To make a user attribute a permission parameter:

- In the **Domain Configuration** view, use DirX Identity Manager to add the user attribute to the list of user attributes that can be used as permission parameters.If the attribute is not in the list, extend your user.xml object description accordingly (**Provisioning** → **Domain Configuration** → **Customer Extensions** → **user.xml**).
- Define the matching rules that use the permission parameter in the permission objects.
- Check that the new permission parameter attribute is visible in the DirX Identity user object.
- Ensure that the permission parameter values are set for the users (fill in default or specific values).
- Extend the object description of target system groups to contain the new permission parameter: specify the new permission parameter as a new specific attribute "dxrRPValues(<property name>)", where the property name is exactly the same as for the user object.
  Note: Place these definitions into the **group.xml** files of the individual target systems of your domain ("**Target Systems** → *targetSystem* → **Configuration** → **Object Descriptions**").Otherwise the next upgrade or update overwrites this definition.
  Note: You can define a single value attribute at the user object and a multi-value attribute at the group object.If the group object definition is missing, the user object definition is inherited automatically.
- Extend the property page description of the target system groups to display the new permission parameter.
- Supply permission parameter values in the target system groups.

The definition and use of permission parameter attributes in the **user.xml** and **group.xml** files must be consistent in your application.If not, incorrect display in the DirX Identity Manager, non-functional login into the DirX Identity Web Center or incomplete privilege resolution could be the result.

## 5.3. Adding a New Role Parameter

Role parameters allow you to define generic roles that become specific on assignment to a user.For example, the My-Company sample domain defines a generic role Project Member (in DirX Identity Manager's **Provisioning** view, see **Privileges** → **Roles** → **Corporate Roles** → **Project Specific** → **Project Member**).This role is linked to the **Project** role parameter (see the **Role Parameters** tab in the **Project** role, and then see **Domain Configuration** → **Customer Extensions** → **My-Company** → **Role Parameters** → **Project**) which controls the specific project that a particular user is to become a member of when the **Project Member** role is assigned to him.The user assigning the role selects or supplies the right value for the role parameter; in this case, the project name (**MoreCustomers**, **HighPerformance**, and **OptimizeIT**).

There are five ways to define a role parameter (which are controlled by the **Type:** field in the

role parameter definition):

- As strings,either using specified values, using a proposal list, or using both.You can specify string values such as "Project A" or "Project B" - these values become selections in a drop-down list at the Web Center interface on user-role assignment.Alternatively, you can use a proposal list for the selections that will be displayed.For example, the **Project** role specifies a **Projects** proposal list (see **Domain Configuration** → **Customer Extensions** → **My-Company** → **Proposal Lists** → **Projects**).The items in this list (see the **Item List** tab) will be displayed for selection on user-role assignment.You can also combine both proposal list and specific values within the same role parameter.
- As text that is to be entered at the Web Center interface; for example, to resolve a parking space assignment in a company lot - the user enters his auto identification number (license plate number) and this value is a role parameter that is added to the role assignment.
- As integer parameters (either within a range or with specific values, like 0 or 1)
- As a distinguished name (DN).If this definition is selected, a simple ordered list is presented for selection.
- As a hierarchical list of DNs; for example, the **Cost Locations** role parameter (see **Domain Configuration** → **Customer Extensions** → **My-Company** → **Role Parameters** → **Cost Locations**) specifies a tree of cost locations that is presented for selection when the **Cost Location Manager** role (see **Privileges** → **Corporate Roles**→ **General** → **Cost Location Manager**) is assigned to a user.

To set up a role parameter:

- In the Domain Configuration view, use DirX Identity Manager to create or maintain role parameter definitions (**Provisioning** → **Domain Configuration** → **Customer Extensions** → **Role Parameters**).
- If you use proposal lists for role parameter definitions, set up these objects (**Provisioning** → **Domain Configuration** → **Customer Extensions** → **Proposal Lists**).
- If you intend to use an object structure or object hierarchy for role parameter objects, set up the corresponding definitions (for example, you can use an existing Business Object structure).
- Define the matching rules that use the role parameter in the role objects.
- Supply the necessary role parameter values in the target system groups (add the possible parameters to the groups or create new groups for these parameters).

# 5.4. Adding Dependent Proposal Lists

This feature allows you to interdependent build drop-down lists in Web Center that are displayed on a single page.The feature is available in the context of creating a new object or modifying an existing object.When a Web Center user selects an item in the first list, the second list becomes populated with choices that are only relevant to the context of the item selected from the first list.When an item in the second list is selected, the corresponding choices in the third list become available, and so on.

The feature is not available within DirX Identity Manager or in Web pages generated automatically from request workflows.

The feature makes use of proposal lists of type **Dependent DN**.Within this type, attributes selected in the related master proposal lists are dynamically replaced in the search filter or the search base defined for that list.

A dependent proposal list is configured by search base, filter, and scope of the search.Furthermore, the attributes containing the value for display and the value to be stored with the managed object are to be configured (**Display Attribute** and **Storage Attribute**).An extra feature is the possibility to define a **Link Attribute** with the proposal list.If you specify such a Link Attribute, this attribute is expected to hold DN links to other objects.The result is that the proposal list data are read from the linked objects rather than from the objects being returned by the search.

### 5.4.1. Configuring Dependent Proposal Lists

Perform these configuration steps to add a dependent proposal list:

- In Identity Manager, define a proposal list of type **Dependent DN**.
- Filter and / or search base of the proposal list have to be expressed in terms of the storage attribute $(<master_attribute>) whose value is used as master for the related list.
- In the object description, define a tag provider for the dependent property. Set the provider's class name to **siemens.dxr.service.tags.DependentProposal**, and its dn to the distinguished name of the proposal list created in the preceding step.
- Reference the master attribute from the description of the dependent attribute in Web Center's forms-config.xml. Assure that the **renderersProperties** attribute of the actual form-property contains the expression **"master:***form___attribute***"** where *form* is the name of the form holding the master attribute and *attribute* contains the master attribute's name.
  Example: rendererProperties="master:modificationForm_c".

### 5.4.2. Example Configuration

In this section, we describe how to configure a three-level dependency between proposal lists; that is, drop-down lists in Web Center. The samples use the My-Company sample scenario.

The hierarchy of proposal list is:

**Countries → Locations → Rooms**

A user first selects a country. Now the list of locations displays only the locations of this country. If the user selects a location, the list of rooms displays only the available rooms of this location.

#### 5.4.2.1. Defining a Dependent Proposal in DirX Identity Manager

We define three new proposal lists and assign them to properties of the user object.

Sample List for Countries

The top-level of our hierarchy is the proposal list for countries defined under **Business Objects → Countries**. Perform these steps:

- Select the view **Domain Configuration**.
- Select **Customer Extensions → Proposal Lists** and then **New Proposal** from the context menu.
- Create the new proposal list with this data:
  Name: Business Countries
  Description: List of countries derived from business objects
  Type: DN
  Searchbase: cn=Countries,cn=BusinessObjects,cn=My-Company
  Searchfilter: objectclass=”dxrCountry”
  Searchscope: one level
  Display attribute: description (for example “Germany”)
  Storage attribute: c (for example “DE”)
  Link attribute: <empty>

To assign the proposal list to the corresponding object property in the object description, perform these steps:

- Select the view **Domain Configuration**.
- Select **Customer Extensions → Object Descriptions → UserCommon.xml**
- Click **Edit** and add the following code:

```
<property name="c"
label="Country"
editor="siemens.dxr.manager.controls.MetaRoleJnbComboBox"
type="java.lang.String"
incremental="false">
<tagprovider class="siemens.dxr.service.tags.Proposal"
  dn="cn=Business Countries,cn=Proposal Lists,cn=Customer
Extensions,cn=Configuration,$(rootDN)"
  proposals="dxrProposedValues"/>
</property>
```

Sample List for Locality

The second level of our hierarchy is a proposal list of company locations assigned to a country. So the list of available locations depends on the selected country.

- Select the view **Domain Configuration**.
- Select **Customer Extensions → Proposal Lists** and then **New Proposal** from the context menu.
- Create the new proposal list with this data:
  Name: Dependent Localities
  Description: List of localities derived from business objects and dependent on the 'Business Countries" list
  Type: Dependent DN
  Searchbase: c=$(c),cn=Countries,cn=BusinessObjects,cn=My-Company
  Searchfilter: objectclass=”dxrLocation”
  Searchscope: subtree
  Display attribute: l (for example “My-Company Berlin”)
  Storage attribute: dn (for example “l=My-Company London, c=GB,cn=Countries,cn=BusinessObjects,cn=My-Company”)

The searchbase contains a placeholder c=$(c) for the selected country. This placeholder will be replaced with the actual value during runtime. The storage attribute is the DN of the selected dxrLocation-object (for example “l=My-Company London, c=GB,cn=Countries,cn=BusinessObjects,cn=My-Company”).

Note that the “c” in $(c) is the name of the user attribute holding the country, not the naming attribute of the country object. If the country is stored in a user attribute with the name userCountry, the search base starts with c=$(userCountry).

To assign the proposal list to the according object-property process these steps:

- Select the view **Domain Configuration**.
- Select **Customer Extensions → Object Descriptions → User.xml**
- Click **Edit** and add the following code (this overloads the existing definition in **Object Descriptions → User.xml**):

```
<property name="dxrLocationLink"
type="java.lang.String"
label="Location Link"
multivalue="false"
incremental="false"
editor="siemens.dxr.manager.controls.MetaRoleJnbComboBox"
editorparams="editable=true">
<tagprovider class="siemens.dxr.service.tags.DependentProposal"
 dn="cn=Dependent Localities,cn=Proposal Lists,cn=Customer
Extensions,cn=Configuration,$(rootDN)"
 proposals="dxrProposedValues"/>
</property>
```

Sample List for Rooms

The third level of our hierarchy is a proposal list of room numbers assigned to a location. In our sample data the room objects are created as context objects to a location.

- Select the view **Domain Configuration**.
- Select **Customer Extensions → Proposal Lists** and then **New Proposal** from the context menu.
- Create the new proposal list with this data:
  Name: Dependent Rooms
  Description: List of rooms derived from business objects and dependent on the 'Dependent Localities" list
  Type: Dependent DN
  Searchbase: $(dxrLocationLink)
  Searchfilter: objectclass=”dxrContext”
  Searchscope: base object
  Display attribute: cn
  Storage attribute: cn
  Link attribute: dxrContextLink

The main difference in this example is the usage of the link attribute. In this case the actual values for the proposal list are read from the dxrContextLink. In our sample you need to set from each location the links to the relevant room objects.

To assign the proposal list to the according object-property process these steps:

- Select the view **Domain Configuration**.
- Select **Customer Extensions → Object Descriptions → UserCommon.xml**
- Click **Edit** and add the following code:

```
<property name="roomNumber"
label="Room"
editor="siemens.dxr.manager.controls.MetaRoleJnbComboBox"
type="java.lang.String"
incremental="false">
<tagprovider class="siemens.dxr.service.tags.DependentProposal"
 dn="cn=Dependent Rooms,cn=Proposal Lists,cn=Customer
Extensions,cn=Configuration,$(rootDN)"
 proposals="dxrProposedValues"/>
</property>
```

#### 5.4.2.2. Changes in the Web Center Configuration Files

To test the dependent proposal list we modify and add some code in the file:

*installation_path*\web\webCenter\WEB-INF\config\identity\forms-config.xml

In our sample data we replace the existing definition for “c” (country) with this code:

```
<form-property name="c"
type="java.lang.String"
fieldRenderer="combobox"
label="ldap.attribute.c"
width="100%" y="3" x="0"/>
```

We replace the existing definition of the property dxrLocationLink with this code:

```
<form-property name="dxrLocationLink"
type="java.lang.String"
fieldRenderer="combobox"
rendererProperties="master:modificationForm_c"
label="ldap.attribute.l"
width="100%" y="3" x="1"/>
```

A new definition is added for room number:

```
<form-property name="roomNumber"
type="java.lang.String"
fieldRenderer="combobox"
rendererProperties="master:modificationForm_dxrLocationLink"
label="Room"
width="100%" y="6" x="1"/>
```

The most important change in the code is the reference to the master field (master:...). If a master is defined then the dependent combo listens to changes in the master combo and refreshes itself according to the proposal list definition.

After performing the above described modifications restart the Tomcat Service.

In the user modification page you can see the three combo box fields for country, locality and room. After a country is selected the combo for locality is filled according to the new value of the master combo and set to an empty value. After a new locality is selected the combo for room number is filled.

# 6. Customizing the Provisioning Tree View

This chapter describes how to customize DirX Identity's Provisioning tree view. It describes how to:

- Structure the DirX Identity Provisioning tree into a hierarchy
- Change the display name of entries in the tree view

## 6.1. Changing the Provisioning Object Tree

To make it easier to manage a large number of users, roles or permissions, you can build a hierarchical structure into the DirX Identity Provisioning tree by creating new folders.

If you use DirX Identity's user integrator workflow to import users from a corporate directory into DirX Identity, the tree structure will be copied.As a result, the DirX Identity store will follow the corporate directory's hierarchical structure.

The DirX Identity default schema in the user subtree allows you to create objects with the following LDAP classes and according to the following structure:

```
Organization
     OrganizationalUnit (recursively)
     locality
          organizationalUnit (recursively)
          organizationalPerson (or inetOrgPerson)
     organizationalPerson (or inetOrgPerson)
organizationalUnit
     organizationalUnit (recursively)
     organizationalPerson (or inetOrgPerson)
domain
     domain (recursively)
     organization
     organizationalUnit (recursively)
     organizationalPerson (or inetOrgPerson)
country
     organization
     organizationalPerson (or inetOrgPerson)
```

You can change this structure according your needs, for example, to organize a large number of flat entries into subfolders, or to build a structure that reflects the administrative responsibilities of DirX Identity user administrators.You can then specify access control for these subfolders using the meta directory administration tools and thus build administrative areas.

You can use the same subfolder mechanism to structure the roles and permissions subtrees.But keep in mind that roles and permissions are located in separate subtrees below the domain root: "cn=RoleCatalogue" and "cn=Permissions".

You cannot use the folder mechanism to structure the target systems subtree; the target system folders themselves structure the view.Target system accounts are flat below the Accounts folder.The DirX Identity tree does not reflect a hierarchical structure for target systems.

Structure rules similar to those used in LDAP or X.500 directories restrict the options for building hierarchical trees.DirX Identity uses the XML "parents" attribute to specify these restrictions in the object descriptions.

## 6.2. Changing the Display Name of Entries in the Tree View

You can change the display name of an entry in the DirX Identity Manager tree view.To display another attribute value, take the appropriate object description (e.g., **user.xml** from the **Customer Extensions** folder) and change the XML attribute **displayattribute** in the XML **definition** element.If this attribute is not specified, create it according the following example fragment, which defines the **cn** attribute for display:

```
<object name="dxrUser">
<definition ...
   displayattribute="cn"
            ...
/>
...
</object>
```

To display a combination of attribute values, use a Java script and define it in the object description as previously described. Enter the script definition near the end of the object description independent of any property description. The following excerpt shows an example that causes Manager to display the surname and the given name of a user entry:

```
<script name="displayname"  return="title" >
<![CDATA[
obj=scriptContext.getObject();
gn=obj.getProperty("givenName");
sn=obj.getProperty("sn");
title=sn;

if (gn!=null && gn.length()>0)
title=title+", "+gn;
```

```
title=title+"";
]]>
</script>
```

# 7. Customizing Wizards

This section describes how to customize and design new Provisioning and Connectivity wizards.

## 7.1. Customizing the Provisioning Target System Wizard

The target system wizard configuration depends on two XML descriptions:

- The basic Target System wizard XML description
- The target system type-specific XML description

The next sections provide customization information about these XML descriptions.

### 7.1.1. Basic Target System Wizard Description

The basic target system wizard definition is located in the following XML file:

**Provisioning View → Configuration → Wizards → newTS.xml**

This file contains a set of variables that define, for example, the name of the Identity Store, the name of the relevant workflow templates (for example, the Policy Execution workflow) and the names of the relevant scenarios for Tcl-based, password and real-time workflows. It also contains the definitions of all workflow steps.

This file is only visible in the Data View and *cannot be customized*.

### 7.1.2. Target System Type-Specific Wizard Description

All target system-specific wizard definitions are located in the following XML file:

**Connectivity View → Configuration → Connected Directory Types →** *type* **→ Wizards → TSWizardConfig.xml**

You are allowed to customize the following definitions in this file:

- Connected directory type specific attribute definitions, for example:

  <cdtypespecific header="Connected Directory Type Specific">
  <specificattribute label="User base" name="user_base" dxroption="AccountRootInTS" />
  <specificattribute label="Group base" name="group_base" d />
  </cdtypespecific>

  which means that dxmSpecificAttributes(group_base) is displayed as a field with the name 'GroupRelative User Base'. The value is stored as specific attribute at the connectivity connected directory. User_base is also stored there. Additionally the same value is stored in the dxrOptions attribute of the Provisioning Target System.

You can customize the following attributes:

- label: displayname
- name: the attribute name in the Connectivity connected directory-specific attribute
- dxroption: the attribute name in the Provisioning target system dxrOptions attribute
- editor: the editor to be used

These attributes appear in the CD Configuration step in the Connected Directory Type Specific section.

- Attribute visibility definitions, for example:

  <service header="Service" ipadress="true"/>

  which means that the ipaddress attribute is requested by the target system wizard. It is shown in the CD Configuration step in the Service section.

  The following table shows the attributes you can customize, where:

  - Name specifies the name of the XML attribute
  - LDAP Name specifies the name of the corresponding LDAP attribute
  - Label specifies the text that appears in the wizard
  - Default specifies whether the attribute is displayed if nothing is defined (true) or not (false).

  | Name | LDAP Name | Label | Default |
  |---|---|---|---|
  | ipadress | dxmadress | IP Address: | false |
  | dataport | dxmdataport | Data Port: | true |
  | servername | DxmServerName | Server Name: | false |

- Additional wizard steps, for example:

  <cdadvancedstep reference="Configuration" />

  which triggers the Target System wizard to display this additional step.Configuration is the reference to a property page defined for this connected directory type.

- Bind profile definitions, for example:

  <bindprofile header="Bind Profile" anchor="Domain_Admin" />

  which selects the bind profile where the anchor attribute is set to "Domain_Admin".As an alternative to using the anchor attribute, you can specify **name=”my-name”** to select the bind profile via the dxmdisplayname attribute.

# 7.2. Customizing a Connectivity Wizard

Wizards provide one way to hide parts of the complexity of DirX Identity and allow you to concentrate on the essential issues when configuring or reconfiguring workflows.XML descriptions allow you to define Connectivity wizards that help to create and maintain connected directories and workflows at the DirX Identity Manager Global View.

To design a Connectivity wizard, you:

1. Write the XML description of the wizard.
2. Give this file the correct name.
3. Link this wizard from the workflow or connected directory object.

Now you can configure the workflow or connected directory object with the wizard. The next sections describe these steps in more detail.

## 7.2.1. Defining Connectivity Wizards

Connectivity wizards always start with a template that the user must select from the configuration database. The wizard derives the new object from this template and cannot create a new object without having an existing template to use as a base

The DirX Identity configuration database does not contain any special templates for this task. All objects that fit can be used as templates. For example:

- The connected directory configuration wizard presents all existing connected directory objects (the default objects and any connected directories you have copied or created).
- The workflow configuration wizard presents all workflow objects that have the same type of endpoints (regardless of the direction of the workflow).

Connectivity wizards can be configured via XML and are simply a selection of forms (property dialogs) that are necessary to configure a connected directory or a workflow. The wizard designer can heavily influence the level of detail that is presented to the user.

Example: If a workflow can be configured with 20 property dialogs in the Expert View, then a wizard could select 5 of these dialogs to reduce the level of detail. It is even possible to extract specific fields from a property dialog and to hide other fields.

It is also important to understand that workflow wizards may vary depending on the complexity of the workflow. You can use a simple wizard to configure a workflow with one activity (Wizard B in the following figure) or a complex wizard to configure a workflow with three or more activities (Wizard Z in the following figure).

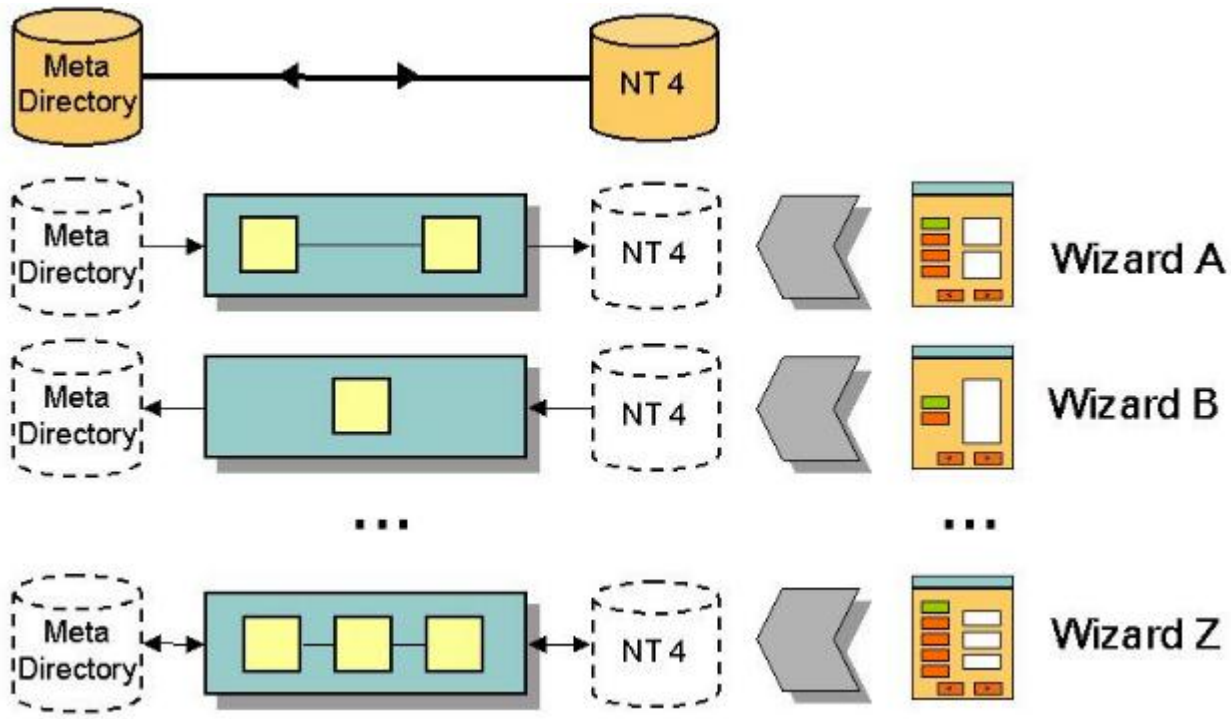



*Figure 5. Wizard Implementations*

The workflow wizard names (which you can view in the field Wizard Type in the workflow object) are constructed in the following way (we recommend using this convention):

*source_type-specifier-target_type*

where *source_type* stands for the type of the source connected directory, *target_type* for the target connected directory, and *specifier* allows you to distinguish between different implementations of the workflow between two connected directories (for example, a one-step workflow or a two-step workflow, which must be configured differently).

Examples: LDAP-oneStep-FILE, LDAP-twoStep-FILE, LDAP-twoStep-ADS

Because DirX Identity cannot automatically detect whether a workflow works in both directions (a Tcl script or a custom agent can both read and write via its channels to connected directories) you can use the Bidirectional attribute in the workflow object to define this case.

Connectivity wizard objects are normally located in the **Configuration** → **Agent Types** → *agenttype* → **Wizards** folder for workflow wizards and **Configuration** → **Connected Directory Types** → *conndirtype* → **Wizards** folder for connected directory wizards. You can add your own wizards there or add new agent and connected directory types together with wizard descriptions.

### 7.2.2. Connectivity Wizard Naming Rules

The naming rules are different for Java-based and Tcl-based workflows. Note that these naming rules are not mandatory but it is good style to use them.

#### 7.2.2.1. Naming Rules for Java-based Workflows

The naming rules for Connectivity wizards of Java-based workflows are as follows:

**wf-***wftype-Specifier***.xml**

where *wftype* defines the workflow type. Possible values are:

**EventBased** - for wizards that handle event-based workflows that work on the Identity Store.

**Password** - for wizards that manage password information within the Identity Store.

**Provisioning** - for wizards for provisioning between the Identity Store and the connected system.

**Source** - for wizards that manage workflows between a source system and the Identity Store.

and where Specifier defines a specific wizard type.

For example:

wf-Provisioning-Synchronize.xml - a standard wizard for synchronization from the target system in the Identity Store to the connected system and backwards.

#### 7.2.2.2. Naming Rules for Tcl-based Workflows

The naming rules for Connectivity wizards of Tcl-based workflows are as follows:

**wf-***SourceType***-***TargetType***.xml**

where *SourceType* defines the type of the source connected directory, *TargetType* defines the type of the target connected directory and *Specifier* allows to distinguish between several types of wizards between these two connected directories.

Note: When you use the Copy functionality of the Global View (Configure) to create a new workflow from a template. The workflow template needs a wizard definition otherwise the copy cannot be performed. You can use the **wf-standard.xml** wizard template for your workflow. Copy it in the Expert View, rename it to your workflow (for example: wf-LDAP-MyDir.xml) and use it in your workflow. This standard wizard only contains a naming step.

Examples:

wf-LDAP-FILE.xml

A wizard that allows for configuring a workflow between an LDAP directory and a FILE directory.

wf-LDAP-2step-EXCHANGE.xml

This wizard allows the configuration of a two-step workflow between an LDAP directory and an EXCHANGE directory.

### 7.2.3. Connectivity Wizard XML Description

A Connectivity wizard's XML description has the following structure:

```
<wizard name=" name" title="title " >
<illustrator name=" illustrator" title="title" />
...
<step illustrator>
<node nodeDefinition/>
<description>
... DESCRIPTIVE TEXT ...
</description>
<propertypage pageDefinition/>
<prestep class="javaClass" />
<poststep class="javaClass" />
</step>
...
</wizard>
```

The wizard description consists of two parts:

- The definition of the illustrator steps (this is the sequence of steps that is displayed on the left side of the wizard)
- The related step descriptions (this is the step content that is displayed on the right side of the wizard)

At the beginning, two properties can be defined:

- **name** - the internal name of this element (not the display name!), Example: name="CDwizard"
- **title** - the display name in the header of the wizard window. Example: title="Create or Maintain a Connected Directory"

The next sections provide details about **illustrator** and **step** descriptions and also about node references.

#### 7.2.3.1. Illustrator Description

This section consists of one or more **illustrator** sections. These sections contain the following properties:

- **name** - the internal name of this element (not the display name!). This name is used to link the step to the illustrator element. Example: name="attrconfig"
- **title** - the display name in the illustrator element. Example: title="Set Attribute Configuration"

The sequence of illustrator definitions defines the sequence of the wizard steps.

**Example:**

Open **Configuration → Connected Directories → LDAP → Wizards → connDir-LDAP.xml** and click the **Content** tab. Select **New Window** from the context menu in the editor window.

```
<illustrator name="General" title="General Information" />
<illustrator name="Schema" title="Schema Information" />
<illustrator name="AttrConfig" title="Attribute Configuration" />
<illustrator name="BindProfiles" title="Bind Profiles" />
<illustrator name="OperAttr" title="Operational Attributes" />
<if var="wizard.templatechooser" value="*" >
<illustrator name="Naming" title="Set Directory Name" />
</if>
```

This example shows a five-step wizard (General Information, Schema Information, Attribute Configuration, Bind Profiles and Operational Attributes). The last step is only used as naming step when a new object has to be created during a copy operation in the global view. In this case the template-chooser step and eventually a direction selection step is automatically added as first steps. These must not be defined in the wizard definition.

#### 7.2.3.2. Step Description

This section consists of one or more **step** sections. These sections contain the following attributes:

- **name** - the internal name of this element (not the display name!).
  Example: name="attrconfig"
- **title** - the display header in the step element.
  Example: title="Attribute Configuration"
- **illustrator** - the link to the illustrator to identify the corresponding illustrator.
  Example: illustrator="attrconfig"
- **description** - a textual description what to do in this step. Example:
  <description> Check and maintain the attribute configuration. Then click Next.
  </description>
- **node** - the path to the element to be displayed (starting with the actual object).
  Example: node="dxmAttrConfigFile-DN" or node="dxmActivity". See below for a detailed description of this point (node references).
- **<propertypage reference ...** - a reference to the property page to be displayed (the reference is based on the node definition).
  Example: reference="general"
- **<propertypage ...** - a definition of the property page to be displayed (with all the features of property pages described in the object descriptions).

#### 7.2.3.3. Node References

The default parameter for a node is **Connected Directory** or **Workflow** (depending on the type of wizard you are defining). Thus, if you need a tab from the connected directory or workflow object, you do not need to specify this parameter.

Alternatively you can specify other objects. The reference mechanism is similar to the reference mechanism used for configuration files (see "Connectivity References" for details). The features are:

- Specify another object by defining the link path beginning from the workflow object.

Example: "dxmActivity-DN"

This works only if the workflow has only one activity. Otherwise, DirX Identity does not know which Activity to take.

- Define a specific object when there are multiple ones with a filter expression.

Example: "dxmActivity-DN[dxmIsStartActivity=true]"

- Define a child object. This definition is especially useful for Java-based workflows.

Example: [cn=error]

In this case the child object (the activity) with the name 'error' in the cn attribute is referenced.

- Specify a path to a child object. This definition is especially useful for Java-based workflows.

Example:
[cn=join]@[cn=TS]@dxmSpecificAttributes[channelParent]@[dxmDisplayName=accounts]

This example selects the activity 'join' and from there the port 'TS'. Next it follows the link to the channel parent and selects there the object with the display name 'accounts', which is a channel object.

- Follow the path to an object.

Example: "dxmActivity-DN@dxmRunObject-DN@dxmTCLFurtherScript-DN[dxmDisplayName=Abc]"

This example shows that you must always build unique references. You can use any attribute that distinguishes the objects at one level. Examples are the "dxmStartActivity" or "dxmEndActivity" attributes of an activity object or the display name of any object. In the last case, a change of the display name would of course break the reference, which means that your wizard will no longer work in this part.

A reference to the common name of an object is secure because this name will never change. On the other hand, this reference will no longer work when the common name is changed (which happens during the copy operations in DirX Identity).

The only secure way that is resistant to copy operations and other changes is to use a special property that does not change. We introduced **dxmAnchor** attributes at some objects to provide such capabilities.

**Example:**

Open **Configuration** → **Connected Directories** → **LDAP** → **Wizards** → **connDir-LDAP.xml** and click the **Content** tab. Select **New Window** from the context menu in the editor window and scroll down to the step definitions.

```
<step name="General" title="Supply General Information"
illustrator="General" >
<description>
  The configuration of the Connected directory starts with the input
of some general information.
  Enter the data requested below. When done, click on the 'Next >>'
button to continue.
</description>
<propertypage reference="general" />
<prestep  class="siemens.dxm.wizard.PreStepImpl" />
<poststep class="siemens.dxm.wizard.PostStepImpl" />
</step>
```

This first step named "General" is displayed as "Supply General Information" in the window's title bar and linked to the illustrator step "General". Note that it is good style to name the step and illustrator the same. The description informs the user what to do in this step.

The propertypage reference points to "general". This is the name of the property sheet definition in the object description. Because we reference the connected directory object directly this must not be specified here.

The next two definitions implement the next and previous buttons in the wizard. Therefore, do not change it!

Now let's look at this definition:

```
<step name="AttrConfig"  title="Check Attribute Configuration"
illustrator="AttrConfig"
node="dxmAttrConfigFile-DN" >
<description>
  The system built up a set of attributes used for synchronization.
Check for completeness.
  The attribute properties beyond the name are mainly used for the
description of export and
```

```
  import file formats. See the documentation for details. Update the
attribute set if necessary.
  When done, click on the 'Next >>' button to continue.
</description>
<propertypage reference="attrconfdetails" />
<prestep  class="siemens.dxm.wizard.PreStepImpl" />
<poststep class="siemens.dxm.wizard.PostStepImpl" />
</step>
```

It is very similar to the previous one. The node definition is new. In this case, we use the propertypage "attrconfdetails" from a referenced object of the connected directory. This referenced object is defined via a distinguished name reference (node="dxmAttrConfigFile-DN"). This type of reference is very resistant to structural changes. It does not matter where the Attribute Configuration object resides - it will always be found via the DN reference.

# 8. Customizing Target Systems

This chapter describes how to:

- Configure a target system
- Define default values to be used in account creation
- Customize the default target systems supplied with DirX Identity
- Handle attribute-based access rights
- Use JavaScript in obligations
- Handle multi-value properties in obligations
- Inherit account attributes from users
- Manage target system accounts and group name spaces
- Set group member limits
- Specify unique account attributes and account naming attributes
- Create accounts automatically in a subtree
- Define an account or user attribute that holds a certificate

## 8.1. Configuring a Target System

A target system usually contains groups and accounts; this is the default for most of the systems.But you also may use target systems without accounts, or you may need to use another attribute instead of the common name to reference the account from the group.

Use the following attributes of the target system object to configure these situations:

**Assignment States**: Set to **true** if the account-group memberships are synchronized with a target system.In this case, the synchronization workflows need the state information in order to know which members to add or delete in the target system.

**Member property**: This attribute is only evaluated when no assignment states are stored; that is, **Assignment States** is set to **false**.In this case, it specifies the group attribute where the group members are stored.By default, DirX Identity uses **dxrGroupMemberOK**.

**Referenced Object Type**: This attribute specifies the object type that is to be stored as a group member.This attribute is usually the account; in this case, you select **SvcTSAccount**.Otherwise, users are the group members and you select **dxrUser**.

**Referenced property**: The attribute specified here is entered into the group **Member property**. It must be a valid attribute of the **Referenced Object Type**.

The following subsections provide some examples.

### 8.1.1. Configuring Virtual Target Systems without Accounts

When creating a new target system, DirX Identity offers the following templates for target

systems without separate accounts:

DirX Identity itself, which is perceived as a target system

The mailing list configuration

The mailing list configuration is an example meant to supply lists (distribution, signature, ...) to some applications that are only consuming and are satisfied with references to the user: the distinguished name (DN), an email address and so on. The members of the list are identified by their email address and are stored in the **dxrGroupMemberOK** attribute of the list:

Assignment States: false

Member property: dxrGroupMemberOK

Referenced Object Type: dxrUser

Referenced property: mail

The DNs of the list members are additionally stored in the **uniqueMember** attribute, which enables DirX Identity to update the list whenever email addresses have changed or when two users have the same email address.

For other target systems, it may be sufficient to store just the DN of the users. Configure this situation with the following options:

Assignment States: false

Member property: uniqueMember

Referenced Object Type: dxrUser

Referenced property: dn

If you synchronize the groups with another system, it is highly recommended to distinguish assignment or membership states. On the one hand, the exporting job must look for new members or members that are to be deleted in the other system by reading only the **dxrGroupMemberAdd** and **dxrGroupMemberDelete** attributes. On the other hand, the administrator easily identifies these members as well as those which are imported from the remote target system. Configure this situation with the following options:

Assignment States: true

Member property:

Referenced Object Type: dxrUser

Referenced property: dn

### 8.1.2. Configuring Virtual Target Systems with Accounts

This is the usual scenario and used for the real target systems such as Windows. These systems have accounts and groups that are both synchronized with DirX Identity.

Other target systems can exist where you don’t want to synchronize the groups; for

example, Microsoft Exchange 5.5. The distribution lists in Exchange are modeled as groups in DirX Identity and they are used in DirX Identity to express the right for a mailbox.Therefore, you may decide not to synchronize the distribution lists and not to distinguish assignment states.Use the following configuration to set up this situation:

Assignment States: false

Member property: dxrGroupMemberOK

Referenced Object Type: SvcTSAccount

Referenced property: dxrPrimaryKey

DirX Identity offers this as a template named “Exchange 5.5 MB”.

## 8.2. Defining Default Values for Account Creation

For each target system, you can define a set of default values that DirX Identity uses when it automatically creates accounts.Open the "Options" pane of the target system object and examine the pre-defined attributes.Enter appropriate values for each of them.

Each target system type offers its individual set of attribute types, which are described in the individual target system topics in this section.The following two attributes are used for all target system types:

- Account Root in TS - the root entry for accounts in the remote target system
- Group Root in TS - the root entry for groups in the remote target system

These attributes are set by the Initial Load workflow and updated by the validation workflows.

To define new default values for the target system, you must change the object description of the target system object and add a line for the new attribute.Navigate to the Object Descriptions subfolder of your target system’s Configuration folder and open the **TS.xml** entry.You will find a list of property definitions like this one:

```
<property name="dxrOptions(AccountRootInTS)" readonly="false"
label="Account Root in TS" multivalue="false" type="java.lang.String"
/>
```

Enter a new line for your new attribute.For more information about the property definition, see the section "Property Sheet and Property Page Elements".Your new attribute will appear in the target system object after you log out and log in again, or after you reload your object descriptions.

## 8.3. Customizing the Default Target Systems

This section provides information about customizing the following default target systems supplied with DirX Identity.

- LDAP target systems
- Windows 20xx target systems
- Windows NT target systems
- Exchange 20xx target systems
- Exchange 5.5 target systems
- RACF target systems
- Database target systems

### 8.3.1. Customizing an LDAP Target System

There are no special attributes for an LDAP target system. For general information about default values for a target system, see the section "Defining Default Values for Account Creation".

The account attribute **dxrPrimaryKey** is used as the **reference attribute** from the group to the account. The object description in DirX Identity and the synchronization workflows assume it holds the distinguished name of the account in the remote LDAP directory.

The same attribute with the same value is used for groups. This is necessary for handling nested group relationships.

### 8.3.2. Customizing a Windows 20xx Target System

For a Windows 20xx target system, you may provide default values for the following special Exchange 20xx attributes: Home MTA, Home MDB, Base OR Address (the prefix for the X.400 mail address) and Base Mail Address (the suffix for the RFC822 mail address), Home Server Name. For general information on default values for a target system, see the section "Defining Default Values for Account Creation".

The account attribute **dxrPrimaryKey** is used as the **reference attribute** from the group to the account. The Provisioning object description in DirX Identity and the synchronization workflows assume it holds the DN of the account in the remote Active Directory.

The same attribute with the same value is used for groups. This is necessary for handling nested group relationships.

### 8.3.3. Customizing a Windows NT Target System

There are no special attributes for a Windows NT target system. For general information on default values for a target system, see the section "Defining Default Values for Account Creation".

The account attribute *cn* is used as the **reference attribute** from the group to the account. The object description in DirX Identity and the synchronization workflows assume it holds the common name of the account in the remote Windows NT directory.

Windows NT groups may support a limited number of members. Enter this limit in the **Group Member Limit** attribute on the Options pane to register this information with DirX

Identity.

### 8.3.4. Customizing an Exchange 20xx Target System

To grant a user a mailbox in an Exchange 20xx server, you don’t have to create an Exchange 20xx target system in DirX Identity. Just use the Windows target system with the user’s local account.

For each Windows target system, DirX Identity provides a special group that is intended to hold all accounts that have the right for a mailbox. This group is named **dxr mailbox users**. By default, it is not synchronized to the Windows target system. When a user becomes member of this group, his/her account is automatically mailbox enabled. DirX Identity generates the email address and a number of other necessary attributes. Conversely, if an account is deleted from this group, the corresponding account attributes are reset.

This customization is achieved by configuring obligations. See the section "Handling Attribute-based Access Rights" for more information.

### 8.3.5. Customizing an Exchange 5.5 Target System

Mailboxes hold references to their associated Windows accounts, which are allowed to send and receive mails. Whenever DirX Identity creates a mailbox, it automatically searches for such an account within some of the DirX Identity target systems and sets the **associated NT Account** attribute in the mailbox.

DirX Identity administrators must specify the target systems that DirX Identity is to search. Open the Advanced tab of the Exchange target system object and use the target system navigator to enter at least one Windows system as an associated target system.

Navigate to those systems and enter their domain name in the **TS domain name** attribute in the Advanced tab of the object. DirX Identity needs this value to generate the correct value for the associated NT Account attribute.

Whenever DirX Identity creates a mailbox, it searches for an account of the user in the associated target systems, enters the reference to it in the **Associated Account (DN)** attribute and calculates the value for the "associated NT Account" attribute in the format *domain name***mailbox name*. It takes the domain name from the associated target system’s "Domain name" attribute that you specified in *TS domain name*. If DirX Identity successfully generates all mandatory attributes of the mailbox, it sets the boolean attribute **Object complete** to **true** to indicate this situation.

If DirX Identity does not find an associated account, you must set it by hand. Navigate to the target system and the account and set the attribute **Associated account (DN)**. DirX Identity again calculates the value for the associated **NT Account attribute** and sets the **Object complete** flag.

By default, DirX Identity uses the values you entered in the Options tab of the Exchange target system object (see "Defining Default Values for Account Creation") to generate the default values for other mailbox attributes. For an Exchange 5.5 system, these mailbox attributes are Home MTA, Home MDB, Base OR Address (the prefix for the X.400 mail address) and Base Mail Address (the suffix for the RFC822 mail address).

### 8.3.6. Customizing a RACF Target System

For general information on default values for a target system, see the section "Defining Default Values for Account Creation".

DirX Identity provides specific groups for the following specific segments in RACF target systems. These groups contain all accounts that are members of this segment.

| Segment | Group |
|---|---|
| TSO | TSO users |
| CICS | CICS users |
| WORKATTR | WORKATTR users |

When a user becomes a member of this group, his/her account is automatically segment-enabled. DirX Identity generates the segment-specific attributes. Conversely, if an account is deleted from this group, the corresponding account attributes are reset. This customization is achieved by configuring obligations. See the topic "Handling Attribute-based Access Rights" for more information.

For the TSO segment, you can provide default values for the following specific attributes:

- SAFAccountNumber
- SAFDefaultCommand
- SAFDestination
- SAFHoldClass
- SAFJobClass
- SAFMessageClass
- SAFDefaultLoginProc
- SAFLogonSize
- SAFMaximumRegionSize
- SAFDefaultSysoutClass
- SAFUserdata
- SAFDefaultUnit
- SAFTsoSecurityLabel

For the CICS segment, you can provide default values for the following specific attributes:

- racfOperatorIdentification
- racfOperatorClass
- racfOperatorPriority
- racfOperatorReSignon
- racfTerminalTimeout

For the WORKATTR segment, you can provide default values for the following specific attributes:

racfWorkAttrUserName

racfBuilding

racfDepartment

racfRoom

racf AddressLine1

racf AddressLine2

racf AddressLine3

racf AddressLine4

racfWorkAttrAccountNumber

For the other segments of RACF, you must configure the specific obligations.

The account attribute **racfid** is used as the **reference attribute** from the group to the account. The Provisioning object description in DirX Identity and the synchronization workflows assume it holds the **racfid** of the account in the remote RACF directory.

#### 8.3.6.1. Inheriting RACF Account Default Group Attributes from the User

In the RACF tab of a RACF group, you can set the parameter **Default Group Type** to one of the possible values of the organizationalUnit attribute of a user.

If a RACF group is assigned to a user, DirX Identity searches whether the value of the organizationalUnit attribute of the user matches the value of the Default Group Type of one of the groups. If the values match, this group is taken as the default group for the account to be created. If the value is not found, the **Account default group** of the Options tab of the target system object is taken.

#### 8.3.6.2. Password Handling

After creation of an account, RACF enforces the user to change the password at first logon.The synchronization workflow simulates this functionality.

During creation of an account, the synchronization workflow sets a default password for the account and then binds with change password.The password from the DirX Identity account object is used as the new password.The result is correct passwords for all new accounts.

### 8.3.7. Customizing a Database Target System

There are no special attributes for a database target system.For general information on default values for a target system, see the section "Defining Default Values for Account Creation".

The account attribute *cn* is used as the **reference attribute** from the group to the

account.The Provisioning object description in DirX Identity and the synchronization workflows assume it holds the common name of the account in the remote database.

## 8.4. Handling Attribute-based Access Rights

In some cases, access rights in a target system are not expressed by a mere group membership, but instead by appropriate settings of account attributes. An example of this is the Active Directory account, which is mailbox-enabled by setting the attributes mail address, home MTA, home MDB, and so on.In DirX Identity, on the other hand, an access right in a target system is modeled only by a group membership.Obligations help to keep group memberships and appropriate attribute settings synchronized.

Obligations are attached to target system groups in DirX Identity. They comprise operations to be performed when an account becomes a member of the group (OnAssignment) and when it is removed from the group again (OnRevocation). To discover existing attribute-based access rights, you need to define a validation filter that is applied to the account attributes. For details, see the section "Understanding Rule Types" in chapter "Managing Policies" in the *DirX Identity Provisioning Administration Guide*. If you have complex rules requiring JavaScript programming, see the section "Using JavaScript in Obligations" in this guide.

You can either use an existing group or create a new "virtual group", which only lives in DirX Identity and is not synchronized with the target system. Use existing groups if they fit with the intended privilege. Specify the OnAssignment and the OnRevocation operations and the validation filter for the group. Use a separate obligation object if several groups share the same operations and filters.Enter the common operations and filter parts into the obligation and the specific parts into each group. Don't forget to link the groups to the obligation. Ensure that there is a validation rule for this target system that includes the modified groups in its group filter.

When you define an obligation for a property that is not defined as a String in the Object Description of the base object, you have to set the type in OnAssignment/OnRevocation attribute manually. This follows the same syntax as in the Object Description.

*Example 7. Obligation for Integer attribute*

When using the editor on the obligation group, it will generate a property definition like this:

```
<properties>
    <property baseObject="SvcUser" name="myHierarchyLevel">
        <namingRule>
            <fixedValue value="0" />
        </namingRule>
    </property>
```

```
</properties>
```

By default, the type is implicitly set to java.lang.String. You have to change it to java.lang.Integer manually. The result should look like this:

```
<properties>
    <property baseObject="SvcUser" name="myHierarchyLevel"
type="java.lang.Integer">
        <namingRule>
            <fixedValue value="0" />
        </namingRule>
    </property>
</properties>
```

Removing a value, e.g. OnRevocation, will also not work using <clear> as this is not an Integer value and will cause an error. You have to set an empty fixedValue to remove the Integer value:

```
<properties>
    <property baseObject="SvcUser" name="myHierarchyLevel"
type="java.lang.Integer">
        <namingRule>
            <fixedValue value="" />
        </namingRule>
    </property>
</properties>
```

## 8.5. Using JavaScript in Obligations

If your rules for the attribute values to be set in the OnAssignment/OnRevocation actions are too complex to be described with standard naming rules, you can use JavaScript to calculate the values.

In this case, the OnAssignment/OnRevocation operations are merely used to trigger the invocation of a JavaScript program.

**Example:**

Let's assume you want to set multiple values of a multi-value attribute **sampleattribute**.Multi-value attributes are not handled with naming rules in obligations, so you use a trigger attribute - for example, **$obligationtrigger** - to initiate the invocation of a JavaScript program **setSampleAttribute**.For this purpose, the JavaScript program is

connected to a second pseudo attribute - for example, **$scripttrigger -** that depends on the **$obligationtrigger** attribute.

Then, in the OnAssignment action, define **$obligationtrigger** with the value **assign**.In the revocation action, define **$obligationtrigger** with the value **revoke**.These definitions allow distinguishing between assignment and revocation of the group.

The dollar sign (**$**) that precedes the attribute name prevents the attribute from being stored to the LDAP Server.

The account's object description contains the following property descriptions:

```
<property
   name="$obligationtrigger"
   type="java.lang.String"
   />
<property
   name="$scripttrigger"
   type="java.lang.String"
   dependsOn="$obligationtrigger"
   defaultvalue="$(script:setSampleAttribute)" >
   <script name="setSampleAttribute"
            return="returnValue"
reference="storage://DirXmetaRole/cn=setSampleAttribute.js,
cn=JavaScripts,…?content=dxrObjDesc"/>
</property>
<property
   name="sampleattribute"
   type="java.lang.String"
   multivalue="true"
   />
```

The JavaScript program writes multiple values **v1**, **v2**, **v3** into **sampleattribute** on assignment and removes the values on revocation. Change this script according to your needs:

```
// sets the value for sampleattribute
importPackage(java.lang);
importPackage(java.lang.reflect);
obj=scriptContext.getObject();
var operation=obj.getValue("$obligationtrigger");
var returnValue = "done.";
```

```
var value;
if (operation.equals("assign")) {
  value = java.lang.reflect.Array.newInstance(java.lang.String, 3);
  value[0] = "v1";
  value[1] = "v2";
  value[2] = "v3";
}
else {
  value = java.lang.reflect.Array.newInstance(java.lang.String, 0);
}
obj.setProperty("sampleattribute ", value);
```

## 8.6. Handling Multi-value Properties in Obligations

If you want to set multi-value properties in the OnAssignment/OnRevocation actions, you can use a special syntax for the values:

- <add> - use this prefix to add a specific value.
- <delete> - use this prefix to delete a specific value.
- <clearall> - use this prefix to clear the property first (only allowed at the beginning).

**Examples:**

To illustrate this syntax, let's use the same actions given in the section "Using JavaScript in Obligations":

`OnAssignment: "<clearall><add>v1<add>v2<add>v3"`

After OnAssignment, the property contains the three given values. The <clearall> prefix takes care of clearing any values that already exist, so in every case after OnAssignment, we have exactly these three values.

`OnRevocation; "<clearall>"`

After OnRevocation, the property is empty.

Now let's suppose that we want to add two specific values during assignment and then remove these values during revocation:

`OnAssignment: "<add>v1<add>v2"`

The given two values are added to the values of the property that already exist.If a value already exists, the value is ignored.Values that already exist are also present after OnAssignment.

`onRevocation: ""<del>v1<del>v2"`

The two given values are deleted from the property.The values are ignored if they are not present.After revocation, these two values are no longer present.All other values are still present.

## 8.7. Inheriting Account Attributes from the User

An account entry usually inherits some attributes from the associated user, sometimes just as a default value when the account is created (see the section "Defining Default Values for Account Creation").Sometimes the account attributes change when the corresponding user attribute changes.In other cases, the account attribute has a different name from the user attribute, but has the same or a deferred value.DirX Identity offers some features that help in such situations.

To simply inherit a user attribute with the same name, use the **master** key.It is used as the default value when the account is created and is updated each time the user attribute changes and DirX Identity either edits the user or the account.Typically, this operation is performed regularly by the RBAM agent when it performs consistency checks.

In the following example, the account inherits the surname from its associated user:

```
<property
     name="sn"
     label="surname"
     type="java.lang.String"
     mandatory="true"
     master="dxrUserLink"
     />
<property
     name="dxrUserLink"
     label="user DN"
     type="java.lang.String"
     />
```

The **master="dxrUserLink"** instruction directs DirX Identity to get the attribute value of the **sn** property from the object referenced by the link in the attribute **dxrUserLink**. This is the corresponding user object. Note that the property must have the same name in both objects.

To inherit a user attribute with another name is a little bit trickier. The **dependson** key helps here, because it allows you to specify a dependency between one attribute in an entry and another attribute of the same entry. Each time the original attribute value changes, the naming rule of the dependent attribute is executed.

As an example, assume that attribute **displayName** stores the account's surname and given name, which in turn should be taken from the corresponding user attributes. The display name is re-calculated whenever the surname or the given name of the user

changes.

The following property definition ensures that the attribute is taken

```
<property name="sn" master="dxrUserLink" ... />
<property name="givenname" master="dxrUserLink" .../>
<property name="displayName"
label="Display Name"    type="java.lang.String"
dependsOn="sn, givenname">
<extension>
 <namingRule>
     <reference
         baseObject="SvcTSAccount"
         attribute="givenname"
      />
     <fixedValue value=" "/>
     <reference
         baseObject="SvcTSAccount"
         attribute="sn"
      />
 </namingRule>
</extension>
</property>
```

## 8.8. Managing Target System Accounts and Group Name Spaces

Different target systems usually require their individual name spaces for their accounts and groups.Each target system type has its individual attributes - for example, Active Directory Service (ADS) recognizes a guid, a surname and a given name, while Windows NT does not - and you may want to use specific rules for generating account names, passwords or other attributes.

To manage target system-specific name spaces, DirX Identity supplies object and property page descriptions that are specific to each target system type.When you create a new target system, type-specific defaults are created and located in the Configuration folder for the target system.

When it reads an account or creates a new one, DirX Identity uses the DN of the entry and matches it to the **mapping** element in each object description to determine which object description to use for the account.Here is an example:

```
     <mapping>
```

```
          <attribute objectclass="{any}dxrTargetSystemAccount"/>
          <attribute dn="*,$(./../../..)"/>
     </mapping>
```

The example **mapping** element directs DirX Identity to choose this object description if the entry in question contains the **dxrTargetSystemAccount** object class (which is valid for all accounts) and contains a part of the object description's DN. This is expressed through the variable $(./../../..), which is to be read analogous to file system notation: $(.) is the DN of the object itself, $(./../) is the DN of its superior, and so on. For an account object description, $(./../../..) is the DN of the target system entry and is replaced by it when the object description is loaded.

Please do not change these **mapping** elements.

Frequently, DirX Identity must create an account automatically when it detects that a user has received an access right (a group assignment) for a target system where he hasn't had one before. Then it must create the mandatory attributes (the unique name of the account, and perhaps the surname and a password) without administrator interaction. Therefore, you must provide appropriate default values or naming rules for DirX Identity to use.

Attributes such as "surname", "givenname" or "telephonenumber" usually receive the same value as the associated user. Furthermore, if the user's attribute value changes, the value in the account must also change. Here you should use the **master** attribute of the **property** element in the object description. You'll find an example for the surname in the section "Properties and Property Elements" in the chapter "Customizing Objects".

For other attributes, the same default value may apply for all accounts; for example, the user profile attribute "dxrPwdNeverExpires". Here you can use the **defaultvalue** inner element of the **property** element:

```
<property name="dxrPwdNeverExpires"
     type="java.lang.Boolean"
     defaultvalue="false"
     />
```

Naming rules must be specified for the generation of the account name and perhaps the password.Naming rules can be simple, in XML notation, or complex, in JavaScript.For more information, see the subsections on naming rules in the section "Customizing Provisioning Object Property Descriptions" in the chapter "Customizing Objects".

## 8.9. Setting Group Member Limits

Some target systems allow only a limited number of members in their groups. DirX Identity offers features to manage these situations.

The target system object offers an attribute **Group Member Limit** where you can enter the limit of allowed group members for this target system. DirX Identity then observes this

limit. When it needs to enter a new member into a group, which causes exceeding the limit, it creates an **extension** group and enters the new member there. The extension group becomes a "member" (an extension) of the "master" group, so that all of its members get the same rights as those in the master group. The logic extends the name of the master group with a counter starting with "001".

The extension groups are not offered for group assignment and thus do not become groups in their own right. In the user views, DirX Identity displays the extension group members as if they were in the master group. So the user administrator does not even notice the extension groups. But they are shown in the group objects, both in the **Member** and **Member of** tabs. Their assignment state is displayed as **EXTENSION**.

Note that this feature is only implemented for target systems where the members in DirX Identity are stored at the groups, not at the accounts.

# 8.10. Specifying Unique Account Attributes

Some target systems require an attribute value that is unique within all accounts or even unique within all accounts and groups (as RACF does).

With the XML attribute **uniqueIn**="..." you specify for a property the root of the subtree at which the property value must be unique.It uses the relative path notation "../.." beginning with the object description entry to navigate to the root.See the object description for the target system type RACF for a sample, which denotes the target system node as the root.

Before storing the new entry, the system checks the uniqueness of the calculated property value.If it is not unique, you need some mechanism to re-calculate the value until it's unique.DirX Identity provides a retry mechanism using a configurable retry counter, a virtual property, to support this requirement.The retry counter virtual property can be read in Java scripts and naming rules via obj.getTries() in order to calculate unique values.

What do you need to configure?

In the property description of your unique property, you reference the counter using the XML attribute **dependson=**....

Here is a sample snippet of the object description:

```
<property   name="$racfIdCount"
     label="RacfID count"
     type="java.lang.Integer"
/>
<property name="racfid"
  uniqueIn="$(./../../..)"
 dependsOn="$racfIdCount"
 applyDependsOn="save"
 ...
```

```
/>
```

This configuration specifies the property **racfid** to be unique within all accounts and groups of the target system. The retry counter is the virtual property **$racfIdCount**. The additional option **applyDependsOn="save"** tells the system to apply the property dependency not on each change, but only before the whole entry is to be stored.

Note that you can define a list of comma-separated properties on which your property depends. Make sure that the counter is the first one in your list.

If storing the entry or the **uniqueIn** check fails, the system recalculates the property values. It automatically increments the retry counter (in the previous example, **$racfIdCount**) and invokes the naming rule or JavaScript program again. The script or rule must use the counter property to generate a unique value.

Here is a JavaScript code snippet that shows how to read the property:

```
...
var triesProperty = obj.getProperty("$racfIdCount");
var tries = 0;
if (triesProperty != null) {
tries = triesProperty.intValue();
}
if (tries > 100) {
// JavaScript.Error stands for "no success"
racfId="JavaScript.Error";
}
...
```

It reads the counter property in string format and then transforms the string to an integer value.If a maximum number of retries has been reached, it notifies the system of an error by storing the string **JavaScript.Error** into the return variable **racfid**.See the appropriate script for the target system type RACF as a full example.

## 8.11. Specifying Unique Account Naming Attributes

Sometimes naming rules result in the same value of an account's naming attribute cn for different users.In this case, you need a mechanism to re-generate the value of the naming attribute until it is unique.

If the system cannot create an account because one already exists with the same naming attribute, it retries it using another retry count.For such scenarios, you need to generate the attribute value using a Java script.The system calls the same script, but the object's getTries() method returns a higher retry count.

This approach gives a lot of flexibility for generating unique values.

The simplest approach is just to compose the value by including the retry count.See the following sample snippet:

```
var tries = obj.getTries();
account=sn.substring(0,1) + tries + givenname;
```

The next snippet shows another alternative, which includes an increasing number of surname characters into the attribute value:

```
var tries = obj.getTries();
if (tries == 0) {
account=sn.substring(0,1);
}
else if (tries < sn.length()) {
account = sn.substring(0,tries+1);
}
else {
account = sn + tries;
}
```

If the number of retries exceeds a given limit, you can cancel the loop.See the following snippet:

```
if (tries > 32) {
account="JavaScript.Error"
}
```

The return value **JavaScript.Error** indicates an error and the system cancels account creation and displays an error message when in interactive mode or raises an exception when in batch mode.

## 8.12. Creating Accounts in a Subtree

For some target systems, you may want to have new accounts automatically created in a subtree, not directly underneath the accounts' base node.To set this up, you must to specify naming rules for the account’s distinguished name (DN).

The virtual attribute **$superior** helps here.It denotes the DN of the supposed parent node of the new account.Use a Java script to calculate an appropriate value using other attributes of the user or account.The system tries to create the new account underneath the assumed parent.If this operation fails because the parent does not yet exist, the system automatically creates this parent.If this operation also fails for the same reason, it repeats this until a

parent exists.See the sample property description for the default target system.

## 8.13. Defining Attributes that Hold Certificates

To define a user or account attribute that holds a binary certificate:

- Define the attribute in the object description of the user and/or the account as follows:

  ```
  <property name="userCertificate" type="[B" label="Certificate"
  editor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapCertificate"/>
  ```

  The **[B** type notation denotes the Java internal notation of a byte array.

- Use this definition in the user object description as well as in the account object description. If you want the account to hold the same certificate as the user, make sure the attribute names are the same and include the XML attribute master in the account object description as follows:

  ```
  <property name="userCertificate" type="[B" label="Certificate"
  editor="siemens.DirXjdiscover.api.ldap.beans.JnbLdapCertificate"
  master="dxrUserLink"/>
  ```

# 9. Customizing Auditing

This chapter explains how to customize DirX Identity's auditing features, including:

- Query folders
- Audit policies
- Certificate owner check
- Status reports

## 9.1. Creating a Query Folder

Query folders are a type of persistent LDAP query.They are displayed as folders in the DirX Identity Manager tree and - when opened - perform an LDAP search using the search parameters specified the query's Filter fields, and displaying the result as nodes below the query folder.In DirX Identity, query folders are primarily used to display objects that are in an "exceptional" (inconsistent or error) state in a prominent location in the tree.As a result, you can use query folders for auditing or maintenance tasks.

The DirX Identity installation provides sample query folders in both Provisioning and Connectivity views.In the Provisioning view group, the sample query folders are located at the root of the user subtree and in each target system's subtree.They can be used, for example, to search for and display:

- Inconsistent users
- Users with values in their **To do** or **Error** fields
- Users that are marked as "to be deleted" (State=TBDEL)
- Unassigned accounts (an account without an associated user, for example, system accounts)
- Active and inactive rules and policies

In the Connectivity view group, the sample query folders are located in the Expert and Monitor views. They can be used for example to search and display:

- Active or inactive workflows and schedules
- Workflow instances with errors or warnings

You can copy these query folders and modify the copies to create your own set of queries.(Changing the sample query folders is not recommended, because the changes you make will be overwritten with each DirX Identity update.).You can also create new query folders according to your requirements.

A query's LDAP search criteria consists of the following parameters (which are displayed in the Filter tab when you open a query folder in DirX Identity Manager):

- The search base
- The search scope (see the online help for details)

- The search filter, in LDAP notation.Refer to RFC 2254, "A String Representation of LDAP Search Filters" for further details.An extension is provided to allow for building timing constraints (see the online help for details).
- The maximum number of entries to be returned as a result; when this limit is reached, DirX Identity terminates the search and notifies you of this exception.

# 9.2. Customizing Audit Policies

The section "Managing the Audit Trail" in the "Managing Auditing" section of the *DirX Identity Provisioning Administration Guide* describes how to manage and customize audit policies.

# 9.3. Customizing the Certificate Owner Check

This section describes

- recommendations and hints for implementing a certificate owner check
- the matching algorithm of DirX Identity's built-in certificate owner check.
- how to write your own custom certificate owner check.

**Recommendations and hints**

Implementing a secure certificate owner check requires to follow some guidelines:

- First check your certificates which user attributes they contain.
- If a GUID is contained (a global unique identifier), use this attribute as check parameter. Such attributes are defined by automatic algorithms and users are not allowed to change this identifying attribute. We highly recommend that only the GUID creation process should have the directory access right (ACI) to write this attribute.
- If no GUID attribute is available, you can use another unique attribute. In many cases the mail attribute is used. This means that you must protect this attribute against unauthorized changes. A process must be in place that defines this attribute, adds it to the certificate during creation and stores it at the user entry. Only this process should have the directory access right (ACI) to write this attribute.

If you follow these recommendations, your certificate owner check is secure.

Built-in Certificate Owner Check Matching Algorithm

DirX Identity supplies a default verifier, that uses the following algorithms to identify the certificate owner:

- The attributes are checked against attributes being read from the X509Certificate Object using the method **getSubjectAlternativeNames()**.
- If no matching value is found, the certificate owner's **DN** components are checked. The check is done case-insensitive, without naming attributes.
- In case **DN** is used as attribute name, the signer's DN is compared to the DN extracted

from the certificate by **getSubjectDN().getName()**.

- The DNs are normalized and compared case-insensitive. Since certificates often contain OIDs, only the DN component values, not the naming attributes, are used in comparison.

In case multiple attributes are entered in 'Attributes to check', each attribute value has to match (AND condition).

Custom Certificate Owner Check Matching Algorithm

If the default verifier does not meet the customer's requirements, he may write an own verifier and enter its fully classified class name in the **Custom Owner Verifier Classname** field at the domain object (Auditing tab). Note that the custom verifier has to implement the Interface **com.siemens.dxm.api.audit.SignatureOwnerVerifier**.

It contains the methods

**setSigner(ImmutableEntry signer)** - passes the logged-in user to the verifier

**setParameters(String parameters)** - passes the 'Attributes to check' to the verifier

**verify(X509Certificate certificate)** - called to verify the owner.

Java interface definition:

```
package com.siemens.dxm.api.audit;
/*
* DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
*
* Copyright (c) Atos 2011
* All rights reserved.
*
* This software is the confidential and proprietary information of
* Atos IT Solutions and Services GmbH ("Confidential Information").
* You shall not disclose such Confidential Information and shall use
* it only in accordance with the terms of the license agreement
* you entered into with Atos IT Solutions and Services GmbH.
*
* END OF COPYRIGHT NOTICE.
*/

import java.security.cert.X509Certificate;
import com.siemens.dxm.api.entry.ImmutableEntry;
/**
* This interface provides the methods to verify the signer owns the
certificate
```

```
*/

public interface SignatureOwnerVerifier {
/**
* sets the person who signed the data
*/

public void setSigner(ImmutableEntry signer);
/**
* sets the parameters for verification (e.g. the signer's attributes
to be verified)
* @param parameters
*/

public void setParameters(String parameters);
/**
* verifies that the correct certificate was used for signing, i.e.
the signer is the owner of the certificate.
* @param certificate
* @return
*/

public boolean verify(X509Certificate certificate);
}
```

## 9.4. Customizing Status Reports

DirX Identity status reports represent a powerful generic mechanism for extracting and viewing information about the current state of objects in the DirX Identity store, particularly bulk information on selected DirX Identity objects and their relationships to other DirX Identity objects.Status reports do not provide history information.

A report consists of an XML configuration for extracting data from the DirX Identity store and an XSLT program that transforms the extracted data.DirX Identity provides a set of pre-configured, ready-to-use reports that return information on the DirX Identity Provisioning and Connectivity objects and their related objects in a format that is tailored to the way these objects are presented in DirX Identity Manager and Web Center.

Reports can be displayed in DirX Identity Manager and Web Center (only for Provisioning objects), and they can be downloaded to files.They can be generated on demand from DirX Identity Manager and Web Center, and they can be generated automatically by DirX Identity workflows that run at scheduled intervals.

In the Provisioning view group, access policies can be used to control who is allowed to run

a given report; for example, to permit only those users who are assigned an auditor role to run a report that checks for SoD exceptions in the DirX Identity store, or to permit a specific group manager to run user-role assignment reports only on the members of his group.

The pre-configured reports provided with DirX Identity are designed to be highly customizable.You can copy these pre-configured report templates and pre-defined subroutines and then modify them to your requirements using XML and XLST/XPATH.

Reports in DirX Identity also support the concept of **virtual attributes**, which represent complex DirX Identity logic for values displayed in DirX Identity Manager or Web Center. A virtual attribute is a value displayed in DirX Identity Manager or Web Center that represents the result of multiple LDAP attribute searches and a computation made on these attributes.

The next sections provide:

- General information about how reports work and how to customize them to your requirements
- Information that is specific to customizing provisioning reports
- Information that is specific to customizing connectivity reports

### 9.4.1. Status Report Architecture

A status report is partitioned into the following elements:

- A producer, which makes the basic supply of DirX Identity objects of interest available to the selector.
- A selector, which defines the attributes to load, handles the relationships these objects may have with other DirX Identity objects (triggering additional LDAP searches to load the related objects) and creates an intermediate XML output that is passed to the transformer.
- A transformer, which creates the desired format (by applying an XSL transformation, in most cases) and passes the result to the consumer.
- A consumer, which typically stores the result in one or more files, depending on its size.

The following figure shows the data flow between these report elements.

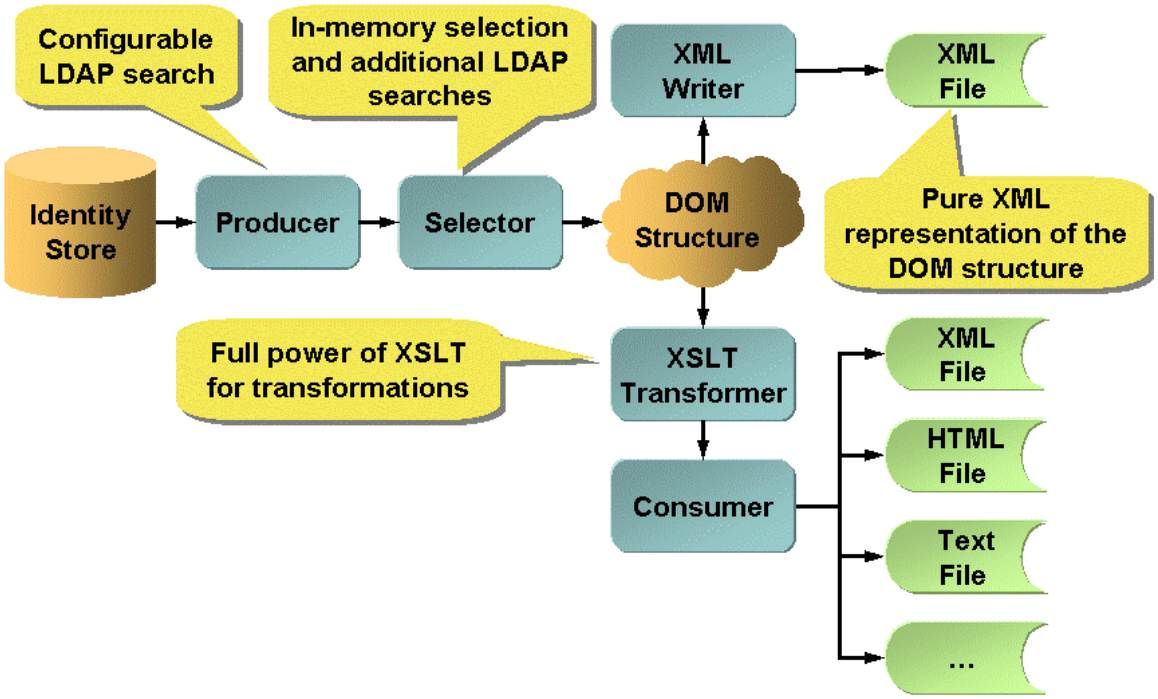



*Figure 6. Report Data Flow*

When displayed in DirX Identity Manager, a report object consists of the following tabs:

- A **General** tab, which contains a **Types** field that determines whether or not the report is made available when right-clicking an object and selecting **Report**.
- A **Content** tab, which stores the report's XML definition. The **Content** tab contains the producer, selector and consumer definitions.
- A **Format** tab, which stores the XSL transformation that turns the output of the selector that is defined in the XML into the desired result.

### 9.4.2. Running Status Reports

DirX Identity allows you to run status reports dynamically from DirX Identity Manager and Web Center and to schedule automatic runs of pre-defined reports.

From DirX Identity Manager, you can:

- Select an object or a folder and select from a set of status reports that can be generated for the object
- Select a report, and then select the object or folder (by specifying the search base) on which the status report is to be generated

(For Provisioning objects only) From Web Center, you can:

- Select a single object and then select from a set of reports that can be generated for this object
- Select a single tree object and then select from a set of reports that can be generated

for this object

- Select a folder in a tree and then select from a set of reports that can be generated for this object

DirX Identity provides provisioning and connectivity status report agents that you can set up to run a report automatically at scheduled intervals. These agents actually consist of a workflow with one activity - the status report agent - that runs in the C++-based server. To set up a scheduled status report, you start one of the status report agents, and then supply the information about the report you want to produce. The report agent will then run the report at a scheduled time.

### 9.4.3. Matching Status Reports to Objects

The **Types** field in the report object's **General** tab determines whether or not the report is made available in a context menu when the object specified in the field is selected (right-clicked). The class attribute of an **action** element in the corresponding object description specifies the reports that can be run for the object. The following figure illustrates this relationship.

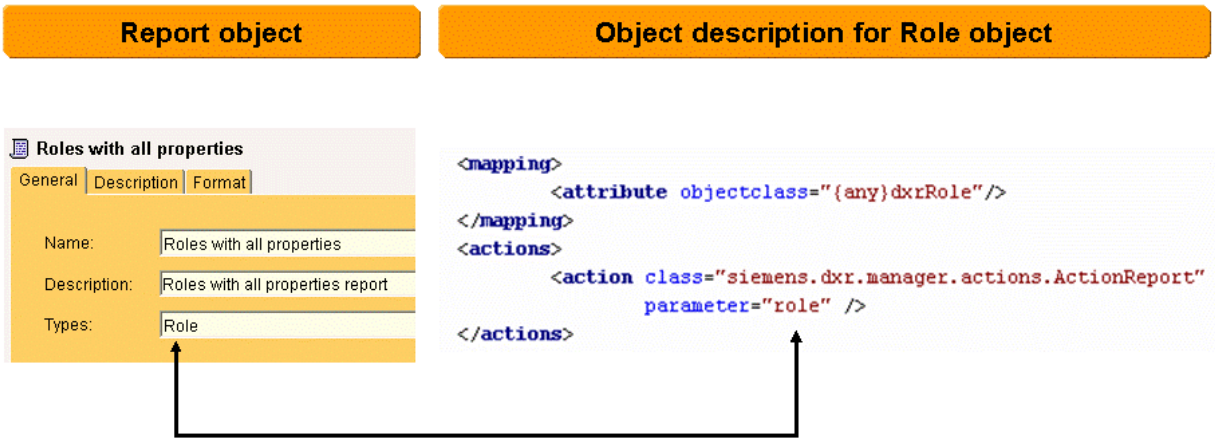



*Figure 7. Report Object and Object Description Matching*

For example, the user object description role.xml (**Provisioning** → **Domain Configuration** → **Object Descriptions > role.xml**) contains the following action class definition:

```
<action class="siemens.dxr.manager.actions.ActionReport" parameter="role" />
```

This definition specifies that all reports whose **General** > **Types** field specify **Role** are considered to be reports that are suitable for role objects. If you change the report's **Type** field to specify **Roles** without changing the object description's action class definition accordingly, the report will no longer be considered a role report.

The **role.xml** object description does not apply to role objects because of its name.Instead, it applies to them due to a clause such as:

```
<mapping><attribute objectclass="{+any+}dxrRole"/></mapping>
```

This clause specifies that this object description applies to all objects where one of its object classes is objectclass=dxrRole.If the prefix "(any)" is omitted, dxrRole must be the only object class.

### 9.4.4. Creating or Changing Status Reports

You should not change the standard connectivity and provisioning status report objects provided with DirX Identity. Instead, you should copy and modify them to create your own new report objects. Although you may want to completely change a report, the most likely changes you might want to make are:

- Changing the search base to reduce the output volume.
- Changing the LDAP search filter to reduce the output or to search for objects with special characteristics.
- Changing the attributes to be displayed with each object.
- Changing the output format and output files.

To create your own status report from the templates provided with DirX Identity:

- Look for a report definition that most nearly matches your requirements
- Copy this report
- Change the producer as you need it
- Produce an XML format to a file to check and understand the XML structure (are all objects and attributes contained?) For performance reasons, check also that no objects and attributes are contained that you do not need.
- Adapt the transformer routines step by step
- Add attributes clauses to display all attributes
- Check for helpful existing sub-routines

### 9.4.5. Status Report Content Tab

A status report's **Content** tab displays the report's object description, which has the following structure:

```
<report>
<import file=.../>
<producer> ... </producer>
<selector> <entry> ... </entry> <entry> ... </entry> ... </selector>
<html><transformer> ... </transformer><file><consumer> ...</consumer>
</file></html> and/or
<txt><transformer> ... </transformer><file><consumer> ...</consumer>
</file></txt>       and/or
<xml> <transformer> ... </transformer><file><consumer> ...</consumer>
```

```
</file></xml>   and/or
<another output type> <transformer> ...
</transformer><file><consumer> ...</consumer> </file></another output
type>
</report>
```

A status report object description begins with an optional import element, followed by a producer, followed by a selector, followed by one or more output-related clauses such as html, txt or xml.

The next sections describe this report structure in more detail using a sample report as it is used or as it could be used for user objects as an example. The following figure shows this report's Content tab in DirX Identity Manager.


*Figure 8. Example Content Tab*

#### 9.4.5.1. Producer Element

A producer defines an LDAP search in XML. The **producer** is a Java class that searches for objects in the DirX Identity store that match an LDAP search and passes the search result to the selector. You define for the producer which objects to search, while you define for the selector the attributes to extract from these objects; for example, the state of a role in addition to its name.

DirX Identity supplies two producer Java classes (other producer Java classes, including customer-supplied ones, are basically possible, too):

- ObjectEnumerationProducer - the standard producer that can perform any LDAP search. It uses the "virtual list view" or the "simple paged results" control (if the server supports them). These controls allow the client to handle large search results in a resource-saving way.
- UnassignedRoles - a specialized producer that is intended to be used in reports that list roles that are not assigned to any user.

Here is an example of a producer:

```
<producer
class="siemens.dxm.storage.report.ObjectEnumerationProducer">
<search base="cn=users,$(rootDN)" filter="$(SearchFilter)"
scope="$(SearchLevel)"  sizelimit="$(MaxNumberOfEntries)"
attributes="$(RequestedLdapAttributes)" order="$(SortOrder)"
/>
</producer>
```

The *class* attribute determines which producer to use.

The *search* clause expresses an LDAP search request. This example uses a number of variables that apparently are defined elsewhere; therefore this producer appears to presume an import element that imports them. For example:

```
<import file="storage://DirXmetaRole/cn=Settings for
Producer/Selector,$(./..)?content=dxrObjDesc"/>
```

In this example:

- The import element addresses a directory object whose last RDN is **cn=Settings for Producer/Selector**.
- **$(./..)** specifies that this object is located below the same node as the object that defines the report.
- **content=dxrObjDesc** specifies that the variables are found in the attribute dxrObjectDesc of this object. This is why the tab that presents this information in the DirX Identity Manager is called "Content".

#### 9.4.5.2. Import Element

The import element is intended to include additional code for example a selected set of variables that are defined elsewhere. The keyword include can also be used as an alternative to import.

#### 9.4.5.3. Selector Element

The **selector** is a Java class that allows you to restrict the amount of data extracted from the DirX Identity store by the producer. It also allows you to expand the amount of data extracted beyond the capabilities of LDAP searches. The use of multiple entry clauses permits you to structure the selector to satisfy relationships that the objects provided by the producer may have with other objects; for example, to create a status report on users and attributes of their assigned roles. (Note that these attributes are role attributes, not user attributes.) The selector provides the data in an intermediate XML format from which any desired output format can be (preferably XSL-) transformed.

DirX Identity provides the following selector Java classes (customer-supplied selectors are also possible):

- dom.SelectorImpl. - the standard selector
- dom.ScriptSelectorImpl - allows you to write your own selector in JavaScript. Since this typically implies programming internal, unpublished interfaces, its use is rather limited.

Here is an example of a selector that refers to the producer example given in the "Producer Element" section.

```
<selector class="siemens.dxm.storage.dom.SelectorImpl">
<entry>
<match><property name="$odname" value="dxrUser" /></match>
<property name="attributes"
value="$(RequestedLdapAttributes),$(RequestedVirtualAttributes)" />
<attr/><value/>
<reference maxDepth="0"><match><property name="name"
value="$(DNsToBeDereferenced)"/></match></reference>
</entry>
<entry>
<match><property name="$odName" value="dxrRole"/></match>
<property name="attributes" value="$(RequestedRoleAttributes)" />
<attr/><value/>
</entry>
<entry>
<match><property name="$odName" value="svcGroup"/></match>
<property name="attributes" value="$(RequestedGroupAttributes)" />
<attr/><value/>
</entry>
</selector>
```

The **class** attribute determines which selector to use.

Within the **entry** clause:

- The **match** element filters entries from the producer's input stream by name and by value (wildcards are supported)
- The **attributes** element selects attributes or virtual attributes
- The **reference** element follows references to directory subentries
- The **child** element follows references to one level of subentry
- The **entry** element defines the details of the subentry (child or referenced object)

In the example provided here:

- The first **entry** clause selects most of the data that the producer has provided and some additional attributes that are subsumed in a variable called RequestedVirtualAttributes. This entry clause filters out - based on the first *match* clause and the *property* clause- all entries from the producer's output that match the object description with "<object name="dxrUser">" (in this example, **Provisioning** view > **Domain Configuration** > **Customer Extensions** > **Object Descriptions** > **User.xml**) and adds the attributes named in the variable RequestedVirtualAttributes as additional attributes.
- The **attr** clause specifies that the *attribute names* are to be provided.
- The **value** clause specifies that the *attribute values* are to be provided.
- The **reference** clause leads to the **entry** clauses that follow; it lists the DNs that are to be de-referenced and expects details in these **entry** clauses. To avoid loops, you should always define the depth to which the references should be followed. In this example, maxDepth="0" specifies that only one level is followed.

In order for the second **entry** clause to be useful, RequestedVirtualAttributes and DNsToBeDereferenced must contain the name of the virtual attribute **Roles.Assigned**. This attribute provides DNs of object class dxrRole and the second **entry** clause specifies the attributes that are to be provided in addition to these DNs. If the reference clause is not present, this entry clause will not match anything, since the producer does not produce any role objects. If the match clause is missing within the reference clause, all references that have a matching entry clause are resolved.

The third **entry** clause follows the same principles, but it manages the virtual attribute **Groups.Assigned** rather than *Roles.Assigned*.

The following figure illustrates another example of a selector.

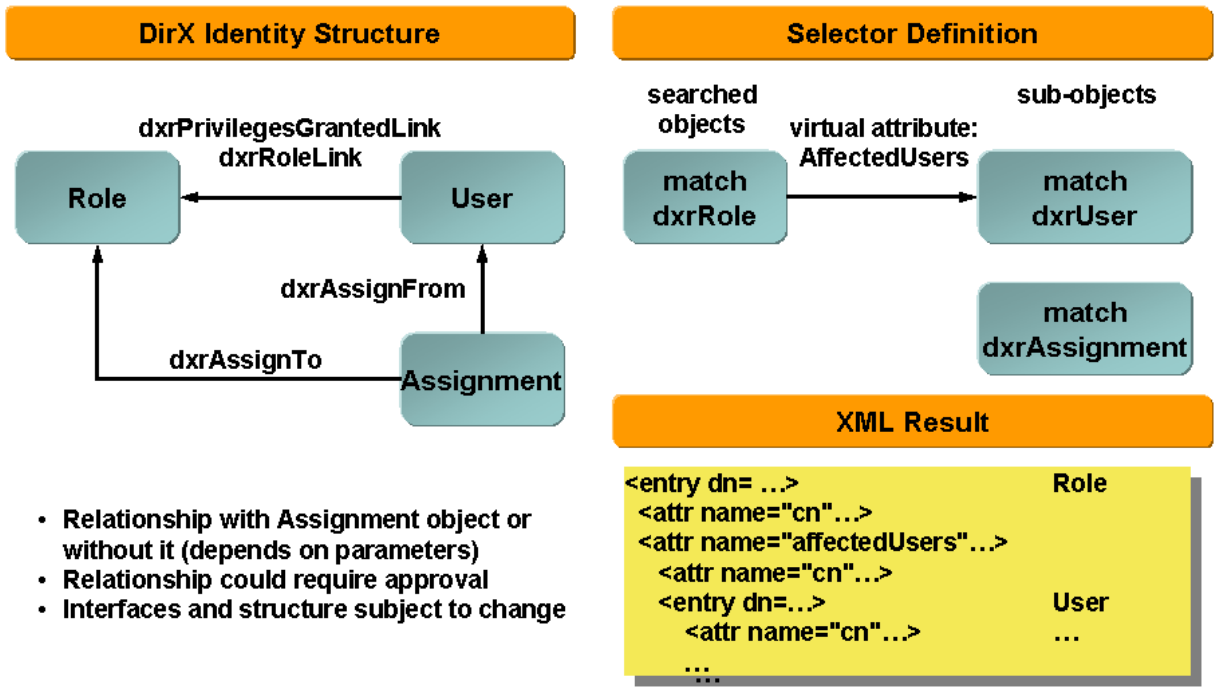



*Figure 9. Example Selector*

In the figure, the left side shows the DirX Identity object structure and the links between the objects. The right side shows a possible selector definition. The basic objects to search are role objects (this is defined in the producer). The assigned user objects can be found via the virtual attribute Affected.Users. This virtual attribute completely hides the complexity of the DirX Identity object structure. This is also important if the object structure must be changed. The virtual attribute will hide this change completely. The link from the User objects to the Assignment objects is followed by the reference statement (not visible in the figure).

Here are some hints about selectors:

- Deleting lines such as **<property name="attributes" value="$(RequestedLdapAttributes),$(RequestedVirtualAttributes)" />** is helpful for test purposes, since this deletion causes *all* attributes - including *all or almost all* virtual attributes - to be revealed in the XML output.
- A selector expects the producer to pass in "storage objects", which represent directory entries enhanced by the specifications in the corresponding object descriptions. If no object description can be mapped, a generic object description is created. We recommend that you provide well-designed object descriptions for all objects.
- A selector may comprise several **entry** clauses. Their order is significant if they refer to the same storage object. In this case, the first **entry** clause "wins". Subsequent **entry** clauses often appear not to match anything produced by the producer. This may make sense under the following conditions:
  - If a previous **entry** clause uses a **reference** clause, as it does in the example. This causes the references to be resolved, as far as matching **entry** clauses are there.

- If a previous **entry** clause uses a **child** clause. This causes all one-level sub-entries to be resolved, as far as matching **entry** clauses are there.

entry clauses only make sense if **reference** or **child** clauses in previous entry clauses are present.

- Values can be specified as comma-separated lists. An asterisk (*) can be used as a wild card.
- The property in a **match** clause such as "<entry><match><property name="$odname" value="dxrUser" /></match>..." can use any attribute name as reported in the XML report. For example, if an XML report contains the following snippet:

```
...
<attr name="dxrState" ><value>ENABLED</value></attr>
...
```

you can specify "<entry><match><property name="dxrState" value="ENABLED" /></match>..." (which means that all produced entries whose dxrState attribute has the value ENABLED will be selected). Or you can specify "<entry><match><property name="dxrState" value="*" /></match>..." (which means that all produced entries whose dxrState attribute has any non-empty value will be selected).

#### 9.4.5.4. Transformer Element

A transformer processes the data from the selector into an output format to be processed by the consumer. The section "Report Format Tab" provides more information about this attribute.

DirX Identity provides the following transformers:

- XMLWriter - use this transformer to determine the data and the data format that the selector provides. This transformer operates on the pure XML-formatted output generated by the selector.
- XSLTWriter - use this transformer to process XSL transformations (XSLT). The programming languages used in this transformer are XSLT and XML Path (Xpath).
- ScriptWriter - use this transformer to write a custom transformer in JavaScript. Using this transformer implies programming internal, unpublished interfaces. Its use is therefore rather limited.

XSL transformation features include:

- Support of the import element to structure report definitions
- Structuring through a variable concept (to separate configuration)
- Structuring through XSLT subroutines with parameters

#### 9.4.5.5. Consumer Element

A consumer defines the output to files or to a simple viewer. Output formats include XML, RAW, HTML, and TXT. The section "Output Formats" provides more information about these output types. DirX Identity currently provides one consumer: DefaultConsumer.

### 9.4.6. Status Report Format Tab

A status report's **Format** tab stores the transformations for the report. Note that XSL transformations are case-sensitive. If in doubt, look at the XML output to see what is upper and what is lowercase.

The following explanations are based on the reports provided with DirX Identity. These reports use include/import mechanisms to better structure the reports, to allow re-using identical parts in different reports and to isolate settings in separate directory objects. Note that a special type of setting (the producer-related settings) has - for technical reasons - been isolated into its own file that is not included here but is imported (using **include** rather than **import** would work, too) within the attribute dxrConfig; for details see the "Producer Element" section).

Start with the last **include** element in the format definition shown in the following figure. It is an XSL transformation that is based on the data the plain XML report reveals and calls miscellaneous templates that must be included in one of the include elements shown in the figure.

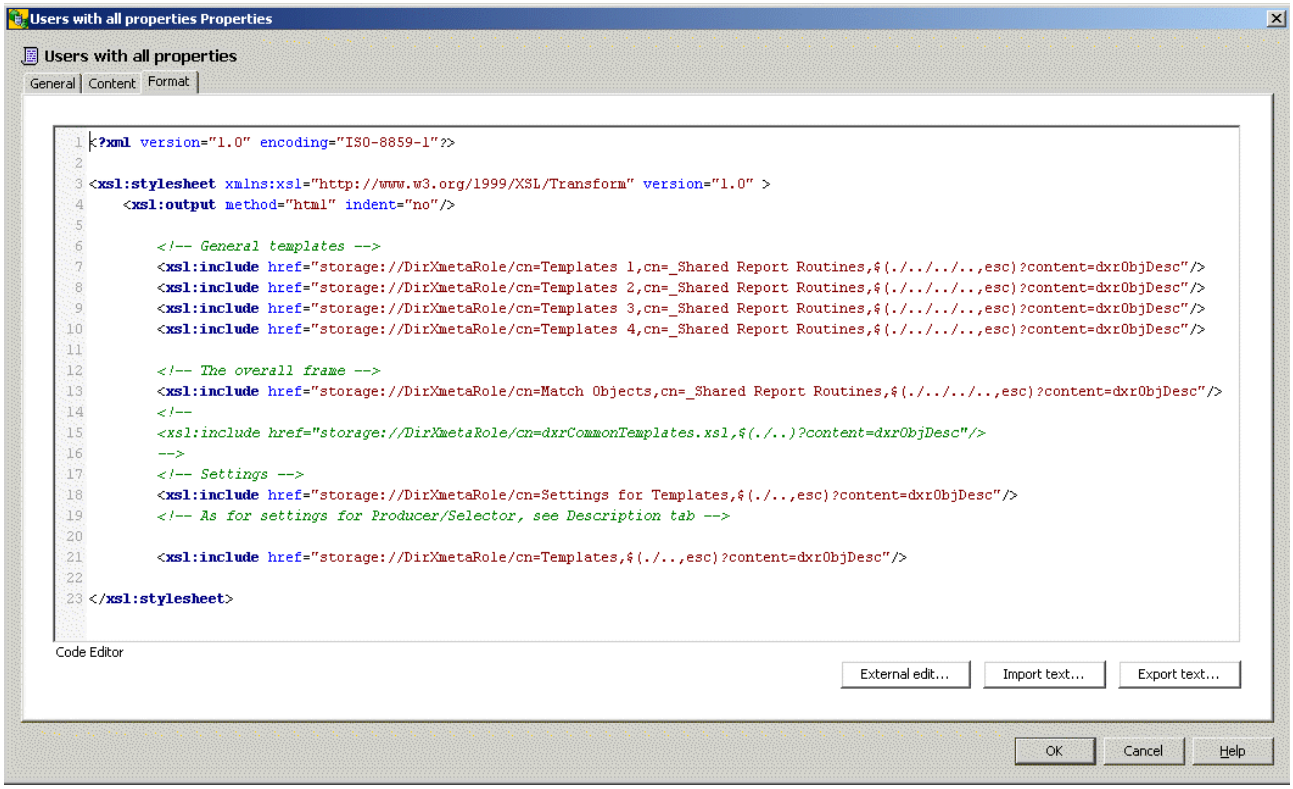



*Figure 10. Example Format Definition*

The following figure shows an excerpt of an XSL template definition for a user report. It singles out all objects with their properties whose type is **dxrUser** and calls a template ("EntryTitleNCols") that creates a title for each user object. The template that follows

("RowHeaderNCols") creates another title for the first block of data ("General"). The last template shown in this figure outputs the user's common name and first name. All these templates must be found in one of the objects included in the "Example Format Definition" figure.

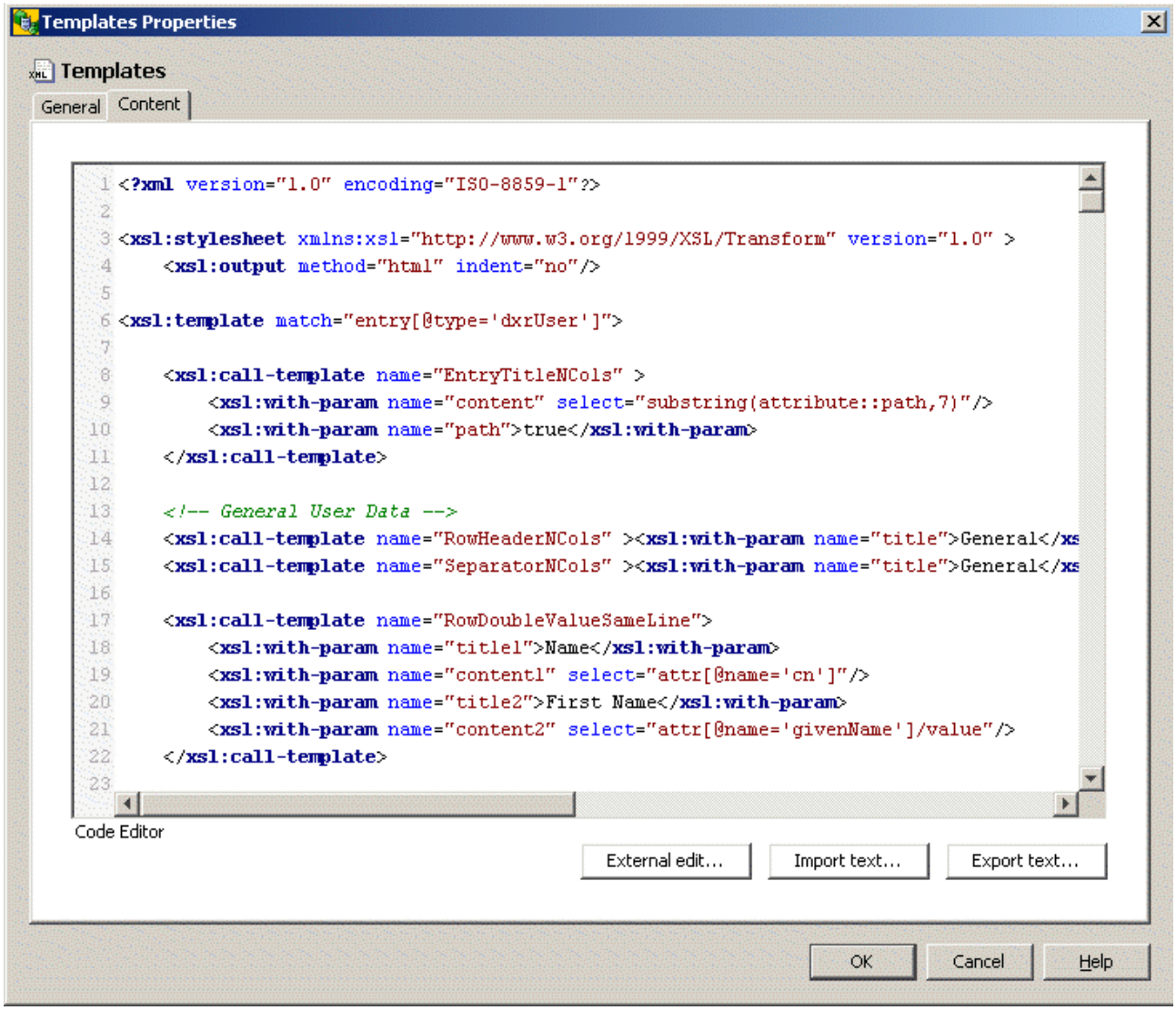



*Figure 11. Example User Report XSL Template Definition*

### 9.4.7. Status Report Output Formats

The output format definition in a status report object description includes the transformer- and the consumer-related settings:

```
<output format, for example, HTML enable="true">
<transformer> ... </transformer>
<FILE><consumer>...</consumer></FILE>
</same output format as above>
```

Although DirX Identity does not restrict the types of output formats, HTML and TXT are the

most common formats. XML is a type of transparent auxiliary format that helps to build the really useful output formats. There is another format called "RAW". By default, it does pretty much the same as the output format XML. It can however be customized, since it is implemented based on XSL transformations (found in the xslFile property) rather than a Java class.

#### 9.4.7.1. XML Output Format

XML format generally corresponds to the output of the selector and therefore serves as a source of information for the XSL transformations carried out for other formats. Here is an example of an XML format:

```
<XML enable="true">
  <transformer class="siemens.dxm.storage.report.XMLWriter"/>
  <FILE>
    <consumer class="siemens.dxm.storage.report.DefaultConsumer">
      <property name="file" value="RoleXMLReport.xml"/>
      <property name="contenttype" value="text/plain"/>
      <property name="includeTimestamp" value="true"/>
      <property name="timestampFormat" value="yyMMddHHmmss"/>
    </consumer>
  </FILE>
</XML>
```

With regard to the preceding example:

- Setting **enable** to "false" prevents the format from being selectable when running a report. The **enable** keyword can be omitted, its default value is "true".
- **RoleXMLReport.xml** is the default file name. You can choose a different one when running a report. You can also have the XML output be displayed in a viewer instead of storing it in a file.
- If enabled, the property **includeTimestamp** adds a timestamp in the default format **yyMMddHHmmss** to the configured file name. The default format can be redefined using property **timestampFormat**. You can use any format that can be correctly used for the common Java class **java.text.SimpleDateFormat**. (See the Java documentation for more details.) Be aware of timestamp formats which can collide with restrictions of the file system you are using. Any configured format using forbidden characters might cause creation of an invalid file name for the selected report object.

#### 9.4.7.2. HTML Output Format

HTML is the preferred format of the reports provided with DirX Identity. Here is an example of an HTML format:

```
<HTML enable="true">
```

```
  <transformer class="siemens.dxm.storage.report.XSLTWriter">
    <property name="xslFile"
value="storage://DirXmetaRole/$(.)?content=dxrFormat" />
    <property name="segmentSize" value="100" />
  </transformer>
  <FILE>
    <consumer class="siemens.dxm.storage.report.DefaultConsumer">
      <property name="file" value="RoleHTMLReport.html"/>
      <property name="contenttype" value="text/html" />
      <property name="includeTimestamp" value="true"/>
      <property name="timestampFormat" value="yyMMddHHmmss"/>
    </consumer>
  </FILE>
</HTML>
```

Regarding the preceding example:

- Setting **enable** to "false" prevents the format from being selectable when running a report. The **enable** keyword can be omitted; its default value is "true".
- **storage://DirXmetaRole/$(.)?content=dxrFormat** specifies that the XSL transformation is to be found in the attribute dxrFormat.
- **RoleHTMLReport.xml** is the default file name. You can choose a different one when running a report. You can also have the HTML output be displayed in a viewer instead of storing it in a file (for test purposes).
- You can set a limit on the size of the output file with the **segmentSize** keyword. The clause: <property name="segmentSize" value="**100**"/> shown in the example specifies that an output file must not contain more than 100 objects that match the first entry clause of the selector. The first 100 objects (including their relationships, if specified) will be stored in the file name specified in the consumer section. The segments that follow are stored in a file of the same name with an additional ascending suffix that starts with "1"; for example, *.1.html instead of *.html, and so on. For HTML files, you are supposed to be offered to navigate between the files within your browser.
- Additionally, you can use the properties **includeTimestamp** and **timestampFormat** to add timestamps to the report file names automatically. See the section "XML Output Format" for details.

#### 9.4.7.3. Other Output Formats

Some DirX Identity reports use TXT or CSV format. Its definition differs from the HTML format definition only by the initial clause "<TXT>" ... "</TXT>" or "<CSV>" ... "<CSV>" which replaces "<HTML>" ... "</HTML>".

## 9.4.8. Customizing Provisioning Status Reports

This chapter provides information specific to customizing provisioning status reports, including:

- Status report locations
- Virtual objects and attributes
- Graphical reports

### 9.4.8.1. Locating Provisioning Reports

DirX Identity Manager's **Auditing** view provides a virtual view of the collection of Provisioning status reports that exist in various locations in the Provisioning configuration. In this view, the **Status Reports** tree contains two sub-trees: **Default** and **Customer Specific**. The **Default** sub-tree contains the default Provisioning status reports, while the **Customer Specific** sub-tree is intended to store status reports that you, the customer, create. Do not change a default report within the Default sub-tree! New releases of DirX Identity will overwrite your changes. Instead, copy a default report to the **Customer Specific** sub-tree and modify it there. You can also right-click in this tree and create your own reports from scratch.

The Customer-Specific tree contains the sub-tree **Target Systems Instances**. This tree is a virtual view into the target system-specific status reports contained in **Target Systems** → *target_system* → **Configuration** → **Reports**, where *target_system* represents a particular system, for example, **SAP R3** or **Intranet Portal**. You can work with target systems status reports in the virtual view or work directly in the target system's **Configuration** → **Reports** view.

### 9.4.8.2. Virtual Objects and Attributes

You can use the whole set of attributes that are defined for an object via the related object description. This includes all custom attributes defined in your specific project. Additionally you can use a set of virtual attributes available for convenience. These attributes represent properties that do not exist directly in the underlying LDAP repository. Their calculation might require access to a set of LDAP attributes and deep knowledge of the calculation algorithm. Such attributes are displayed in the DirX Identity Manager or DirX Identity Web Center. Using them in reports guarantees consistent display. Changes and extensions in the calculation algorithms do not require report adaption.

The following section lists and explains the virtual attributes of the most important objects.

**User** (dxrUser) - represents the basic user object.

**Roles.Assigned** - contains all roles assigned to a user.

The assignment attributes for a role assignment are:

assign.dxrStartDate - the start date of the assignment.

assign.dxrEndDate - the end date of the assignment.

assign.dxrNeedsReapproval - the reapproval flag

assign.dxrRoleParamValue - the role parameters

assign.mode - mode of assignment (manual, rule, ...)

assign.displayState - state of the assignment

**UserPermissions.Assigned** - contains all permissions assigned to a user.

The assignment attributes for a permission assignment are:

assign.dxrStartDate - the start date of the assignment.

assign.dxrEndDate - the end date of the assignment.

assign.dxrNeedsReapproval - the reapproval flag

assign.mode - mode of assignment (manual, rule, ...)

assign.displayState - state of the assignment

**Groups.Assigned** - contains all groups assigned to a user.

The assignment attributes for a group assignment are:

assign.dxrStartDate - the start date of the assignment.

assign.dxrEndDate - the end date of the assignment.

assign.dxrNeedsReapproval - the reapproval flag

assign.mode - mode of assignment (manual, rule, ...)

assign.displayState - state of the assignment

**Accounts.Assigned** - contains all accounts assigned to a user.

assign.state - state of the assignment

**subjectmodifyorders** - list of orders for object modification with these properties:

duedate

attribute

oldvalues

newvalues

**resourceorders** - list of orders for resources with these properties:

$mergegroup - a pseudo property that defines the rows that could be merged in a table

operation

duedate

resourcetype

privilege

attribute

oldvalues

newvalues

mode

**sodexceptions** - all granted SoD exceptions for this user with these properties:

- policy.cn
- dxracceptedsodviolationslink
- activityid
- reason
- who
- when

**sodviolations** - all pending SoD violations with these properties:

- policyname
- privilege

**allactivedelegations.assigned** - all active delegation assignments for the user with these properties:

- dxrName
- description
- dxrAssignFrom@cn
- dxrAssignFrom@ou
- dxrStartDate
- dxrEndDate
- dxrPreventDelegation

**gotdelegations.assigned** - all delegations for a user (active and inactive ones) with these properties:

- dxrName
- description
- dxrAssignFrom@cn
- dxrAssignFrom@ou
- dxrStartDate
- dxrEndDate
- dxrPreventDelegation

**granteddelegations.assigned** - all delegations this user has delegated to other users with these properties:

- dxrName
- description
- dxrAssignFrom@cn
- dxrAssignFrom@ou
- dxrStartDate
- dxrEndDate

dxrPreventDelegation

**delegateabledelegations.assigned** - all delegations a user could delegate to other users with these properties:

dxrName

description

dxrAssignFrom@cn

dxrAssignFrom@ou

dxrStartDate

dxrEndDate

**Roles** (dxrRole) - represents the role object.

**Roles.Assigned** - contains all assigned junior roles (lower hierarchy level).

**Permissions.Assigned** - contains all assigned permissions.

**seniorroles** - contains all assigned senior roles (higher hierarchy level).

**affectedUsers** - contains all assigned users. Note: this list can be very long for specific roles. Display of this property can result in poor performance or memory problems.

**Permissions** (dxrPermission) - represents the permission object.

**Matchrules** - the match rules with these properties:

AndOr

UserAttribute

Operator

Object

Value

**Groups.Assigned** - contains all assigned groups.

**Group** (dxrTargetSystemGroup) - represents the target system group object.

**Permissions** - all permissions that use this group.

**memberAccounts** - all accounts that are assigned to this group.

**remoteMembers** - members of this group that do not have an account.

**Groups.memberOf** - all groups this group is a member of.

**Account** (dxrTargetSystemAccount) - represents the target system account object.

**Groups.Assigned** - groups assigned to an account.

#### 9.4.8.3. About Graphical Status Reports

In DirX Identity's Provisioning view, users, roles, permissions, groups, target systems etc. are connected by various relationships. DirX Identity Manager is currently unable to represent

these relationships in a graphical way. However, the report mechanism of both Manager and the DirX Identity report agent can be used to generate output format that can be interpreted by visualization tools.

DirX Identity provides a set of standard reports that generate output in GVF (Graph Visualization Framework) format. A trailing _GVF in a DirX Identity report name indicates that it generates output in GVF format.

For more information on the GVF visualization tool, see

http://gvf.sourceforge.net/

and

http://www.cwi.nl/InfoVisu/

## 9.4.9. Customizing Connectivity Status Reports

This section provides information specific to customizing connectivity status reports, including information about:

- Locating the connectivity status reports
- Types of connectivity status reports
- XSL template definitions for the connectivity status reports

### 9.4.9.1. Locating Connectivity Reports

The pre-configured Connectivity "template" reports provided with DirX Identity cover the important Connectivity objects and their relationships: scenarios, schedules, workflows, jobs, channels and connected directories - including sub-elements, for example, Tcl scripts and INI files. The reports produce a complete list of these objects and their direct assignments in the domain together with the set of attributes describing them.

currently the default reports do not cover samples for Java-based workflows.

Using DirX Identity Manager's **Connectivity** view, you can find the Connectivity reports in **Status Reports** → **Default**. You can also find them in the **Expert View** under **Configuration** → **GUI** → **Reports** → **Default**. The following figure illustrates the relationships between the Connectivity report objects in the Default subdirectory.

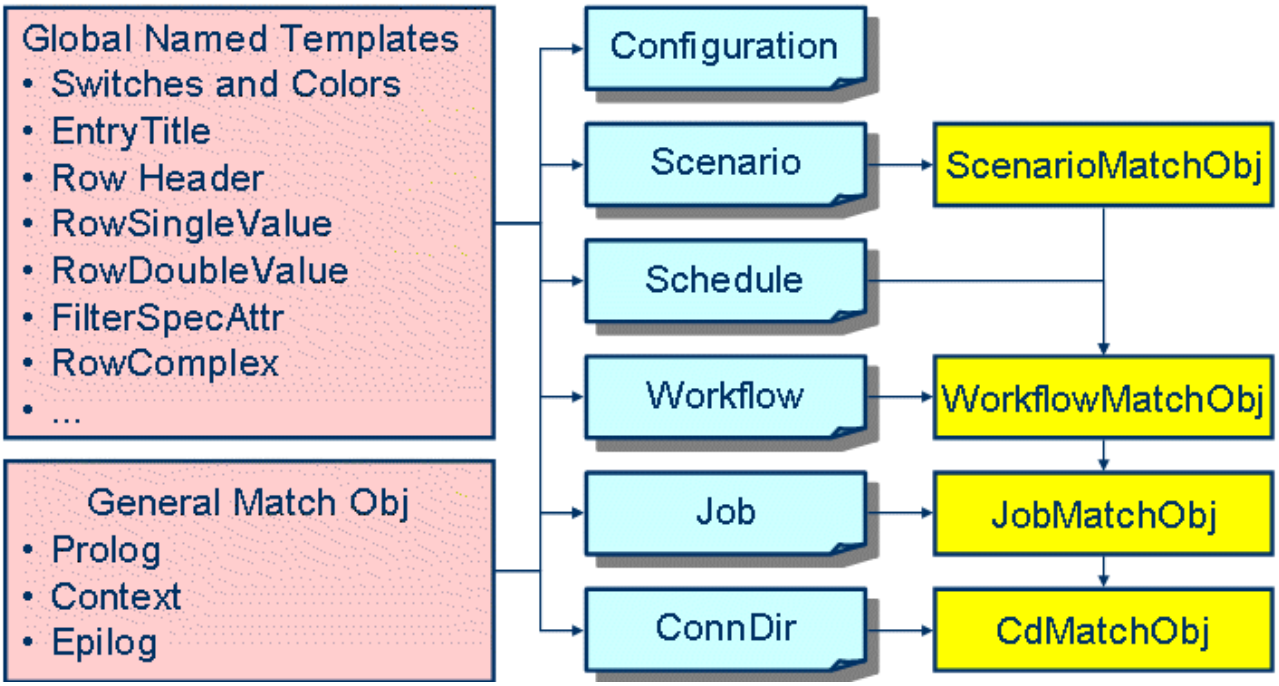



*Figure 12. Default Connectivity Report Structure*

The **Common** folder contains the XSLT template definitions required by the default DirX Identity Connectivity reports. Any report can use the **GlobalNamedTemplates** and the **GeneralMatchObj** XSLT routines.

#### 9.4.9.2. Types of Connectivity Reports

There are two basic types of Connectivity reports. The **Generic** report creates information about almost any object in the configuration database in a generic table format. The other reports produce the requested information in format that is similar to the object representation at the user interface level (DirX Identity Manager and Web Center).

- **Configuration** - you can use this report either at the Connectivity Configuration Data root object or at Configuration. It produces:
  - The root object itself
  - The (central) configuration object
  - All C++based Server objects
  - All Service objects
  - All System objects
- **Scenario** - available only at the Scenario folder or objects. It produces:
  - All scenario objects contained in the folder (if a folder was selected) or the selected scenario object
  - The contained workflows and activities
  - The related jobs and sub-objects (configuration files, notification objects)
  - The contained connected directories, sub-objects and channels
- **Schedule** - use this report either on Schedule folders or objects. It produces:

- All schedule objects contained in the folder (if a folder was selected) or the schedule object
- The contained workflows and activities
- The related jobs and sub-objects (configuration files, notification objects)
- The contained connected directories, sub-objects and channels

- **Workflows** - available only on Workflow folders or objects. It produces:
  - All workflow objects contained in the folder (if a folder was selected) or the workflow and its activities
  - The related jobs and sub-objects (configuration files, notification objects)
  - The contained connected directories, sub-objects and channels
- **Jobs** - use this report on Job folders or objects. It produces:
  - All job objects contained in the folder (if a folder was selected) or the related jobs and sub-objects (configuration files, notification objects)
  - The contained connected directories, sub-objects and channels
- **ConnDirs** - use this report on Connected Directory folders or objects. It produces:
  - All connected directory objects contained in the folder (if a folder was selected) or the selected connected directory object

This report currently does not contain channels.

Reports can be very resource-consuming. Do not try to produce too much data at one time. If you run into problems, try to produce several sub-reports instead of one big one. You can also try to reduce the level of detail.

#### 9.4.9.3. XSL Template Definitions

The XSLT definitions for the DirX Identity Connectivity reports allow you to control display properties and the level of detail for all reports except for the generic report.

**GeneralMatchObj.xslt**, **CdMatchObj.xslt**, **JobMatchObj.xslt**, **WorkflowMatchObj.xslt** and **ScenarioMatchObj.xslt** specify the definition of the related objects in a hierarchical way

**GlobalNamedTemplates.xslt** contains the variables and constants that control display and the level of detail as well as common routines that are used to produce the common representation of the objects. This routine controls all reports except for the generic report.

To control the level of detail, use:

**DisplayEmpty** - defines whether attributes with empty values shall be displayed. Default is false.

**ConfigFileDisplay** - controls whether or not configuration file objects are processed. Default is true.

**NotifyDisplay** - defines whether or not Notify objects are displayed. Default is true.

**IniTextDisplay** - controls INI file text content display. Default is false.

**TclTextDisplay** - displays Tcl file content. Default is false.

**MapTableDisplay** - defines whether or not map tables are displayed. Default is true.

**SelectedAttributesDisplay** - controls the display of selected attributes tables. Default is true.

**AttributeConfigDisplay** - displays attribute configuration tables. Default is false.

To control display behavior, use:

**textstyle** - defines the style of the output text.

**objcolor** - controls the color of object headers.

**groupcolor** - defines the color of group separators.

**tblheadcolor** - sets the color of a table heading.

**tablecolor** - defines the color of the table body.

# 10. Customizing Program Logic

The topics in this section describe how to:

- Use JavaScript files
- Write Java extensions
- Customize the creation workflows for persona and functional users
- Customize the consistency rules for automatic persona, functional user and user creation
- Customize the JavaScript that exchanges personas and users

## 10.1. Using JavaScript Files

DirX Identity provides another powerful technique for customer extensions: **JavaScript** files.JavaScript files contain **JavaScript** code, a programming language that is slightly different from **Java** and a little bit easier to use.JavaScript files are used in DirX Identity to run small customized programs in conjunction with particular DirX Identity operations.The JavaScript programs are stored in separate script entries that can be referenced in object attributes and properties.

You can currently use JavaScript files to:

- Create default values for object attributes
- Make post-editing consistency checks of object attributes
- Extend object operations like save, remove, and so on

Before you can use JavaScript files to extend DirX Identity, you need to know how to:

- Use the JavaScript programming language and its pre-defined objects
- Create a JavaScript file in the DirX Identity system
- Integrate your JavaScript files into the DirX Identity system.

The next sections describe these tasks.

### 10.1.1. JavaScript Programming Overview

This section provides an overview of the JavaScript programming language, including:

- Variables, values, and literals
- Comments
- Keywords
- Operators
- Functions
- Objects, methods, and properties

- Predefined objects

This section is not intended for use as a JavaScript reference. For detailed information about JavaScript, refer to the various guides and references available in textbook form and on the Internet.

#### 10.1.1.1. JavaScript Variables, Values, and Literals

You can declare variables in JavaScript directly by writing the variable name and assigning it a value. A variable can also remain unassigned. In this case, use the keyword **var** to avoid a runtime error if the variable is never assigned a value referenced in some statement.

JavaScript recognizes the following types of values:

- Numbers, such as 42 or 3.14159.
- Logical (Boolean) values, either **true** or **false**.
- Strings, such as "Howdy!".
- **null**, which is a special keyword that indicates a null value; null is also a primitive value. Because JavaScript is case-sensitive, null is not the same as Null, NULL, or any other variant.
- **undefined**, which is a top-level property whose value is undefined; undefined is also a primitive value.

Since JavaScript is a dynamically typed language, the type of assignment may vary during runtime. A variable can initially take a number value, and then take a string value later in the program. When a variable identifier is set by assignment outside a function, it is called a *global* variable, because it is available everywhere in the current document. When a variable is declared within a function, it is called a *local* variable, because it is available only within the function. Using the keyword **var** to declare a global variable is optional. However, **var** is must be used to declare a variable inside a function.

Literals are used to represent values. The following types of literals are valid:

- Array literals [...,...,...] such as [1, 4, 9, 16, 25] or ["green", "yellow", "red"].
- Boolean literals, either **true** or **false**.
- Floating-point literals (signed or unsigned), for example 3.1415 or 1.523e-15.
- Integers (signed or unsigned), such as -25 or 0x2BFF.
- Object literals, which are sets of comma-separated properties and assigned values enclosed in curly braces; for example, flower = \{name: "Rose", height: 65, price: 2.25}
- String literals enclosed in single quotes ( ' ) or double quotes ( "); for example, 'blah', "blah", or "one line \n another line".

The following table lists the special characters that can be used in string literals:

*Table 6. Special Characters in String Literals*

| Character | Meaning |
|---|---|
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Tab |
| \' | Apostrophe or single quote |
| \" | Double quote |
| \\ | Backslash character (\) |
| \n*nn* | The character with the Latin-1 encoding specified by up to three octal digits *nnn* between 0 and 377. For example, \251 is the octal sequence for the copyright symbol. |
| \x*nn* | The character with the Latin-1 encoding specified by the two hexadecimal digits *nn* between 00 and FF. For example, \xA9 is the hexadecimal sequence for the copyright symbol. |
| **\u***nnnn* | The Unicode character specified by four hexadecimal digits *nnnn*. For example, \u00A9 is the Unicode sequence for the copyright symbol. |

#### 10.1.1.2. JavaScript Comments

Like the C or C++ programming languages, a single-line comment begins with “//” at any position in a line and terminates with the subsequent line-feed character.

A multi-line comment starts with “/*” at any position in a line and ends with “*/” at any position in a subsequent line or behind the start sequence on the same line.

#### 10.1.1.3. JavaScript Keywords

The following table lists the most frequently used keywords and their meanings.

*Table 7. JavaScript Keywords*

| Keyword | Description |
|---|---|
| break | Terminates the current while or for loop and transfers program control to the statement following the terminated loop. |
| continue | Terminates execution of the block of statements in a while or for loop, and continues execution of the loop with the next iteration. |
| delete | Deletes an object, an object’s property, or an element at a specified index in an array. |
| do...while | Executes the specified statements until the test condition evaluates to false. Statements execute at least once. |
| export | Allows a signed script to provide properties, functions, and objects to other signed or unsigned scripts. |

| Keyword | Description |
|---|---|
| false | Boolean value literal. |
| for | Creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a block of statements executed in the loop. |
| for...in | Iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements. |
| function | Declares a function with the specified parameters. Acceptable parameters include strings, numbers, and objects. |
| if...else | Executes a set of statements if a specified condition is true. If the condition is false, another set of statements can be executed. |
| import | Allows a script to import properties, functions, and objects from a signed script that has exported the information. |
| label | Provides an identifier that can be used with break or continue to indicate where the program should continue execution. |
| new | Creates an instance of a user-defined object type or of one of the built-in object types. |
| null | The null-value for objects. |
| return | Specifies the value to be returned by a function. |
| switch | Allows a program to evaluate an expression and attempt to match the expression’s value to a case label. |
| this | Keyword that you can use to refer to the current object. |
| true | Boolean value literal. |
| typeof | Returns a string indicating the type of the unevaluated operand. |
| undefined | The null-value for variables (especially unassigned variables declared by "var". |
| var | Declares a variable, optionally initializing it to a value. |
| void | Specifies an expression to be evaluated without returning a value. |
| while | Creates a loop that evaluates an expression, and if it is true, executes a block of statements. The loop then repeats, as long as the specified condition is true. |
| with | Establishes the default object for a set of statements. |

#### 10.1.1.4. JavaScript Operators

The following table lists the operators used in JavaScript expressions and statements.

*Table 8. JavaScript Operators*

| Operator | Description |
|---|---|
| + (Addition) | Adds 2 numbers. |

| Operator | Description |
| --- | --- |
| ++ (Increment) | Adds 1 to a variable representing a number (returning either the new or old value of the variable) |
| - (Unary negation, subtraction) | As a unary operator, negates the value of its argument. As a binary operator, subtracts 2 numbers. |
| — (Decrement) | Subtracts 1 from a variable representing a number (returning either the new or old value of the variable) |
| * (Multiplication) | Multiplies 2 numbers. |
| / (Division) | Divides 2 numbers. |
| % (Modulus) | Computes the integer remainder of dividing 2 numbers. |
| + (String addition) | Concatenates 2 strings. |
| += | Concatenates 2 strings and assigns the result to the first operand. |
| && (Logical AND) | Returns the first operand if it can be converted to false; otherwise, returns the second operand. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false. |
| \|\| (Logical OR) | Returns the first operand if it can be converted to true; otherwise, returns the second operand. Thus, when used with Boolean values, \|\| returns true if either operand is true; if both are false, returns false. |
| ! (Logical NOT) | Returns false if its single operand can be converted to true; otherwise, returns true. |
| & (Bitwise AND) | Returns a one in each bit position if bits of both operands are ones. |
| ^ (Bitwise XOR) | Returns a one in a bit position if bits of one but not both operands are one. |
| \| (Bitwise OR) | Returns a one in a bit if bits of either operand is one. |
| ~ (Bitwise NOT) | Flips the bits of its operand. |
| << (Left shift) | Shifts its first operand in binary representation the number of bits to the left specified in the second operand, shifting in zeros from the right. |
| >> (Sign-propagating right shift) | Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off. |
| >>> (Zero-fill right shift) | Shifts the first operand in binary representation the number of bits to the right specified in the second operand, discarding bits shifted off, and shifting in zeros from the left. |
| = | Assigns the value of the second operand to the first operand. |
| += | Adds 2 numbers and assigns the result to the first. |
| -= | Subtracts 2 numbers and assigns the result to the first. |

| Operator | Description |
|---|---|
| *= | Multiplies 2 numbers and assigns the result to the first. |
| /= | Divides 2 numbers and assigns the result to the first. |
| %= | Computes the modulus of 2 numbers and assigns the result to the first. |
| &= | Performs a bitwise AND and assigns the result to the first operand. |
| ^= | Performs a bitwise XOR and assigns the result to the first operand. |
| \|= | Performs a bitwise OR and assigns the result to the first operand. |
| <⇐ | Performs a left shift and assigns the result to the first operand. |
| >>= | Performs a sign-propagating right shift and assigns the result to the first operand. |
| >>>= | Performs a zero-fill right shift and assigns the result to the first operand. |
| == | Returns true if the operands are equal. |
| != | Returns true if the operands are not equal. |
| === | Returns true if the operands are equal and of the same type. |
| !== | Returns true if the operands are not equal and/or not of the same type. |
| > | Returns true if the left operand is greater than the right operand. |
| >= | Returns true if the left operand is greater than or equal to the right operand. |
| < | Returns true if the left operand is less than the right operand. |
| ⇐ | Returns true if the left operand is less than or equal to the right operand. |
| ...?...:..... | Performs a simple "if...else" |
| , | Evaluates two expressions and returns the result of the second expression. |
| delete | Deletes an object, an object's property, or an element at a specified index in an array. |
| new | Creates an instance of a user-defined object type or of one of the built-in object types. |
| this | Keyword that you can use to refer to the current object. |
| typeof | Returns a string indicating the type of the unevaluated operand. |
| void | Specifies an expression to be evaluated without returning a value. |

#### 10.1.1.5. JavaScript Functions

A function definition consists of the **function** keyword, followed by

- The name of the function.
- A list of arguments to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly braces, \{ }. The statements in a function can include calls to other functions defined in the current application.

Defining a function does not execute it. Defining the function simply names the function and specifies what to do when the function is called. Calling the function actually performs the specified actions with the indicated parameters.

JavaScript has several top-level predefined functions:

- **eval** - Evaluates a string of JavaScript code without reference to a particular object, for example eval(expr).
- **isFinite** - Evaluates an argument to determine whether it is a finite number, for example isFinite(number).
- **isNaN** - Evaluates an argument to determine if it is "NaN" (not a number), for example isNaN(testValue).
- **parseInt** and **parseFloat** - Return a numeric value when given a string as an argument, like parseInt(numberString) or parseFloat(numberString).
- **Number** and **String** - Convert an object to a number or a string, like Number(object) or String(object).
- **escape** and **unescape** - Encode and decode strings. The escape function returns the hexadecimal encoding of an argument in the ISO Latin character set. The unescape function returns the ASCII string for the specified hexadecimal encoding value.

#### 10.1.1.6. JavaScript Objects, Methods and Properties

JavaScript is based on a simple object-based paradigm. An object is a construct with properties that are JavaScript variables or other objects. An object also has functions associated with it that are known as the object's methods. JavaScript contains a number of predefined objects and new objects can also be defined.

A JavaScript object has properties associated with it. The properties of an object are accessed with a simple notation:

**objectName.propertyName**

Both the object name and property name are case-sensitive. A property is defined by assigning it a value.

A method is a function associated with an object. A method is defined the same way a standard function is defined. Then the following syntax is used to associate the function with an existing object:

**objectName.methodName = functionName**

where objectName is the name of an existing object, methodName is the name to be assigned to the method, and functionName is the name of the function.

In JavaScript, you create an object by using an object initializer or, alternatively, instantiating it with a constructor function and the **new** operator. The constructor function is assigned the name of the object to be created. An existing object may be deleted by the **delete** operator.

## 10.1.2. Pre-Defined JavaScript Objects

This topic briefly describes the pre-defined JavaScript objects, including:

- Array - An object that represents and manipulates an array of data.
- Boolean - An object that represents a boolean value.
- Date - An object that represents and manipulates a date value.
- Function - An object that represents a function. The **function** keyword can be used instead.
- Math - An object with predefined functions and constants for mathematical calculations.
- Number - An object that represents a numerical value.
- Packages (java) - An object for accessing Java classes
- RegExp - An object that handles expressions.
- String - An object that represents a string and manipulates it.

For more information about these objects, consult a JavaScript language reference.

### 10.1.2.1. Array

JavaScript does not have an explicit array data type. However, the predefined Array object and its methods can be used to work with arrays in applications. The Array object has methods for manipulating arrays in various ways, such as joining, reversing, and sorting them. It has a property for determining the array length and other properties for use with regular expressions.

**Constructors**

arrayObjectName = new Array(element1, ..., elementN)

arrayObjectName = new Array(elementCount)

*arrayObjectName* is the name of the array object to be created. *element1*, ..., *elementN* are the data items added to the array. This is an explicit construction of the array. In the second statement, *elementCount* is just the number of the data items in the array, there is indeed no item assigned yet.

**Methods**

Array contains the following methods:

concat - Joins two arrays and returns a new array.

join - Joins all elements of an array into a string.

pop - Removes the last element from an array and returns that element.

push - Adds one or more elements to the end of an array and returns that last element added.

reverse - Transposes the elements of an array: the first array element becomes the last and the last becomes the first.

shift - Removes the first element from an array and returns that element

slice - Extracts a section of an array and returns a new array.

splice - Adds and/or removes elements from an array.

sort - Sorts the elements of an array.

unshift - Adds one or more elements to the front of an array and returns the new length of the array.

#### 10.1.2.2. Boolean

The Boolean object is a wrapper around the primitive Boolean data type.

**Constructor**

booleanObjectName = new Boolean(value)

Do not confuse the primitive Boolean values **true** and **false** with the true and false values of the Boolean object. Any object whose value is not **undefined** or **null**, including a Boolean object whose value is false, evaluates to true when passed to a conditional statement.

#### 10.1.2.3. Date

JavaScript does not have a date data type. However, the Date object and its methods can be used to work with dates and times in applications. The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

**Constructor**

dateObjectName = new Date([parameters])

where *dateObjectName* is the name of the Date object being created; it can be a new object or a property of an existing object. The *parameters* in the preceding syntax can be any of the following:

- Nothing: creates today's date and time. For example, today = new Date().
- A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, Xmas95 = new Date("December 25, 1995 13:30:00"). If you omit hours, minutes, or seconds, the value will be set to zero.

- A set of integer values for year, month, and day. For example, Xmas95 = new Date(1995,11,25). A set of values for year, month, day, hour, minute, and seconds. For example, Xmas95 = new Date(1995,11,25,9,30,0).

**Methods**

The Date object methods for handling dates and times fall into these broad categories:

"set" methods, for setting date and time values in Date objects.

"get" methods, for getting date and time values from Date objects.

"to" methods, for returning string values from Date objects.

parse and UTC methods, for parsing Date strings.

#### 10.1.2.4. Function Object

The predefined Function object specifies a string of JavaScript code to be compiled as a function.

**Constructor**

functionObjectName = new Function ([arg1, arg2, ... argN], functionBody)

*functionObjectName* is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as window.onerror. *arg1, arg2, ... argN* are arguments to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example "x" or "theForm". *functionBody* is a string specifying the JavaScript code to be compiled as the function body.

Function objects are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

#### 10.1.2.5. Math

The predefined Math object has properties and methods for mathematical constants and functions. For example, the Math object's PI property has the value of pi (3.141...), which would be used in an application as

Math.PI

Similarly, standard mathematical functions are methods of Math. These include trigonometric, logarithmic, exponential, and other functions. Note that all trigonometric methods of Math take arguments in radians.

**Constructor**

None. Math is predefined and has static methods which can be directly called as

Math.methodName

where *methodName* is any of the methods listed below in the next section.

#### Methods

Math contains the following methods:

abs - Absolute value of the argument.

sin, cos, tan - Standard trigonometric functions; argument in radians.

acos, asin, atan - Inverse trigonometric functions; return values in radians.

exp, log - Exponential and natural logarithm, base e.

ceil - Returns least integer greater than or equal to argument.

floor - Returns greatest integer less than or equal to argument.

min, max - Returns greater or lesser (respectively) of two arguments.

pow - Exponential; first argument is base, second is exponent.

round - Rounds argument to nearest integer.

sqrt - Square root.

#### 10.1.2.6. Number

The Number object has properties for numerical constants, such as maximum value, not-a-number, and infinity. The values of these properties cannot be changed.

#### Constructor

None. Number is predefined and has static properties which can be directly called as

Number.propertyName

where propertyName is the name of the respective property to be used. The valid properties are listed below.

#### Properties

Number contains the following properties:

MAX_VALUE - The largest representable number.

MIN_VALUE - The smallest representable number.

NaN - Special "not a number" value.

NEGATIVE_INFINITY - Special infinite value; returned on overflow.

POSITIVE_INFINITY - Special negative infinite value; returned on overflow.

#### 10.1.2.7. Packages (java)

This object allows the access of Java classes by JavaScript code

#### Constructor

None. Packages and java are predefined and provide static access to Java packages.

**Properties**

The Packages object provides the following properties:

- className - The fully qualified name of a Java class in a package other than netscape, java, or sun that is available to JavaScript.
- java - Any class in the Java package java.*. Instead of writing Packages.java the synonym java can be used.
- netscape - Any class in the Java package netscape.*.
- sun - Any class in the Java package sun.*.

#### 10.1.2.8. RegExp

The RegExp object lets you work with regular expressions.

**Constructor**

None. RegExp is predefined and has static methods which can be directly called as

RegExp.methodName

where *methodName* is the name of the method to be used for the manipulation of a regular expression.

**Methods**

RegExp contains the following methods:

- exec - Executes a search for a match in a string. It returns an array of information.
- test - Tests for a match in a string. It returns **true** or **false**.

#### 10.1.2.9. String

The String object is a wrapper around the string primitive data type. Do not confuse a string literal with the String object. You can call any of the methods of the String object on a string literal value - JavaScript automatically converts the string literal to a temporary String object, calls the method, then discards the temporary String object. You can also use the String.length property with a string literal.

**Constructor**

stringObjectName = new String(stringLiteral)

where *stringObjectName* is the name of the new string object and *stringLiteral* is the value of the string object.

**Methods**

The methods of the String object are the following:

- anchor - Creates HTML named anchor.

big, blink, bold,fixed, italics, small,strike, sub, sup - Creates HTML formatted string

charAt, charCodeAt - Returns the character or character code at the specified position in string.

indexOf, lastIndexOf - Returns the position of specified substring in the string or last position of specified substring, respectively.

link - Creates HTML hyperlink.

concat - Combines the text of two strings and returns a new string.

fromCharCode - Constructs a string from the specified sequence of ISO-Latin-1 codeset values.

split - Splits a String object into an array of strings by separating the string into substrings.

slice - Extracts a section of an string and returns a new string.

substring, substr - Returns the specified subset of the string, either by specifying the start and end indexes or the start index and a length.

match, replace, search - Used to work with regular expressions.

toLowerCase, toUpperCase - Returns the string in all lowercase or all uppercase, respectively.

## 10.1.3. Creating JavaScript Entries

JavaScript entries are contained in JavaScript folders located in a target system configuration or in the DirX Identity system configuration. To add a new JavaScript to DirX Identity, right-click any JavaScript folder and then select **New JavaScript**. You can then compose your JavaScript statements in the JavaScript code editor window.

## 10.1.4. Integrating JavaScript Entries

JavaScript entries can be easily integrated into a DirX Identity configuration. The different purposes require different techniques.

#### 10.1.4.1. Constructing Object Property Values

JavaScript programs can be used to construct default values for an object's properties. To integrate this type of a JavaScript entry, you extend the XML object description to reference the JavaScript entry from the object's property definition, as follows:

```
<property name=... type =... defaultvalue="$(script:entryName)">
<script name="entryName" return="returnValue"

reference="storage://DirXmetaRole/cn=entryName,cn=JavaScripts,$(./../
..)?content=dxrObjDesc"/>
</property>
```

where *entryName* is the name of the JavaScript entry and *returnValue* is the constructed value to be used as the default value. The notation "$(script.entryName)" is a placeholder for the returned value of the JavaScript program. The variable notation $(./../..) in the **reference** property indicates a relative path starting from the XML object description that contains the property definition code. In the example shown here, the variable notation indicates a node two levels above the node of the object description (which can be a target system node or the domain configuration node).

#### 10.1.4.2. Creating Consistency Checks for Property Values

JavaScript programs can also be used to check the consistency of a property value, usually during save operations. To add a consistency check, you add another reference within the property definition in the object description, as follows:

```
<property name=... type =... >
<script name="verifyContent"

reference="storage://DirXmetaRole/cn=verifyContent,js,cn=JavaScripts,
$(./../..)?content=dxrObjDesc"/>
</property>
```

The verifyContent.js JavaScript entry can, for example, check the length of the property value and display a log message if it is too short or too long.

You do not need to create a separate script to handle consistency checking. Instead, you can write the JavaScript code directly into the **script** tag, as follows:

```
<property name=... type =... >
<script name="verifyContent">
<![CDATA[
   importPackage(java.lang);
   obj=scriptContext.getObject();
   String value=obj.getProperty("someProperty");
   if (value.length() >= 10)
       System.out.println("someProperty is too long");
   else if (value.length < 5)
       System.out.println("someProperty is too short");
]]>
</script>
</property>
```

The location of the **script** tag determines the scope of the JavaScript program's application. When inserted as shown in the previous example, it applies only to the **property** tag that

encloses the script tag. When it is located within the **properties** tag, it applies to all properties of the object.

If the verifyContent script is property-specific (applied inside the property tag), you can get the value of the property via the scriptContext:

```
<property name=... type =... >
<script name="verifyContent">
<![CDATA[
  importPackage(java.lang);
  String value=scriptContext.getPropertyValue();
  if (value.length() >= 10)
      System.out.println("someProperty is too long");
  else if (value.length < 5)
      System.out.println("someProperty is too short");
]]>
</script>
</property>
```

In this case, ScriptContext also provides getPropertyDescriptor(), which returns the StoragePropertyDescriptor of this property.

#### 10.1.4.3. Extending Object Operations

The following JavaScript programs can be used to extend operations on objects:

- **verifyContent** - to be called before a property of an object is set.
- **save** - to be called prior to saving of an object.
- **reset** - to be called prior to resetting an object.
- **refresh** - to be called prior to refreshing an object.
- **remove** - to be called prior to removing an object.
- **commit** - to be called prior to committing (performing) a transaction.
- **rollback** - to be called prior to rolling back (discard) a transaction.

The scope of these scripts is again determined by the location of the script in the XML object description. If the script is located within the **object** tag, it runs on all corresponding operations for this object. If it is located outside any **object** tag located in a **scripts.xml** file entry, it runs on all respective operations for all objects. The file **scripts.xml** can be located as an additional object description in the corresponding folder.

#### 10.1.4.4. Using the ScriptContext Object

A JavaScript program that is to be executed is passed the ScriptContext object as a global variable. This variable provides access to the application context and particularly to the

object that is currently being manipulated. The ScriptContext object is referenced through the variable **scriptContext**. The following methods are relevant for script writers:

**getObject()** - Returns the StorageObject instance which is currently operated on. This object provides a getProperty method to return its properties and a setProperty(Object value) method to set a particular property.

**getTransaction()** - Returns the Transaction object which is currently used for the object manipulation. The Transaction object provides methods commit and rollback for performing or discarding a transaction.

Normally, you should neither commit or rollback a transaction in the script. This should be left to the calling component, which created the transaction and might want to perform other changes. In case of a failure, which prevents storing the changes you should return an exception back to the caller rather than cause the transaction to rollback yourself.

### 10.1.5. Multi-value Attribute Handling

You can set default values of a multi-value attribute of an entry by setting these values in the assignment string, for example:

```
<property name="objectclass" defaultvalue="{dxrAccessPolicy,top}"/>
```

This instruction does not work if the values contain commas or if you want to perform more detailed operations. In this case, you must write a save script for an object. In this script, you can manage multi-value properties to any level of detail; for example:

```
obj.addMissing(propname, value) - add a missing value
obj.delExisting(propname, value) - delete a value
```

These methods work even if the value already exists. The system recognizes this automatically and simply does nothing.

You can call

```
obj.propertyModified(propname)
```

to check if the value of an attribute has changed to determine whether you need to calculate another value.

DirX Identity V8.6 allows you to create an array of your default values in the Javascript. In this case, you do not have the restrictions with containing commas or curly braces.

Here is an example returning an array of storage objects that should be used as default values for the secondary location link in combination with **dependson**:

Object description:

```
<property name="dxrSecLocationLink"
          type="siemens.dxm.storage.StorageObject"
          dependson="attname"
          label="More Locations"
          multivalue="true"
          defaultvalue="$(script:createDefLocs)"

editorparams="choosefilter=dxrLocation;chooserootdn=cn=Countries,cn=B
usinessObjects,$(rootDN)"
          editor="siemens.dxm.storage.beans.JnbReference"

multivalueeditor="siemens.dxr.manager.controls.MetaRoleJnbMultiValue"
>
    <script name="createDefLocs"
            return="locs"

reference="storage://DirXmetaRole/cn=defloc.js,cn=JavaScripts,cn=Conf
iguration,$(rootDN)?content=dxrObjDesc"/>
</property>
```

Java script coding:

```
importPackage(java.lang);
importPackage(java.lang.reflect);
importPackage(java.util);
importPackage(Packages.siemens.dxm.storage);

var obj = scriptContext.getObject();
var sess = obj.getStorage();
// 2 fixed values
var l1 = sess.getObject("o=Mercato
Aurum,cn=Companies,cn=BusinessObjects,cn=My-Company");
var l2 = sess.getObject(
"o=MultiMarket,cn=Companies,cn=BusinessObjects,cn=My-Company");
// add coding to handle dependencies for example add some more
elements depending on attribute specified in dependson
…
// build array of StorageObjects
```

```
var objects = new Array(l1,l2);
var locs = toJavaArray(objects); // locs is the return variable

// convert JavaScript Array to Java Array
function toJavaArray(jsArray) {
    var javaArray = java.lang.reflect.Array.newInstance(
StorageObject, jsArray.length);
    for (var i=0; i<jsArray.length; i++) {
        javaArray[i] = jsArray[i];
    }
    return javaArray;
}
```

### 10.1.6. LDAP Searches in JavaScript

Sometimes you need to perform LDAP searches in a JavaScript file, either in account creation or in consistency rules. DirX Identity lets you access the existing LDAP connection and offers a simple interface to receive the search result.

First you need a storage object from the script context, which holds the LDAP connection. Then you create a new object collection, which represents the search result. Pass the storage object and the usual LDAP search attributes to its constructor. See the following code snippet for a sample:

```
searchBase = "cn=Accounts,cn=myTargetSystem,cn=TargetSystems,cn=My
Domain";
filter = ...;
stg = scriptContext.getObject.getStorage();
searchResult = new ObjectCollection(stg, searchBase, filter, 2, 0, 0,
attrs, sort_attrs, ascending, null);
```

The other parameters are known from the LDAP interface of the Mozilla Netscape API: scpe, sizelimit, timelimit, requested attributes, sorting attributes.

### 10.1.7. Creating a Provisioning Entry

The following script shows how to create an entry - here a permission - in a provisioning domain:

```
importPackage(java.lang);
var obj=scriptContext.getObject();
// the parent folder for the permission to be created
var permissionFolder = "cn=Test,cn=Permissions,cn=My-Company";
```

```
// Create a permission with common name MyTestPermission in
permissionFolder
var permission = createPermission(permissionFolder, "
MyTestPermission");
if (permission != null) {
   scriptContext.logInfo("Permission dn="+permission.getDN()+"
successfully created.");
   result = "SUCCESS";
}
else {
   result = "FAILURE";
}

// creates a permission with DN cn=<commonName>,<parentDN>
// using the object description with name dxrPermission.
function createPermission(parentDN, commonName) {
   var storage = scriptContext.getUserStorage();
   var od = storage.getObjectDescriptor("dxrPermission");
   var permission = storage.createObject(od, parentDN, commonName);
   try {
     permission.save();
   }
   catch (e) {
     scriptContext.logError("Create permission "+commonName+" failed:
Error during save: "+e.toString());
     permission = null;
   }
   return permission;
}
```

### 10.1.8. Other Useful Methods

This section explains some other useful methods:

- **obj.save()** - saves the object.
- **obj.checkAndSave()** - runs a privilege resolution and saves the object.

### 10.1.9. Logging and Debugging

You can write your own debug messages:

```
obj.logger.log("DBG", "_text_");
```

for example

```
obj.logger.log("DBG", "dxrNameForAccount: tries="+tries);
```

To view the debug messages in the DirX Identity Manager log, set the following switches in the dxi.cfg file:

```
trace.level=9
trace.transcript=on
```

The second switch doubles some standard messages, that means you should use this switch only for debugging. For example the check for uniqueness is logged per default as well as the call of the Java Script and the result. You will find the messages in the **system.nnn.log** files in the folder *install_path***GUI\log**.

An example logging with the above described debug message looks like:

```
DBG(SVC102): Checking uniqueness of attribute=dxrname
value=FFFFLLLLLLLL in subtree cn=ADS,cn=TargetSystems,cn=My-Company
DBG(SVC102): Check failed!
LOG(STG200): execute script 'dxrNameForAccounts' ...
LOG(STG200): execute script 'dxrNameForAccounts' ...
DBG(SVC102): dxrNameForAccount: tries=3
LOG(STG200): result is 'FFFFFLLLLLLL' (0ms)
LOG(STG200): result is 'FFFFFLLLLLLL' (0ms)
DBG(SVC613): Saving object
DN=cn=test4,cn=Accounts,cn=TargetSystems,cn=My-Company.
```

# 10.2. Writing Java Extensions

This chapter contains hints for writing Java extensions, including:

- How to write an extension to implement a consistency rule
- How to write, configure and deploy a custom status module for users, personas and functional users

## 10.2.1. Writing an Extension to Implement a Consistency Rule

You can write your own Java extensions; for example, to implement a consistency rule.

The next sections describe a sample for a consistency rule using a Java action. The sample shows the interrelation of the Java program, the configuration and how to install it.

#### 10.2.1.1. About the Java Action

The sample contains a Java action implemented in the method **setSubjectAttribute** of class **com.mycompany.actions.ConsistencyActions**.

Object setSubjectAttribute(StorageObject node, String attribute, String namingRule);

The method is used to create a new value for the subject's attribute whose name is given in 'attribute' by evaluation of the supplied **namingRule**. The subject is passed in the parameter node. In simulation mode, the change is reported but node is not saved. In real mode, the node is saved making the change permanent.

#### 10.2.1.2. Building the Project

We assume here that you are familiar with building Java projects and that ant and a Java compiler are installed and in your path.

To build the project, an ant build script (**build.xml**) is provided. The path to the DirX Identity installation must be adjusted: just correct the value of the property **metarole.dir**.

A successful run of ant will create the jar-file **rules.jar** in the local lib directory.

#### 10.2.1.3. Installing rules.jar

To make **rules.jar** available to the DirX Identity Policy Agent, it must be added to the agent's class path. The path for rules.jar must be configured:

```
REM add custom rules.jar here
SET MYRULES=C:\my-company\rules\lib\rules.jar
```

Next, the classpath of the original batch script **MetaRolePolicyAgent.bat** must be extended by %MYRULES%, for example:

```
"%DXI_JAVA_HOME%\bin\java" %debug% -Xmx512m -classpath "%DXICP%;%MYRULES%"
-Djava.net.preferIPv4Stack=true siemens.dxr.policy.agent.PolicyAgent ...
```

Don't just copy this line from the original: extend -classpath ("...;%MYRULES%") to ensure that the currently referenced jars are not removed from the path.

#### 10.2.1.4. Configuring the Action

The action is configured in the Operations tree of the Policy view of DirX Identity Manager. You may add some OperationContainers to structure this view. Then, a 'Java Class' object must be added whose name corresponds to the fully qualified classname of the Java class that implements the actions.

In our sample, this is **com.mycompany.actions.ConsistencyActions**.

Next, select the 'Java Class' configuration object and add a 'Java Method Action' for each method of this class that is intended to be used in consistency rules.

In our example, this is the method 'setSubjectAttribute'.

Next, configure the method's parameters. The variable node contains the 'subject', the object being returned by the search configured in the consistency rule. Choose "subject" for the parameter's context.

Attribute and namingRule are not determined at runtime but must be supplied by the rule using the action. Thus, choose "fixed" for their context. You can add more actions that are implemented by the same class.

#### 10.2.1.5. Viewing the Configuration Sample

A configuration sample for the My-Company sample domain is provided in the file **rules.zip** on your DVD in the folder:

**Documentation\DirXIdentity**

Just import this sample. The action is configured under Operations/Java action samples/Java, a sample rule is provided under Rules/Java consistency rule samples.

The rule 'create account description' adds a description for all accounts that have an empty description attribute. It references the action 'setSubjectAttribute'.

When the rule is run, the policy agent performs the search defined in the rule's filter. This search returns all accounts with empty description. For each account, the method 'setSubjectAttribute' is called with the current account and the fixed parameters 'attribute' and 'namingRule' defined by the rule.

You can define multiple rules that make use of the same action; for example, you could modify a user's attribute that makes use of another naming rule.

#### 10.2.1.6. Handling Multi-value Attributes

If you want to handle a multi-value attribute, you need to define this attribute to multi-value in the user.xml definition. Then you can work on this attribute (in this example, the attribute 'verified'):

```
importPackage(java.lang.reflect);

var verified = obj.getProperty("verifiedBy");

for (var j=0; j<java.lang.reflect.Array.getLength(verified); j++) {
    if ( verified[j].equals("somestring") ) {
```

```
        dosomething();
    }
}
```

### 10.2.2. Implementing Custom User Status Modules

The section "Managing States" in the *DirX Identity Provisioning Administration Guide* describes the default state machines for users, personas and functional users provided by DirX Identity.

To write your own customized state machines for users, personas or functional users, you:

- Write your own Java class that implements the StateMachine interface
- Configure and deploy your customized state machine for the object

#### 10.2.2.1. Implementing the StateMachine Interface

The StateMachine interface contains String constants for the states and a definition of the method calculateState:

```
package siemens.dxr.service.states.api;
import siemens.dxr.service.ServiceException;
import siemens.dxr.service.nodes.SvcNode;
import siemens.dxr.service.store.SvcSession;
public interface StateMachine {
    public static final String ENABLED = "ENABLED";
    public static final String TBDEL = "TBDEL";
    public static final String TEMPLATE = "TEMPLATE";
    public static final String NEW = "NEW";
    public static final String DISABLED = "DISABLED";
    /**
    * Calculate the state for a user, persona or functional user
    * @param session The session holding the ldap bind
    * @param obj The object whose state is to be calculated
    * @return The new state
    */
     public String calculateState(SvcSession session, SvcNode obj)
throws ServiceException;
}
```

Here is the implementation of the persona's standard state machine. It is adapted from the user's state machine with the following extensions:

- The first statement in the method reads the state of the associated user (the persona's owner). If this state equals TBDEL, no further calculation is done but the state is also used for the persona.
- At the end of the method, it is checked if the user's state equals DISABLED and the persona's state is NOT TBDEL. In this case, DISABLED is also used as the persona's state.

```
package siemens.dxr.service.states.impl;

import com.siemens.dirxcommon.logging.LogSupport;
import siemens.dxm.storage.StorageException;
import siemens.dxm.storage.StorageObject;
import siemens.dxm.util.GeneralizedTime;
import siemens.dxr.service.ISvcCodes;
import siemens.dxr.service.PropName;
import siemens.dxr.service.ServiceException;
import siemens.dxr.service.msgout.MsgID;
import siemens.dxr.service.nodes.SvcNode;
import siemens.dxr.service.states.api.StateMachine;
import siemens.dxr.service.store.SvcSession;

public class PersonaStateMachine implements StateMachine {

    LogSupport logger = LogSupport.forName(PersonaStateMachine.
class);

    @Override
    public String calculateState(SvcSession session, SvcNode obj)
throws ServiceException {
        String userState = getAssociatedUserState(session, obj);
        // take TBDEL state from user
        if (StateMachine.TBDEL.equals(userState)) {
            return userState;
        }
        int rc = ISvcCodes.SVC_UNCHANGED;
        String state;
        String newState = obj.getValue(PropName.dxrState);
        int counter = 0;
        // create actDate and initialize with today's date
        GeneralizedTime actDate = new GeneralizedTime();
        actDate.initDate();
        // get dates
        GeneralizedTime startDate = (GeneralizedTime) obj.
```

```
getProperty(PropName.dxrStartDate);
        GeneralizedTime disableStartDate = (GeneralizedTime) obj
.getProperty(PropName.dxrDisableStartDate);
        GeneralizedTime disableEndDate = (GeneralizedTime) obj
.getProperty(PropName.dxrDisableEndDate);
        GeneralizedTime endDate = (GeneralizedTime) obj.getProperty
(PropName.dxrEndDate);
        do {
            state = newState;
            // TEMPLATE does not change
            if (state.equalsIgnoreCase(StateMachine.TEMPLATE)) {
            } else if (state.equalsIgnoreCase(StateMachine.NEW)) {
                // start date empty or reached ==> change to ENABLED
                if (startDate == null || (startDate.compareDateTo
(actDate) <= 0)) {
                    newState = StateMachine.ENABLED;
                }
            } else if (state.equalsIgnoreCase(StateMachine.ENABLED))
{
                // start date in future ==> change to NEW
                if ((startDate != null) && (startDate.compareDateTo
(actDate) > 0)) {
                    newState = StateMachine.NEW;
                }
                // disableStartDate reached ==> disable user
                if ((disableStartDate != null) && (disableStartDate
.compareDateTo(actDate) <= 0)) {
                    if (disableEndDate == null || (disableEndDate
.compareDateTo(actDate) >= 0)) {
                        newState = StateMachine.DISABLED;
                    }
                }
                // end date passed ==> change to TBDEL
                if (endDate != null && endDate.compareDateTo(actDate)
< 0) {
                    newState = StateMachine.TBDEL;
                }
            } else if (state.equalsIgnoreCase(StateMachine.DISABLED))
{
                // disableStartDate empty or in future ==> enable
user
```

```
                if (disableStartDate == null || disableStartDate
.compareDateTo(actDate) > 0) {
                    newState = StateMachine.ENABLED;
                }
                // disableEndDate passed ==> enable user
                if (disableEndDate != null && disableEndDate
.compareDateTo(actDate) < 0) {
                    newState = StateMachine.ENABLED;
                }
                // end date passed ==> change to TBDEL
                if (endDate != null && endDate.compareDateTo(actDate)
< 0) {
                    newState = StateMachine.TBDEL;
                }
            } else if (state.equalsIgnoreCase(StateMachine.TBDEL)) {
                // Enable user in case end date does not exist or is
in future
                if (endDate == null || endDate.compareDateTo(actDate)
> 0) {
                    newState = StateMachine.ENABLED;
                }
            } else {
                // User DN=%s has an illegal state: '%s'.
                logger.log(MsgID.ERR_USER_STATE_WRONG, obj.getDN(),
state);
                rc = ISvcCodes.SVC_RESOLUTION_ERROR;
                throw new ServiceException("Illegal state detected",
rc);
            }
            logger.log(MsgID.DBG, "persona state is " + state + ",
newState is " + newState);
        }
        while (!state.equalsIgnoreCase(newState) && counter++ < 20);
        // state cycle detected
        if (!state.equalsIgnoreCase(newState)) {
            // State cycle detected for user DN=%s: last state=%s,
new state=%s.
            logger.log(MsgID.ERR_USER_STATE_CYCLE, obj.getDN(),
state, newState);
            rc = ISvcCodes.SVC_RESOLUTION_ERROR;
            throw new ServiceException("State cycle detected", rc);
```

```
        }
        // if the user is DISABLED and the persona is not TBDEL ==>
DISABLE persona
        if (!StateMachine.TBDEL.equals(state) && StateMachine
.DISABLED.equals(userState)) {
            state = userState;
        }
        return state;
    }

    /**
     * Checks whether the associated user exists and is in TBDEL
state.
     *
     * @param session The session holding the ldap bind
     * @param obj     The object whose state is to be calculated
     * @return The associated user's state
     */
    private String getAssociatedUserState(SvcSession session, SvcNode
obj) {
        String associatedEntryDN = obj.getValue(PropName.owner);
        if (associatedEntryDN != null && associatedEntryDN.length() >
0) {
            try {
                StorageObject associatedEntry = session.getObject
(associatedEntryDN);
                return associatedEntry.getValue(PropName.dxrState);
            } catch (StorageException e) {
                // Cannot retrieve state of associated user DN={0} of
persona DN={1}: {2}.
                logger.log(MsgID.ERR_ASSOC_USER_STATE,
associatedEntryDN, obj.getID());
            }
        }
        return null;
    }
}
```

#### 10.2.2.2. Configuring the Custom Status Module

The status module called for state calculations is configured in the object description; for

example, PersonaCommon.xml for a persona. It uses the **defaultstatusmodule** element. This element must contain the fully-qualified Java class name of your implementation:

`<defaultstatusmodule>siemens.dxr.service.states.impl.PersonaStateMachine</defaultstatusmodule>.`

You can configure your own status modules for all user types for which you have created your own object descriptions.As long as the **defaultstatusmodule** tag is not defined for a user type, the standard status handling of the user is used.

#### 10.2.2.3. Deploying the Custom Status Module

Compile your custom Java class into a **jar** file and then copy this **jar** file to all of the locations at which **dxrServices.jar** is installed.This is the only task you need to perform to have the Java-based Server, Web Center and Web Services find your code.

For applications that are started by a batch script (for example, DirX Identity Manager, the Policy Agent, the Service Agent, and so on) you must also add your custom **jar** file to the classpath or a **ClassNotFoundException** will occur.

## 10.3. Customizing User Creation Workflows

This section describes how to customize the creation workflows for personas, user facets and functional users.

### 10.3.1. Customizing the Persona Create Workflow

Whenever you use Web Center to create a new persona for a selected user, it starts the persona create workflow.This section describes the workflow's PersonaFromUser activity and gives hints on how to customize it to your requirements.

When it starts the PersonaFromUser workflow, Web Center adds two attributes to the workflow context:

- **associatedEntry** - the DN of the persona's owner
- **destinationType** - the object description name of the object to be created; that is, **dxrPersona** for a persona

These two context attributes are configured in the workflow's PersonaFromUser activity with the parameters **Context attribute for associated entry**/**destinationType**. As long as you do not change Web Center's configuration of the attribute names, do not change these settings.

The PersonaFromUser activity first uses these context attributes to calculate the parent folder for the new persona:

- The activity reads the associated entry using the DN contained in the **associatedEntry** context attribute

- The activity uses associated entry's parent folder as the default folder for the new persona. In the PersonaFromUser activity, you can override this default folder by configuring the activity's **Parent Folder for Subject** parameter. In Web Center, whether you can change the default folder depends on the configuration of the Enter Attributes' User Base parameter: if the parameter is configured, the default value can be changed by the administrator. If not, the folder is not visible in Web Center and the default value is used.

The PersonaFromUser activity next uses the context attributes to calculate default values for some attributes using the associate entry as a template. To calculate these values, the activity uses a special object description that exists only for this purpose. To determine the object description's name, the activity:

- Reads the object description name from the **destinationType** context attribute. For a persona, this is **dxrPersona**, that is, the object description for a persona object if it is read from the Identity Store or is created in Identity Manager.
- Searches the associated entry's object description for the createFrom element to get the name of the object description for persona creation. For a user, two createFrom elements are present in **UserCommon.xml**:

```
<createFrom destinationType="dxrPersona"
objectDescriptionName="PersonaFromUser"/>
<createFrom destinationType="dxrFunctionalUser"
objectDescriptionName="FunctionalUserFromUser"/>
```

With the **destinationType dxrPersona**, the first createFrom tag matches and the activity creates the default values by applying the PersonaFromUser object description to the **associatedEntry**.

This mechanism has the advantage of allowing you to use the persona create workflows to create personas from personas (that is, a persona is the persona's owner). For this purpose, the following createFrom tags are defined in **PersonaCommon.xml**:

```
<createFrom destinationType="dxrFunctionalUser"
objectDescriptionName="FunctionalUserFromPersona"/>
<createFrom destinationType="dxrPersona"
objectDescriptionName="PersonaFromPersona"/>
```

You can override this object description name by configuring the PersonaFromUser activity's **Name of Object Description** parameter.

The PersonaFromUser activity creates a new persona object by applying the object description for persona creation (for example, **UserToPersona**) to the **associatedEntry** (which is a user in this case). You can modify this object description according to your needs.

The new persona's common name is created using the JavaScript **CommonNameForPersona.js**:

```
<property name="cn" defaultvalue="$(script:CommonNameForPersona)">
    <script name="CommonNameForPersona"
    return="cn"

reference="storage://DirXmetaRole/cn=CommonNameForPersona.js,cn=JavaS
cripts,cn=Customer
Extensions,cn=Configuration,$(rootDN)?content=dxrObjDesc"/>
</property>
```

The script uses the user's sn and givenName together with a UID to create a unique common name:

```
// calculates a unique common name for a persona
  importPackage(java.lang);
  var obj=scriptContext.getObject();
  if (obj.isNewObject()) {
     var givenName = obj.getValue("givenName");
     var sn = obj.getValue("sn");
     var uid = obj.getStorage().getUID("persona");
     obj.setProperty("dxrUID", uid);
     var cn = sn + " " + givenName + " " + uid;
  }
  var result = "SUCCESS";
```

You can change this script to use a more convenient common name that complies with your processes.

### 10.3.2. Customizing the User Facet Create Workflow

Whenever you use Web Center to create a new user facet for a selected user, it starts the user facet create workflow.

### 10.3.3. Customizing the Functional User Create Workflow

Whenever you use Web Center to create a new functional user for a selected user, it starts a functional user create workflow. This section describes the workflow's FunctionalUserFromUser activity and gives hints on how to customize it to your requirements.

When it starts the FunctionalUserFromUser workflow, Web Center adds two attributes to

the workflow context:

- **associatedEntry** - the DN of the functional user's sponsor
- **destinationType** - the object description name of the object to be created; that is, **dxrFunctionalUser** for a functional user

These two context attributes are configured in the workflow's FunctionalUserFromUser activity with the parameters **Context attribute for associated entry/destinationType**. As long as you do not change Web Center's configuration of the attribute names, do not change these settings.

The FunctionalUserFromUser activity first uses these context attributes to calculate the parent folder for the new functional user:

The activity reads the associated entry using the DN contained in the associatedEntry context attribute. The activity uses the associated entry's parent folder the default folder for the new functional user. In the activity, you can override this default folder by configuring the FunctionalUserFromUser activity's **Parent Folder for Subject** parameter. The sample domain does this for the functional user create workflows, since it is a common practice to place the resources under a common folder. In Web Center, whether or not you can change this default folder depends on how the **Enter Attributes' User Base** parameter is configured: if the parameter is configured, the default value can be changed by the administrator. If it is not configured, the folder is not visible in Web Center and the default value is used. Thus, for the create functional user workflows, the **Enter Attributes' User Base** parameter is left empty, and all functional users are created under the folder configured in the **Parent Folder for Subject** parameter. See the section "Customizing the Persona Create Workflow" for instructions on how to configure a parent folder that can be changed by the administrator.

The FunctionalUserFromUser activity next uses these context attributes to calculate default values for some attributes, using the associate entry as a template. To calculate these values, the activity uses a special object description that exists only for this purpose. To determine the object description's name, the activity:

- Reads the object description name from the **destinationType** context attribute. For a functional user, this is **dxrFunctionalUser**; that is, the object description for a functional user object if it is read from the Identity Store or is created in DirX Identity Manager.
- Searches for the createFrom element in the associated entry's object description to get the name of the object description for functional user creation. For a user, two createFrom elements are present in **UserCommon.xml**:

```
<createFrom destinationType="dxrPersona"
objectDescriptionName="PersonaFromUser"/>
<createFrom destinationType="dxrFunctionalUser"
objectDescriptionName="FunctionalUserFromUser"/>
```

With the **destinationType dxrFunctionalUser**, the second createFrom tag matches and the activity creates the default values by applying the **FunctionalUserFromUser** object

description to the **associatedEntry**.

This mechanism has the advantage of allowing the functional user create workflows to be used to create functional users from personas (that is, a persona is the functional user's sponsor). For this purpose, the following createFrom tags are defined in PersonaCommon.xml:

```
<createFrom destinationType="dxrFunctionalUser"
objectDescriptionName="FunctionalUserFromPersona"/>
<createFrom destinationType="dxrPersona"
objectDescriptionName="PersonaFromPersona"/>
```

You can override this object description name by configuring FunctionalUserFromUser activity's **Name of Object Description** parameter.

The FunctionalUserFromUser activity creates a new functional user object by applying the object description for functional user creation (for example, **FunctionalUserFromUser**) to the **associatedEntry** (which is a user in this case). You can customize this object description according to your needs.

The new functional user's common name is created using the JavaScript **CommonNameForFunctionalUsers.js**. The script uses a UID to create a unique common name:

```
importPackage(java.lang);
var obj=scriptContext.getObject();
if (obj.isNewObject()) {
    var cn = obj.getStorage().getUID("persona");
    obj.setProperty("dxrUID", cn);
}
var result = "SUCCESS";
```

In Web Center, the common name is editable, so you can use this UID or enter a new common name for your resource.

You can also change this script to use a more convenient common name that complies with your processes.

## 10.4. Customizing the Consistency Rules for Automatic Creation

This section describes how to customize the consistency rules for automatic persona, functional user and user creation.

### 10.4.1. Customizing the Consistency Rule for Automatic Persona Creation

If multiple accounts in a target system are assigned to a single user, DirX Identity only manages one of these accounts: the primary account.DirX Identity's role resolution process only manages the primary account's group assignments and state.

In the chapter "Managing Target Systems" in the *DirX Identity Provisioning Administration Guide*, the subsections "Full Provisioning" and "Partial Provisioning" (see the section "Managing Personal Accounts") describe the different options for handling multiple accounts in a target system.If you decide to use the "Full Provisioning" approach, you must create personas for each assigned non-primary account.You can create the personas interactively in Web Center, and then manually link the account to the new persona, or you can use the consistency rule **CreatePersonasForNonPrimaryAccounts** to automate the process.

The **CreatePersonasForNonPrimaryAccounts** rule offers you the following features:

- It checks to see if a primary account already exists for the user in the account's target system.
- Its persona creation process is configured in a JavaScript file that can be specific to the account's target system. The JavaScript controls which attributes are read from the account, which attributes are read from the user, and the new persona's parent folder and naming.
- The user referenced from the account by its dxrUserLink is the persona's owner.
- It re-links the accounts' dxrUserLink to the new persona.
- It can either create the new persona directly in the Identity Store, or it can start an approval workflow for persona creation that permits an approver to make changes to some attributes.

The next sections describe the components of this rule and their default operations and indicate how they can be customized.

#### 10.4.1.1. How the Consistency Rule Works

The **CreatePersonasForNonPrimaryAccounts** consistency rule is located under the path **Policies→Rules→Default → Consistency→Personas**. By default, the rule performs the account search under all target systems with the following filter:

```
( objectclass="dxrTargetSystemAccount" and dxruserlink=* and ( not (
dxrisprimary=* ) or dxrisprimary="false" ) )
```

As you can see from the filter definition, the rule processes all accounts that are already assigned to a user and which are not flagged as primary accounts.

The rule passes to its **AccountToPersona** operation the name of the JavaScript that controls the persona creation process (in the **nameOfJavaScript** parameter). By default, the

JavaScript **AccountToPersona.js** is configured as the name.

#### 10.4.1.2. How the Operation Works

The **AccountToPersona** operation is located under the path **Policies→Operations→Default → Java→ siemens.dxr.policy.actions.PersonaManagement →AccountToPersona**.

First, the operation checks whether the user already has a primary account in the target system. If this is not the case, an error message is issued and the operation is aborted for the selected account.

Next, the operation next searches for JavaScripts with the RDN provided in the **nameOfJavaScript** parameter (by default, **AccountToPersona.js**) in the following order:

- In the **JavaScripts** container of the account's target system folder (target system-specific configuration).
- In the **JavaScripts** container of the **Customer Extensions** folder. This folder is a customer-specific default configuration that is independent of the target system.
- In the system configuration's **JavaScripts** container. A template JavaScript is installed in this location. We recommend copying this template to the appropriate location - target system-specific or general **Customer Extensions** - and then tailoring it to your requirements.

If the operation finds a JavaScript with the configured RDN, it runs it. This JavaScript controls the creation of the persona from the user's and the account's data.

#### 10.4.1.3. How the JavaScript Works

The **AccountToPersona.js** JavaScript calls a handler, implemented in Java that contains the program logic. In its configuration section, the JavaScript provides the following variables for configuring this handler:

- createGroupAssignments - a Boolean flag. When set to **true**, the handler creates direct group assignments for the groups assigned to the account.
- userAttributesToCopy - an array of strings that contain the names of attributes to be copied from the related user to the persona.
- userAttributesToCopyWithNameMapping - similar to userAttributesToCopy, but source and target attribute names are different and are separated by a period (.). For example, **dn.owner** specifies that the user's dn is copied to the persona's owner attribute.
- accountAttributesToExclude - an array of strings that contain the names of attributes that are not to be copied. By default, the handler copies all attributes that exist in the persona's and in the account's object description with the same name. Use this variable to define attributes that the handler should not copy.
- accountAttributesToCopyWithNameMapping - the attributes to copy from the account to the persona with different source and target attribute names, separated by a period (.).
- approvalWorkflowDN - the DN of an approval workflow to be started for approval of persona creation.

The configuration section is followed by:

- Providing the handler, by calling scriptContext.getHandler().
- Reading account and user from the handler.
- Supplying the persona's parentID with the parentID of the user.
- Creating the persona's cn from the user's and the account's cn.
- Initializing the handler with the configured persona's name and parent folder parameters: handler.initialize(cnOfPersona, parentID).
- Calling the handler's preparePersona method. This method performs all copying actions defined in the configuration section. This call creates a complete persona in memory with all attributes populated according to the variable definitions given the configuration section.
- Reading the persona from the handler, handler.getPersona()
- Calling the JavaScript "addAttributes" method. This method provides the option of creating persona attribute values using the full flexibility of JavaScript. In the sample, the description is populated.
- Selecting whether to create the persona with an approval workflow or directly in the Identity Store:
  - To start an approval workflow, use handler.startApprovalWorkflow(approvalWorkflowDN).
  - To create the persona directly in the Identity Store, use handler.createPersonaInLdap().

#### 10.4.1.4. How the Approval Workflow Works

The JavaScript variable **approvalWorkflowDN** contains the DN of the sample workflow **Approve Create Persona from Account**.

This workflow is a simplified variant of the **Create Persona** workflows, with only one approval activity - a Calculate GUID activity - and an Apply Change activity. The workflow does not contain any Enter Attributes or Request Privileges activities, since this data is already provided by the consistency rule in the form of subject and resource orders that are passed to the workflow when it is started. The absence of activities for entering data and assignments makes this workflow unsuitable for creating a new persona in Web Center.

You can tailor the **Approve Create Persona from Account** workflow to your needs; for example, you can change the set of attributes in the approval step, remove the calculate GUID activity or replace it with your own GUID generating activity. But note that the Apply Change activity uses a user hook called **siemens.dxr.service.order.impl.ModifyAccountUserHook**. This user hook is mandatory for re-linking the **dxrUserLink** to the persona prior to saving (and resolving) the new persona; otherwise the role resolution performed in the Apply Change step would create a new account for the new persona if direct group assignments are created for the persona.

## 10.4.2. Customizing the Consistency Rule for Automatic Functional User Creation

If you have imported accounts that cannot be assigned to a user, and you want to use the “Full Provisioning” approach (see the subsection "Full Provisioning" in the section "Managing Personal Accounts" in the chapter "Managing Target Systems" in the *DirX Identity Provisioning Administration Guide*), you need to decide whether you want to create a user or a functional user for the account. If the account represents a resource, like a global mailbox, a meeting room, or a trainee account, you may want to create a functional user for this account, rather than a user.

You can create the functional user interactively in Web Center and then manually link it to this functional user, or use the **CreateFunctionalUsersForUnassignedAccounts** consistency rule for automatic functional user creation.

You must be extremely careful when running this rule, since it will create functional users for all unassigned accounts of all target systems, and this is not easy to undo! You are responsible for ensuring that this action is only executed on accounts for which it makes sense to create a functional user. To accomplish this task, you can:

- Run the rule in simulation mode first and examine its results, if you want to create the functional users directly in the Identity Store.
- Create an approval workflow for each functional user, where the approval step verifies that functional user creation makes sense.

The **CreateFunctionalUsersForUnassignedAccounts** rule offers you the following features:

- Its functional user creation process is configured in a JavaScript file that can be specific to the account’s target system. The JavaScript controls which attributes are read from the account, and the new functional user’s parent folder and naming.
- It links the account’s dxrUserLink to the new functional user.
- It can create the new functional user directly in the Identity Store or start an approval workflow for functional user creation that permits an approver to make changes to some attributes or reject the functional user creation entirely.

The next sections describe the components of this rule and their default operations and indicate how they can be customized.

#### 10.4.2.1. How the Consistency Rule Works

The **CreateFunctionalUsersForUnassignedAccounts** consistency rule is located under the path **Policies**→ **Rules**→ **Default**→ **Consistency**→ **FunctionalUsers**. By default, the rule performs the account search under all target systems using the filter definition:

```
( objectclass="dxrTargetSystemAccount" and not ( dxruserlink=* ) )
```

As you can see from the filter definition, the rule processes all accounts that are not already assigned to a user.

The rule passes name of the JavaScript that controls the functional user creation (in the parameter **nameOfJavaScript**) to its **AccountToFunctionalUser** operation. By default, the **AccountToFunctionalUser.js** JavaScript is configured as the name.

#### 10.4.2.2. How the Operation Works

The **AccountToFunctionalUser** rule operation is located under the path **Policies**→ **Operations**→ **Default**→ **Java**→ **siemens.dxr.policy.actions.FunctionalUserManagement** → **AccountToFunctionalUser**.

The operation searches for JavaScripts with the RDN provided in the nameOfJavaScript parameter (by default, the name **AccountToFunctionalUser.js**) in the following order:

- In the **JavaScripts** container of the account's target system (target system-specific configuration).
- In the **JavaScripts** container of the **Customer Extensions** folder. This folder is a customer-specific default configuration that is independent of the target system.
- In the system configuration's **JavaScripts** container. Here a template JavaScript is installed in this location. We recommend copying this template to the appropriate location (target system-specific or general **Customer Extension**) and tailoring it to your requirements.

If the operation finds a JavaScript with the configured RDN, it runs it. This JavaScript controls the creation of the functional user from the account's data.

#### 10.4.2.3. How the JavaScript Works

The **AccountToFunctionalUser.js** JavaScript calls a handler, implemented in Java that contains the program logic. In its configuration section, the JavaScript provides the following variables for configuring this handler:

- createGroupAssignments - a Boolean flag. When set to **true**, the handler creates direct group assignments for the groups assigned to the account.
- accountAttributesToExclude - an array of strings that contain the names of attributes that are not to be copied. By default, the handler copies all attributes that exist in the functional user's and in the account's object descriptions with the same name. Use this variable to define attributes that the handler should not copy.
- accountAttributesToCopyWithNameMapping - the attributes to copy from the account to the functional user with different source and target attribute names, separated by a period (.).
- approvalWorkflowDN - the DN of an approval workflow to be started for approval of functional user creation.

The configuration section is followed by:

- Providing the handler, by calling scriptContext.getHandler().
- Reading the account from the handler.
- Supplying the functional user's parentID with a constant DN.

- Creating the functional user's cn from the account's cn.
- Initializing the handler with the configured functional user name and parent folder: handler.initialize(cnOfFunctionalUser, parentID).
- Calling the handler's prepareFunctionalUser method. This method performs all copying actions defined in the configuration section. This call creates a complete functional user in memory with all attributes populated according to the variable definitions given the configuration section.
- Reading the functional user from the handler, handler.getFunctionalUser().
- Calling the JavaScript "addAttributes" method. This method provides the option of creating functional user attribute values using the full flexibility of JavaScript. In the sample, the description is populated.
- Selecting whether to create the functional user with an approval workflow or directly in the Identity Store:
  - To start an approval workflow, use handler.startApprovalWorkflow(approvalWorkflowDN).
  - To create the functional user directly in the Identity Store, use handler.createFunctionalUserInLdap().

#### 10.4.2.4. How the Approval Workflow Works

The JavaScript variable **approvalWorkflowDN** contains the DN of the sample workflow **Approve Create Functional User from Account**.

This workflow is a simplified variant of the Create Functional User workflows, with only one approval activity - a Calculate GUID activity - and an Apply Change activity. The workflow does not contain any Enter Attributes or Request Privileges activities, since this data is already provided by the consistency rule in the form of subject and resource orders that are passed to the workflow when it is started. The absence of activities for entering data and assignments makes this workflow unsuitable for creating a new functional user in Web Center.

You can tailor the **Approve Create Functional User from Account** workflow to your needs; for example, you can change the set of attributes in the approval step, remove the calculate GUID activity or replace it with your own GUID generating activity. But note that the Apply Change activity uses a user hook called **siemens.dxr.service.order.impl.ModifyAccountUserHook**. This user hook is mandatory for re-linking the **dxrUserLink** to the functional user prior to saving (and resolving) the new functional user; otherwise the role resolution performed in the Apply Change step would create a new account for the new functional user if direct group assignments are created for the functional user.

### 10.4.3. Customizing the Consistency Rule for Automatic User Creation

If you have imported accounts that cannot be assigned to a user, and you want to use the "Full Provisioning" approach (see the subsection "Full Provisioning" in the section "Managing Personal Accounts" in the chapter "Managing Target Systems" in the *DirX Identity Provisioning Administration Guide*), you must decide whether you want to create a

user or a functional user for the account.

If the criteria for creating a functional user do not apply, or you have a target system that is connected to your human resources system that is the master for user management, you may want to create a user from the unassigned account.

You can create the functional user interactively in Web Center and then manually link it to this functional user, or use the **CreateUsersForUnassignedAccounts** consistency rule for automatic user creation.

You must be extremely careful when running this rule, since it will create users for all unassigned accounts of all target systems, and this is not easy to undo! You are responsible for ensuring that this action is only executed on accounts for which it makes sense to create a user. To accomplish this task, you can:

- Run the rule in simulation mode first and examine its results, if you want to create the users directly in the Identity Store.
- Create an approval workflow for each functional user, where the approval step verifies that functional user creation makes sense.

The **CreateUsersForUnassignedAccounts** rule offers you the following features:

- Its user creation process is configured in a JavaScript file that can be specific to the account's target system. The JavaScript controls which attributes are read from the account, and the new user's parent folder and naming.
- It links the account's dxrUserLink to the new user.
- It can create the new user directly in the Identity Store or start an approval workflow for user creation that permits an approver to make changes to some attributes or reject the user creation entirely.

The next sections describe the components of this rule and their default operations and indicate how they can be customized.

#### 10.4.3.1. How the Consistency Rule Works

The **CreateUsersForUnassignedAccounts** consistency rule is located under the path **Policies**→ **Rules**→ **Default**→ **Consistency**→ **Users**. By default, the rule performs the account search under all target systems using the filter definition:

```
( objectclass="dxrTargetSystemAccount" and not ( dxruserlink=* ) )
```

As you can see from the filter definition, the rule processes all accounts that are not already assigned to a user.

The rule passes name of the JavaScript that controls the functional user creation (in the parameter **nameOfJavaScript**) to its **AccountToUser** operation. By default, the **AccountToUser.js** JavaScript is configured as the name.

#### 10.4.3.2. How the Operation Works

The **AccountToUser** rule operation is located under the path **Policies**→ **Operations**→ **Default**→ **Java**→ **siemens.dxr.policy.actions.UserManagement** → **AccountToFunctionalUser**.

The operation searches for JavaScripts with the RDN provided in the nameOfJavaScript parameter (by default, the name **AccountToUser.js**) in the following order:

- In the **JavaScripts** container of the account's target system (target system-specific configuration).
- In the **JavaScripts** container of the **Customer Extensions** folder. This folder is a customer-specific default configuration that is independent of the target system.
- In the system configuration's **JavaScripts** container. A template JavaScript is installed in this location. We recommend copying this template to the appropriate location (target system-specific or general **Customer Extension**) and tailoring it to your requirements.

If the operation finds a JavaScript with the configured RDN, it runs it. This JavaScript controls the creation of the user from the account's data.

#### 10.4.3.3. How the JavaScript Works

The **AccountToUser.js** JavaScript calls a handler, implemented in Java that contains the program logic. In its configuration section, the JavaScript provides the following variables for configuring this handler:

- createGroupAssignments - a Boolean flag. When set to **true**, the handler creates direct group assignments for the groups assigned to the account.
- accountAttributesToExclude - an array of strings that contain the names of attributes that are not to be copied. By default, the handler copies all attributes that exist in the user's and in the account's object descriptions with the same name. Use this variable to define attributes that the handler should not copy.
- accountAttributesToCopyWithNameMapping - the attributes to copy from the account to the user with different source and target attribute names, separated by a period (.).
- approvalWorkflowDN - the DN of an approval workflow to be started for approval of user creation.

The configuration section is followed by:

- Providing the handler, by calling scriptContext.getHandler().
- Reading the account from the handler.
- Supplying the user's parentID with a constant DN.
- Creating the user's cn from the account's cn.
- Initializing the handler with the configured user name and parent folder: handler.initialize(cnOfFUser, parentID).
- Calling the handler's prepareUser method. This method performs all copying actions defined in the configuration section. This call creates a complete functional user in memory with all attributes populated according to the variable definitions given the

configuration section.

- Reading the functional user from the handler, handler.getFunctionalUser().
- Calling the JavaScript "addAttributes" method. This method provides the option of creating user attribute values using the full flexibility of JavaScript. In the sample, the description is populated.
- Selecting whether to create the user with an approval workflow or directly in the Identity Store:
  - To start an approval workflow, use handler.startApprovalWorkflow(approvalWorkflowDN).
  - To create the user directly in the Identity Store, use handler.createUserInLdap().

#### 10.4.3.4. How the Approval Workflow Works

The JavaScript variable **approvalWorkflowDN** contains the DN of the sample workflow **Approve Create User from Account**.

This workflow is a simplified variant of the Create User workflows, with only one approval activity - a Calculate GUID activity - and an Apply Change activity.The workflow does not contain any Enter Attributes or Request Privileges activities, since this data is already provided by the consistency rule in the form of subject and resource orders that are passed to the workflow when it is started.The absence of activities for entering data and assignments makes this workflow unsuitable for creating a new user in Web Center.

You can tailor the **Approve Create User from Account** workflow to your needs; for example, you can change the set of attributes in the approval step, remove the calculate GUID activity or replace it with your own GUID generating activity.But note that the Apply Change activity uses a user hook called **siemens.dxr.service.order.impl.ModifyAccountUserHook**.This user hook is mandatory for re-linking the **dxrUserLink** to the user prior to saving (and resolving) the new user; otherwise the role resolution performed in the Apply Change step would create a new account for the new user if direct group assignments are created for the user.

## 10.5. Customizing the JavaScript for Persona and User Exchange

Web Center permits you to exchange a persona and its related user.This process is controlled by the JavaScript ExchangeUserPersona.js, which is located in the JavaScripts folder in the Customer Extensions configuration tree.

The JavaScript calls a handler implemented in Java that contains the program logic.You can tailor it to your needs by changing the following JavaScript configuration variables:

- **attributesToPreserve** - the list of attributes whose values do not change at the persona.The values are copied from the persona to the user before the object types are exchanged.
- **attributesToMove** - the list of attributes whose values do not change at the user.The values are copied from the user to the persona before the object types are exchanged.

The configuration section that defines these attributes is followed by script code that:

- Passes the attributesToPreserve to the handler
- Passes the attributesToMove to the handler
- Calls the handler.exchangeUserPersona()

The handler's exchangeUserPersona method exchanges user and persona in the following steps:

- Clears the owner and dxrUserUid at the persona.
- Adds the auxiliary object class dxrPersona to the user.
- Removes the auxiliary object class dxrPersona from the persona.
- Exchanges the object descriptors of user and persona.
- Copies the user attributes defined in attributesToMove from user to persona.
- Copies the preservedPersonaAttributes from persona to user.
- Set owner and dxrUserUid and owner at the user.
- Saves the user. In the Identity Store, it is a persona from now on.
- Saves the persona. In the Identity Store, it is a user from now on.

# 11. Customizing Password Management

This chapter describes how to customize the password checker and the password generator of the DirX Identity password management component and how enable account locking in the DirX Server so that DirX Identity can use it.

## 11.1. Customizing the Password Checker and Generator

Whenever a password of a user or an account is changed or reset, the password management component makes sure that the new password meets the password policy in force.Likewise, if the password manager is asked to auto-generate a new password, it returns a password that complies with the password policy.

You can customize the password checker in order to enforce additional restrictions on the new password.You can also replace the default checks with your own checks.And you can replace the default password generator with a custom generator.

You activate a custom checker or generator by assigning it to the relevant password policies.For example, you can assign the same custom components to all policies.Or you can assign different custom components to different policies.

### 11.1.1. Checking Passwords

The password manager performs a password check in two steps:

- History checks.
- Default checks concerning character classes, minimum and maximum length and Windows compatibility.

You can hook custom checks into this sequence at two different points, namely before and after the default checks:

- History check.
- Custom pre-checks.
- Default checks.
- Custom post-checks.

Custom checks may be performed in addition to the default ones, or replace them altogether.

Samples for custom checks are:

- A password may contain characters from restricted sets only, like
- ASCII characters only.
- Hyphen and dot as only non-alphanumeric characters.

- A password may not contain the same character more than once.
- A password may not start with a question mark.
- A password must be sufficiently different from the old one.
- A password is not the reverse of any password in the history.
- A password is not a round-robin shift of any password in the history.
- A password may not contain parts of a user's last name and first name.
- A password may not match any word from a dictionary.

### 11.1.2. Generating Passwords

To generate a random password, the password manager proceeds in the following steps:

- Generate a password. This is based on some hard-coded character sets for digits, lower case letters, upper case letters and non-alphanumeric characters.
- Check if the password complies with the password policy.

If the check succeeds, the password is accepted. If not, the sequence begins anew. After ten failed attempts to generate a password, the generation is aborted and throws an exception.

You can hook custom generators into the above sequence at the beginning:

- Custom generators.
- Default generator.
- Password check (including custom checks).

A hook may create a password itself, or it may just instruct the default generator to generate a password based on custom character sets instead of the default ones.

### 11.1.3. Restrictions

You cannot customize the default history check. It is always performed by the password management component according to the password policy in force. It assures that the new password doesn't match one of the previous n passwords.

You can perform additional checks against the password history in a custom checker. Note, however, that the history contains only hashes of the previous passwords. Therefore, it is not possible to get their cleartext values from the history.

### 11.1.4. Writing Custom Classes

This section provides information about writing custom classes.

#### 11.1.4.1. Customizing Password Checks

A custom checker is a Java class implementing the interface **net.atos.dirx.dxi.password.custom.Checker**. The interface consists of the two methods

**preCheck** and **postCheck**. The class must also provide a public no-args constructor.

A custom checker may do the real work in just one of them, or distribute it over both methods. You can also implement more than one custom checker class and hook them into the sequence. Custom checkers may perform checks in addition to the default ones, or replace the default checks altogether.

The password manager passes two arguments to **preCheck** and **postCheck**:

- The password to be checked.
- A context object providing access methods to all the information that might be necessary to check the password, like
- The old password.
- The password policy.
- The user or account DN.
- A password historian providing a method to perform checks against the password history.
- An object providing methods to integrate Windows compatibility checks into a custom checker.
- An LDAP connection to the provisioning database bound as DomainAdmin.

The context object is created before the check sequence starts, and then passed to all the methods in the chain. Changes to the context object by one checker will be visible to all subsequent ones.

A **preCheck** returns one of the following results:

- The password passed the check. Skip any further checks.
- The password passed the check. Continue with the next custom or default check if any.
- The password passed the check. Disable the default check. Continue with the next custom check if any.
- The password failed to pass the check. Skip any further checks.

A **postCheck** returns one of the following results:

- The password passed the check. Skip any further checks.
- The password passed the check. Continue with the next custom check if any.
- The password failed to pass the check. Skip any further checks.

The password is rejected if it fails to pass one of steps in the above sequence; in this case, the remaining steps are skipped. Otherwise, the password is accepted.

#### 11.1.4.2. Customizing Password Generation

A custom generator is a Java class implementing the interface **net.atos.dirx.dxi.password.custom.Generator**. The interface consists of the single method

**generate**. The class must also provide a public no-args constructor.

You can also implement more than one custom generator and hook them into the sequence.

The password manager passes a single argument to **generate**:

- A context object providing access methods to all the information that might be necessary to generate the password, like
- The password policy.
- The user or account DN.
- An object providing methods to integrate Windows compatibility checks into a custom checker.
- An LDAP connection to the provisioning database bound as DomainAdmin.

The context object is created before the generation sequence starts, and then passed to all the methods in the chain. Changes to the context object by one generator or checker will be visible to all subsequent ones.

A **generate** returns either a password candidate, or **null**. In the latter case, the next generator (custom or default) is invoked.

The default generator takes the characters for passwords from 4 different character sets:

- Digits - **123456789**
- Lower case letters - **abcdefghjkmnpqrstuvwxyz**
- Upper case letters - **ABCDEFGHJKMNPQRSTUVWXYZ**
- Non-alphanumeric characters - **!@#$%^&**()_+-=\{}|[]:";<>?,./*

You can replace one or more of the sets with custom ones. The context object provides interface methods to inject custom sets. To let the default generator generate a new password based on one or more custom sets, write a generator that injects the sets, but does not return a new password.

#### 11.1.4.3. Customizing Interfaces

The interfaces are provided by the package

**net.atos.dirx.dxi.password.custom**

The package is delivered in the Java archive **dxmPassword.jar**.

This section provides only a very brief overview. For details, refer to the Java documentation of the package.

#### 11.1.4.4. Checker

A class performing custom password checks must implement this interface with the methods **preCheck** and **postCheck**. The enumeration class **Checker.Result** defines the

codes to be returned by the methods.

#### 11.1.4.5. Generator

A class performing custom password generations must implement this interface with the single method **generate**.

#### 11.1.4.6. Context

The context object is passed to any of the customization methods. It provides access to the information that might be needed to perform the customization task.

#### 11.1.4.7. Policy

The policy object provides access to the password policy settings in force for a password check or password generation. The policy can be obtained via the context.

#### 11.1.4.8. Historian

The historian provides a method to check if a given password is contained in the password history.

#### 11.1.4.9. WindowsChecker

The windows checker object provides methods to integrate Windows compatibility checks into custom checkers. The object can be obtained via the context.

#### 11.1.4.10. Customizing Samples

The samples are provided by the packages

```
net.atos.dirx.dxi.password.custom.samples
net.atos.dirx.dxi.password.custom.samples.trace
```

The packages are delivered in the Java archive dxmPassword.jar.

This section gives only a very brief overview. For details, refer to the Java documentation and to the source code of the packages.

Each sample class implements either interface Checker or interface Generator. A single custom class, however, might quite as well implement both interfaces.

#### 11.1.4.11. SyntaxChecker

A pre-check making sure that a password is not empty and contains only ASCII letters (a-z, A-Z), ASCII digits (0-9) and the special characters dot (.), comma (,), hyphen (-), semicolon (;) and question mark (?).

#### 11.1.4.12. DiffChecker

A pre-check making sure that the password history doesn't contain the reverse of the new password, or any round-robin shift of the new password or the reverse thereof.

A post-check making sure that a new password does not contain three or more subsequent characters from the old one.

#### 11.1.4.13. CheckerTracer

Doesn't do any checks but just logs the parameters passed to its pre- and post-check. Old and new passwords are hashed. Note that the tracer costs resources and should not be enabled in a productive environment.

#### 11.1.4.14. CustomGenerator

Generates a random password from the character set imposed by the SyntaxChecker.

#### 11.1.4.15. CustomCharSetsGenerator

Customizes the set of non-alphanumeric characters for the default password generator. Does not generate a password itself.

#### 11.1.4.16. GeneratorTracer

Doesn't generate a password but just logs the parameters passed to the generate method. Old and new passwords are hashed. Note that the tracer costs resources and should not be enabled in a productive environment.

### 11.1.5. Compiling Custom Classes

To compile your custom code, add the Java archive **dxmPassword.jar** to your class path. If your code requires direct access to the provisioning directory via an LDAP connection, additionally add **ldapjdk.jar**.

You can find the archives in folders **lib/java** and **lib/java/ext** of your DirX Identity installation.

### 11.1.6. Integrating Custom Classes

Put your custom classes into a Java archive and make the archive available to any affected process, especially Web Center and the Java-based server.

#### 11.1.6.1. Web Center

Copy the archive to Web Center's **WEB-INF/lib** folder. Then restart Web Center.

#### 11.1.6.2. Java-based Server

Copy the archive to the **confdb/common/lib** subfolder of the Java-based Server's base folder and then restart the Java-based Server.

### 11.1.7. Registering Custom Classes

Register your custom classes for each affected password policy in the policy attribute **dxrPwdCustomizationClass**.

The attribute is multivalued. Each value contains a comma-separated list of full class names. The classes specified in a single value are invoked in the same order as listed there. The invocation order of classes specified in different values is undefined.

You can assign the custom classes to a policy via the Web Center user interface.

#### 11.1.7.1. Sample

We assign two different attribute values, one for the checkers and one for the generators:

```
…CheckerTracer,…SyntaxChecker,…DiffChecker
…GeneratorTracer,…CustomGenerator
```

We use the ellipsis (…) here to save space. In real life, you must replace them with the full package name, like

```
net.atos.dirx.dxi.password.custom.samples.CheckerTracer
```

The first checker invoked is the CheckerTracer, next follows the SyntaxChecker and finally the DiffChecker.

The first tracer invoked is the GeneratorTracer, and then comes the CustomGenerator.

### 11.1.8. Logging

The samples use the **java.util.logging** package for logging. Use the same package in your custom classes and enable logging for your classes, mutatis mutandis, as described below.

#### 11.1.8.1. Web Center

Enable logging for the custom packages within Web Center in Tomcat's **conf/logging.properties** file, for example:

```
net.atos.dirx.dxi.password.custom.level = FINE
net.atos.dirx.dxi.password.custom.handlers =
java.util.logging.ConsoleHandler
```

#### 11.1.8.2. Java-based Server

Enable logging for the custom packages within the Java-based server using the Java-based

server's Web Admin page "Set log levels".

- Package name: **net.atos.dirx.dxi.password.custom**
- Level: **FINE**

## 11.1.9. Adapting Web Center for Customized Password Policies

Whenever a user changes his password in Web Center or an administrator resets the password of another user, Web Center displays the password policy in force in order to let the user know the restrictions the new password must meet. The standard dialogs, however, do not show any additional restrictions imposed by your customized password policies. Therefore, you have to adapt them to your needs. This section give some hints for how this can be done.

We can classify customized password policies into three types:

- A single customized password policy for all users.
- Different password policies for different users but each policy is customized in the same way.
- Different password policies for different users and the policies are customized in different ways.

### 11.1.9.1. Renderer

We define a new renderer with name **customPasswordPolicyDescription**. The renderer will fit all customization types.

### 11.1.9.2. Definition

The renderer definition is simple:

```
<renderer id="customPasswordPolicyDescription"
 type="java.lang.String"
 defURL="/WEB-INF/custom/snippets/password/
         passwordPolicyDescription.htm"/>
```

### 11.1.9.3. HTML Snippet

The code for the snippet **passwordPolicyDescription.htm** is

```
<div class="wcpRule">${value}</div>
```

The expression is without any extension (like in ${value.h}) since the expression will be replaced with HTML code.

#### 11.1.9.4. Form Beans

For each customization type, we give a sample for how to integrate custom text into a password change form.

#### 11.1.9.5. A Single Customized Password Policy

The standard password policy display is replaced with a policy-independent custom text.

```
<form-bean name="resetPasswordForm" …>
   <form-property name="$displayName" …>
   <form-property-group name="userPasswordPolicyLabel" …>
   <form-property
       name="userPasswordPolicyDescription"
       type="java.lang.String"
       value="description"
       label="none"
       fieldRenderer="customPasswordPolicyDescription"
       messagePrefix="custom.passwordPolicy"
       y="+1" spanX="4" readonly="true"/>
   <form-property-group name="userPasswordEnterLabel" …>
        …
</form-bean>
```

Note that attributes **messagePrefix** and value define the message key **custom.passwordPolicy.description** whose text will replace the value expression in the renderer snippet.

#### 11.1.9.6. Different Password Policies Customized in the Same Way

The form displays the password policy in the usual way, but includes an additional property for the policy-independent custom text.

```
<form-bean name="changeMyPasswordForm" …>
   <form-property-group name="userPasswordPolicyLabel" …>
   <form-property name="userPasswordPolicy" …>
   …
   <form-property
        name="userPasswordPolicy.dxrpwdwindowscompatible">
   <form-property
       name="userPasswordPolicyDescription"
       type="java.lang.String"
       value="addendum"
```

```
        label="none"
        fieldRenderer="customPasswordPolicyDescription"
        messagePrefix="custom.passwordPolicy"
        y="+1" spanX="4" readonly="true"/>
    <form-property-group name="userPasswordEnterLabel" …>
         …
</form-bean>
```

Attributes **messagePrefix** and value define the message key **custom.passwordPolicy.addendum.addendum** whose text will replace the value expression in the renderer snippet.

#### 11.1.9.7. Different Password Policies Customized in Different Ways

The form displays the password policy in the usual way, but includes an additional property for the per policy custom text.

```
<form-bean name="setNewPasswordForm" …>
   <form-property-group name="userPasswordPolicyLabel" …>
   <form-property name="userPasswordPolicy" …>
   …
   <form-property
        name="userPasswordPolicy.dxrpwdwindowscompatible">
   <form-property
       name="userPasswordPolicy.cn"
       type="java.lang.String"
       label="none"
       fieldRenderer="customPasswordPolicyDescription"
       messagePrefix="custom.passwordPolicy.addendum"
       y="+1" spanX="4" readonly="true"/>
   <form-property-group name="userPasswordEnterLabel" …>
        …
</form-bean>
```

This time the form property references the common name of the password policy. Together with attribute **messagePrefix**, it defines the policy-dependent message key *custom.passwordPolicy.addendum.*_common name_ whose text will replace the value expression in the renderer snippet.

#### 11.1.9.8. Message Texts

The custom texts should be defined per supported language in custom message files like

**text_en.properties**, for example

```
# Policy-independent custom text

custom.passwordPolicy.description=\
<ul class="wcpList">\
<li>The passwort may include the following characters only:\
<ul class="wcpList">\
<li>Digits (0-9)</li>\
<li>Lower case letters (a-z)</li>\
<li>Upper case letters (A-Z)</li>\
<li>Dot (.), comma (,), hyphen (-), semicolon (;) and question mark
(?)</li>\
</ul>\
</li>\
<li>The password must contain at least 6 and at most 10
characters.</li>\
<li>The new password must not contain three or more subsequent
characters from the old one.</li>\
</ul>

# Policy-independent custom addendum

custom.passwordPolicy.addendum=\
The new password must not contain three or more subsequent characters
from the old one.

# Per policy custom addenda

custom.passwordPolicy.addendum.default=\
The new default password must not contain three or more subsequent
characters from the old one.

custom.passwordPolicy.addendum.critical\ areas=\
The new critical password must not contain three or more subsequent
characters from the old one.

custom.passwordPolicy.addendum.services=\
The new service password must not contain three or more subsequent
characters from the old one.
```

The addendum keys contain the common names in lower cases and that spaces in the keys must be escaped with backslashes.

### 11.1.10. Documentation and Sample Source Code

You can find the Java documentation of the customization interfaces and samples on the DirX Identity DVD in the folder:

**Documentation/DirXIdentity/PasswordManagement**

The sample source code is delivered in the folder:

**Additions/PasswordManagement**.

Web Center adaptation samples can be found in the folder:

**Additions/WebCenter/CustomPasswordPolicies**

## 11.2. Configuring the Global Password Policy for Access Locking

The DirX Directory Server offers the option to lock accounts via its Global Password Policy subentry settings.If the DirX Server's Global Password Policy subentry is configured to enable account locking and the DirX Identity domain-wide flags **Lock Disabled Users** or **Lock Tbdel Users** are checked, DirX Identity sets the Password Account Locked Time operational attribute to **19700101000000Z** (GMT format) to block users who are disabled (state DISABLED) and/or "to be deleted" (state TBDEL) from access via their accounts.The DirX Server performs the lock, so it affects all LDAP clients.

Use the **dirxadm** command line to configure the Global Password Policy subentry to enable account locking.The following example displays the settings required to enable it:

```
show /cn=globalpasswordpolicy -alla -p
 1) /CN=GlobalPasswordPolicy
    DSE-Type                        : SUBENTRY
    Object-Class                    : PPO
                                    : SUBE
                                    : TOP
    Common-Name                     : GlobalPasswordPolicy
    Modification-Time               : 20130417055628.731Z
    pwdPol-pwdMinAge                : 0
    pwdPol-pwdMaxAge                : 0
    pwdPol-pwdInHistory             : 0
    pwdPol-pwdCheckSyntax           : 0
    pwdPol-pwdMinLength             : 0
```

```
    pwdPol-pwdMinSpecialChar       : 0
    pwdPol-pwdMaxLength            : 0
    pwdPol-pwdExpireWarning        : 0
    pwdPol-pwdGraceLoginLimit      : 0
    pwdPol-pwdLockout              : TRUE
    pwdPol-pwdLockoutDuration      : 0
    pwdPol-pwdMaxFailure           : 1000
    pwdPol-pwdFailureCountInterval : 1
    pwdPol-pwdMustChange           : FALSE
    pwdPol-pwdStorageScheme        : SSHA
    pwdPol-pwdStorageSchemeLevel   : 2
    dirx-entry-uuid                : 8706f8a4-22db-4bd2-…
```

Use the following **dirxadm** commands to change the pwdPol-pwdLockout (PPLOCK), pwdPol-pwdMaxFailure (PPMAXF) and pwdPol-pwdFailureCountInterval (PPFCIN) subentry attributes to the values required to enable account locking:

```
modify /cn=globalpasswordpolicy -removeattr PPLOCK=FALSE
-addattr PPLOCK=TRUE
modify /cn=globalpasswordpolicy -removeattr PPMAXF=0 -addattr
PPMAXF=1000
modify /cn=globalpasswordpolicy -removeattr PPFCIN=0 -addattr
PPFCIN=1
```

See the *DirX Administration Reference* for details on **dirxadm** and Global Password Policy attributes. See the context-sensitive help for the domain object for details on domain-wide properties and controls.

# 12. Customizing Certification Campaigns

This chapter describes how to customize certification campaigns with user hooks.

## 12.1. Customizing Certification Campaigns with User Hooks

DirX Identity allows you to extend certification campaigns with user hooks.You can use the following user hooks:

- **Find Approvers** user hook.Use this hook to change the default approvers for certifications.
- **Find Subjects** user hook.Use this hook to select the subjects (for a user certification, the users) of a campaign.
- **Limit Resources** user hook.Use this hook to reduce the list of assignments to be certified.
- **Send Email** user hook.Use this hook to send notifications your own way.
- **Campaign Creator** user hook.Use this hook to completely override the creation of a campaign: find the subjects, select the assignments and calculate the approvers.

The Use Case Document *Certification Campaigns* explains how to set up and run certification campaigns.

# Appendix A: Deprecated Features

This chapter describes features in DirX Identity that are obsolete and will not be supported in future DirX Identity releases.

## A.1. Customizing Property Page Descriptions

A DirX Identity provisioning object description describes an object and its properties.In previous releases, DirX Identity used property page descriptions to implement highly flexible object presentations.In the latest release, DirX Identity's generic property pages mechanism has been enhanced.As that result, you should use this feature instead.It solves most of the flexibility requirements and is much easier to configure.The description of the Beans Markup Language (BML) feature is only provided here for compatibility reasons and will be removed from future versions of this document.

Property page descriptions are written in Beans Markup Language (BML) and interpreted at runtime by a BML player.Property page descriptions allow you to customize a Manager "form" (tabs and their fields) by modifying the form's property page description (its BML file); the BML player interprets the file and generates the corresponding form in DirX Identity Manager.

The specification of a GUI layout using BML requires the knowledge of a Java GUI designer.Therefore, we recommend that you modify the existing property page descriptions to add or remove a few properties, but that you do not change the overall layout or the style.Experienced users can use IDEs like Eclipse IDE for Java Developers or Netbeans' Java IDE to draw the forms and store them in a type of XML format.They must then transform this format to the XML format accepted by DirX Identity.

The elements (beans) that appear in a property page are described in hierarchical (nested) order: the description starts with the underlying panel, continues with the contained beans, and if there is a compound bean, it is described before the contained beans are listed.

For each bean, the property page description defines

- The beans class
- The beans properties, mainly the layout manager but also other parameters to be set prior to displaying
- The layout properties for the bean

These items are necessary to define and use property pages. For example, suppose you want to display new properties of the user object. To perform this task:

1. Ensure that the directory schema already supports the new properties. If not, change the directory schema before you update the XML object description and the BML property page.
2. Update the object description and add the new attributes.
3. Modify the property page description by entering a bean for each of the new attributes.

The subsequent topics describe these steps in detail using the user object as an example.

### A.1.1. Locating the Property Page Descriptions

Like the object descriptions, the property page descriptions are stored in the DirX Identity store. Thus, modifications made to these descriptions are simultaneously available to all distributed DirX Identity Manager clients. The property page descriptions are stored in a separate folder of each domain below the configuration container, next to the object descriptions:

**cn=Property Page Descriptions,cn=Configuration,***RDN of domain root*

Descriptions for target system groups and accounts are stored below the target system folder and thus may be specified differently for each target system:

**cn=Property Page Descriptions,cn=Configuration,***DN of target system*

### A.1.2. Selecting the Property Page to Modify

To select the appropriate property page description, you must inspect the object description of the object in question. In our example we want to modify the user object. Therefore we take a look at the object description **user.xml.**

The XML element **propertysheet** contains a list of all property pages (tabs). Each page is described in one XML element **propertypage**. This element specifies the implementing class, the title to be displayed at the top of the page and a reference to the property page description in its **layout** attribute.

Take a look at an excerpt of the user's object description. It lists property pages named "UserGeneral", "UserAttributes", "UserRoles", "UserGroups" and "UserAccounts":

```
<propertysheet>
     <propertypage name="UserGeneral"

class="siemens.DirXjdiscover.api.nodes.customizer.BMLNodePropertyPage
"
           title="General"

layout="storage://DirXmetaRole/cn=UserGeneral.bml,cn=Property Page
Descriptions,
     cn=Configuration,$(rootDN)?content=dxrObjDesc"
           />
     <propertypage name="UserAttributes"

class="siemens.DirXjdiscover.api.nodes.customizer.BMLNodePropertyPage
"
```

```
           title="General"

layout="storage://DirXmetRole/cn=UserAttributes.bml,cn=Property Page
Descriptions,
    cn=Configuration,$(rootDN)?content=dxrObjDesc"
          />
     <propertypage name="UserRoles"
           ...
          />
     <propertypage name="UserGroups"
           ...
          />
     <propertypage name="UserAccounts"
           ...
          />
</propertysheet>
```

Do not change the **class** attribute. It directs DirX Identity to display the page using the BML player.

The **layout** attribute holds the reference to the property page description. Its structure is:

**storage://DirXmetaRole/***DN of property page description*

The **DirXmetaRole** prefix directs DirX Identity to search for the description in the DirX Identity store. The DN string:

"cn=UserAttributes.bml,cn=Property Page Descriptions,
cn=Configuration,$(rootDN)?content=dxrObjDesc"

contains the variable "**$(rootDN)**". At runtime, it is replaced by the context prefix of the domain root.

The appropriate place to enter custom properties is the second page named "UserAttributes". The **layout** attribute tells us that we need to edit the property page description **UserAttributes.bml**.

### A.1.3. Specifying the Properties in the Object Description

Before you edit the property page description, you must ensure that the properties are defined in the object description. In our example, we want to display the user properties "postalCode" and "roomNumber". As a result, we need to provide property definitions for them in the user object description as follows:

```
<properties>
```

```
    <property name="..." .../>
    <property name="postalCode" label="Postal Code"
type="java.lang.String"/>
    <property name="roomNumber" label="Room Number"
type="java.lang.String"/>
</properties>
```

Both properties are of type "String" and do not have any default values.

### A.1.4. Adding Attributes to a Property Page Description

To modify the description with DirX Identity Manager, navigate to the domain view and select **UserAttributes.bml** in the Property Page Descriptions folder.

Before the modification, the form may look like this:

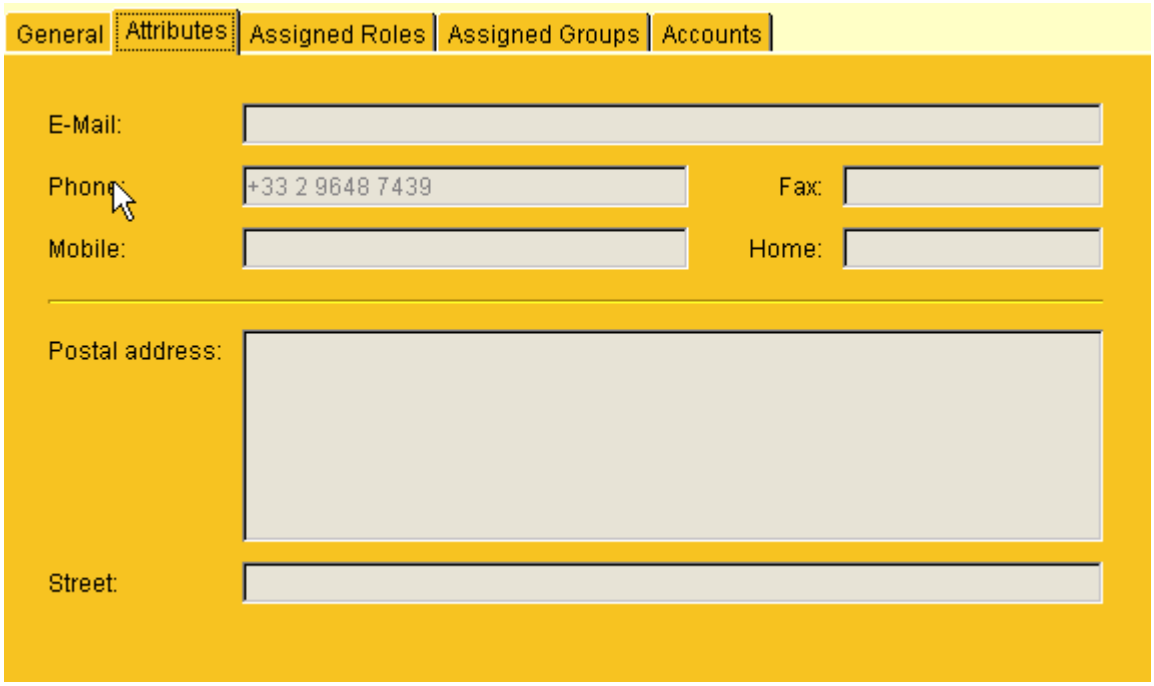



*Figure 13. Form before Modification*

The form uses the GridBag layout to place the labels and fields into four columns and as many rows as needed. The first and third columns are used for labels, and the second and fourth columns are used for the field values. We want to place our properties into the row above the last row, just above the "Street" property.

We want to display the properties "postalCode" and "roomNumber" in one row just above the "Street" property: "postalCode" on the left, the "roomNumber" on the right side.

#### A.1.4.1. Adding the Postal Code

First, we must enter the label. In the **UserAttributes.bml** property page description file, we search for the label for the "Street" property. It is placed in the description of the first

column below the "Postal address":

```
<!-- Postal address label -->
<add>
     <bean class="javax.swing.JLabel">
          <property name="text" value="Postal address:"/>
     </bean>
     <bean source="cons">
          <field name="gridx" value="0"/>
          <field name="gridy" value="4"/>
     </bean>
</add>
<!-- Street label -->
<add>
     <bean class="javax.swing.JLabel">
          <property name="text" value="Street:"/>
     </bean>
     <bean source="cons">
          <field name="gridx" value="0"/>
          <field name="gridy" value="5"/>
     </bean>
</add>
```

Let’s add the label for the "Postal code". The result looks like this:

```
<!-- Postal address label -->
<add>
     <bean class="javax.swing.JLabel">
          <property name="text" value="Postal address:"/>
     </bean>
     <bean source="cons">
          <field name="gridx" value="0"/>
          <field name="gridy" value="4"/>
     </bean>
</add>
<!-- Postal code label -->
<add>
     <bean class="javax.swing.JLabel">
          <property name="text" value="Postal code:"/>
     </bean>
```

```
          <bean source="cons">
               <field name="gridx" value="0"/>
               <field name="gridy" value="5"/>
          </bean>
     </add>
     <!-- Street label -->
     <add>
          <bean class="javax.swing.JLabel">
               <property name="text" value="Street:"/>
          </bean>
          <bean source="cons">
               <field name="gridx" value="0"/>
               <field name="gridy" value="6"/>
          </bean>
     </add>
```

The label is placed in column 0 (gridx: 0) and row 5 (gridy: 5). The "Street" label is moved to row 6. The label value may be chosen independently from the XML object description.

To enter the field for the "postalCode" value, we search for the "street" field in the second column and add the necessary XML element resulting in this:

```
     <!-- Postal address field -->
     <add>
          <bean
class="siemens.dxr.manager.controls.MetaRoleJnbGeneric" id="addr">
               <property name="nodePropertyName"
value="postalAddress"/>
          </bean>
          <bean source="cons">
               <field name="gridx" value="1"/>
               <field name="gridy" value="4"/>
               <field name="gridwidth" value="4"/>
          </bean>
     </add>
     <!-- Postal code field -->
     <add>
          <bean
class="siemens.dxr.manager.controls.MetaRoleJnbGeneric">
               <property name="nodePropertyName" value="postalCode"/>
          </bean>
```

```
          <bean source="cons">
               <field name="gridx" value="1"/>
               <field name="gridy" value="5"/>
               <field name="gridwidth" value="1"/>
          </bean>
     </add>
     <!-- Street field -->
     <add>
          <bean
class="siemens.dxr.manager.controls.MetaRoleJnbGeneric">
               <property name="nodePropertyName" value="street"/>
          </bean>
          <bean source="cons">
               <field name="gridx" value="1"/>
               <field name="gridy" value="6"/>
               <field name="gridwidth" value="3"/>
          </bean>
     </add>
```

The property name must match the one specified in the object description: "postalCode". The field is placed in column 1, row 5 with a width of 1 (gridwidth: 1). If we prefer the field covering the rest of the row, we could define a gridwidth of 3 (as for the "street"). Do not change the other values.

We have finished the first part and get the following intermediate result (after restarting DirX Identity Manager):

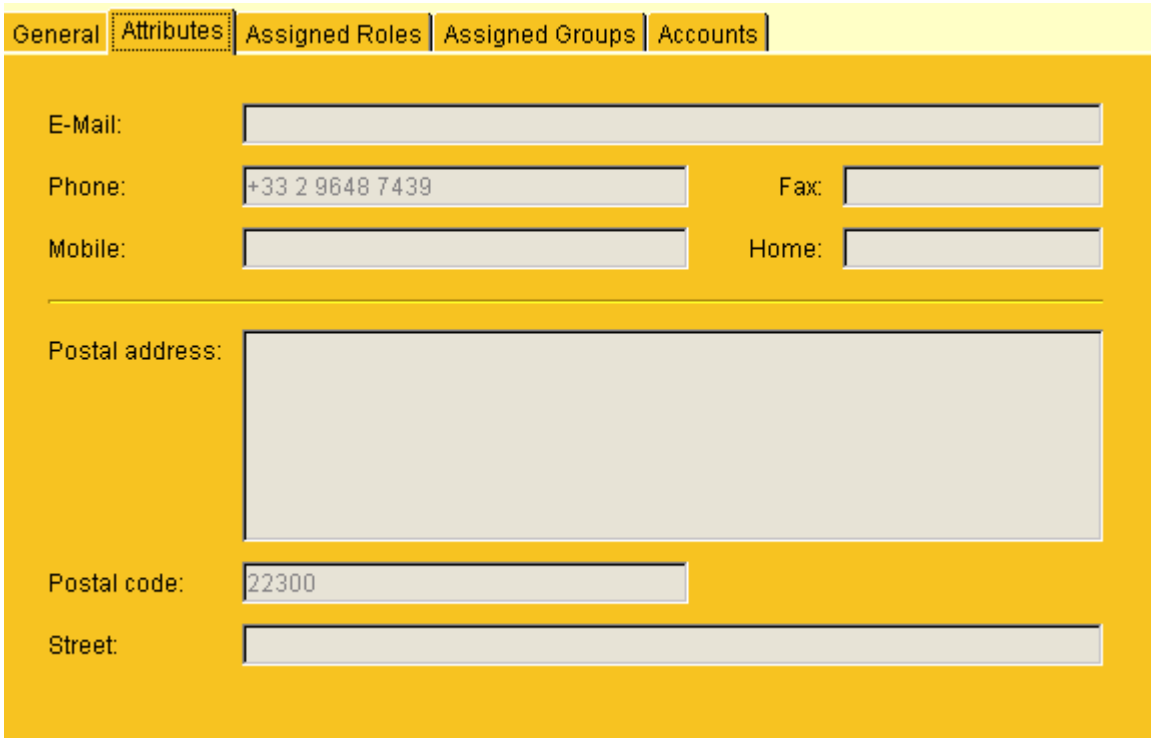



*Figure 14. Form during Modification*

#### A.1.4.2. Adding the Room Number

Let's make the same type of modifications for the "roomNumber" property: first the label in the third column:

```
     <!-- Room number label -->
     <add>
          <bean class="javax.swing.JLabel">
               <property name="text" value="Room:"/>
          </bean>
          <bean source="cons">
               <field name="gridx" value="2"/>
               <field name="gridy" value="5"/>
          </bean>
     </add>
```

and the value in the fourth column ...

```
     <!-- Room number field -->
     <add>
          <bean
class="siemens.dxr.manager.controls.MetaRoleJnbGeneric">
               <property name="nodePropertyName" value="roomNumber"/>
```

```
            </bean>
            <bean source="cons">
                 <field name="gridx" value="3"/>
                 <field name="gridy" value="5"/>
            </bean>
       </add>
```

Now we have the desired result:

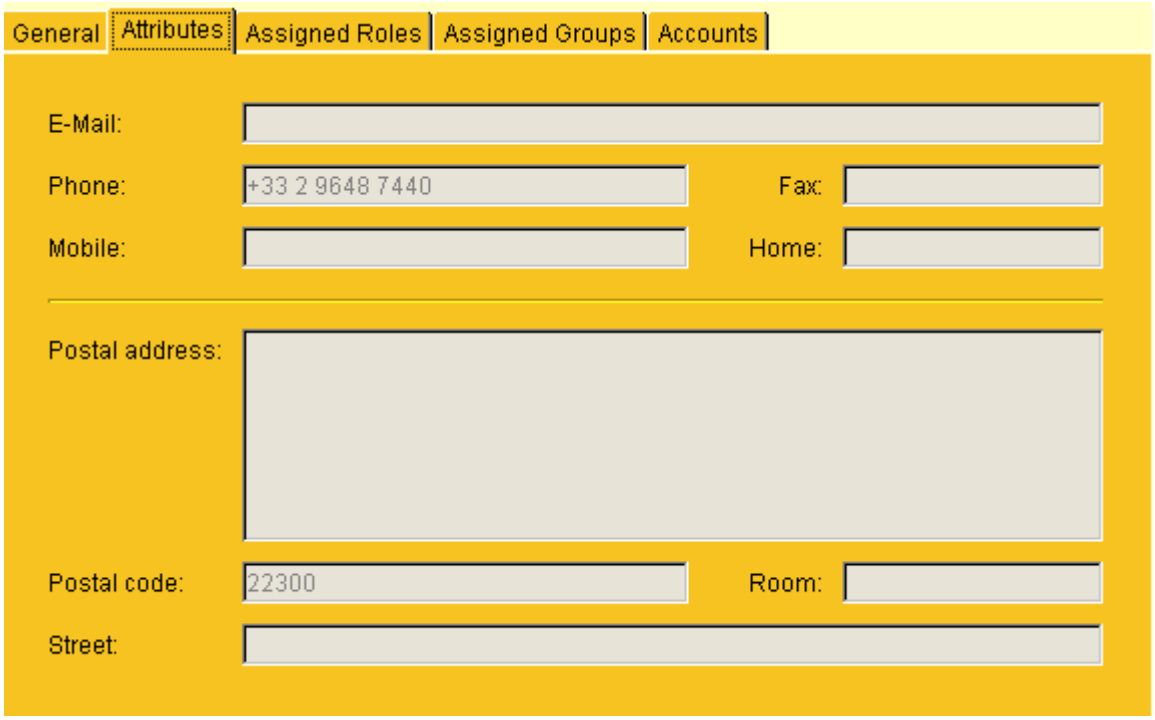



*Figure 15. Final Modified Form*

# DirX Product Suite

The DirX product suite provides the basis for fully integrated identity and access management; it includes the following products, which can be ordered separately.

## DirX Identity

DirX Identity provides a comprehensive, process-driven, customizable, cloud-enabled, scalable, and highly available identity management solution for businesses and organizations. It provides overarching, risk-based identity and access governance functionality seamlessly integrated with automated provisioning. Functionality includes lifecycle management for users and roles, cross-platform and rule-based real-time provisioning, web-based self-service functions for users, delegated administration, request workflows, access certification, password management, metadirectory as well as auditing and reporting functionality.

## DirX Directory

DirX Directory provides a standards-compliant, high-performance, highly available, highly reliable, highly scalable, and secure LDAP and X.500 Directory Server and LDAP Proxy with very high linear scalability. DirX Directory can serve as an identity store for employees, customers, partners, subscribers, and other IoT entities. It can also serve as a provisioning, access management and metadirectory repository, to provide a single point of access to the information within disparate and heterogeneous directories available in an enterprise network or cloud environment for user management and provisioning.

## DirX Access

DirX Access is a comprehensive, cloud-ready, scalable, and highly available access management solution providing policy- and risk-based authentication, authorization based on XACML and federation for Web applications and services. DirX Access delivers single sign-on, versatile authentication including FIDO, identity federation based on SAML, OAuth and OpenID Connect, just-in-time provisioning, entitlement management and policy enforcement for applications and services in the cloud or on-premises.

## DirX Audit

DirX Audit provides auditors, security compliance officers and audit administrators with analytical insight and transparency for identity and access. Based on historical identity data and recorded events from the identity and access management processes, DirX Audit allows answering the “what, when, where, who and why” questions of user access and entitlements. DirX Audit features historical views and reports on identity data, a graphical dashboard with drill-down into individual events, an analysis view for filtering, evaluating, correlating, and reviewing of identity-related events and job management for report generation.

For more information: support.dirx.solutions/about

Legal remarks

On the account of certain regional limitations of sales rights and service availability, we cannot guarantee that all products included in this document are available through the Eviden sales organization worldwide. Availability and packaging may vary by country and is subject to change without prior notice. Some/All of the features and products described herein may not be available locally. The information in this document contains general technical descriptions of specifications and options as well as standard and optional features which do not always have to be present in individual cases. Eviden reserves the right to modify the design, packaging, specifications and options described herein without prior notice. Please contact your local Eviden sales representative for the most current information. Note: Any technical data contained in this document may vary within defined tolerances. Original images always lose a certain amount of detail when reproduced.