



# Object Modeling Guide

/ ForgeRock Identity Management 7

Latest update: 7.0.4

ForgeRock AS.  
201 Mission St., Suite 2900  
San Francisco, CA 94105, USA  
+1 415-599-1100 (US)  
[www.forgerock.com](http://www.forgerock.com)

---

Copyright © 2011-2020 ForgeRock AS.

## Abstract

### Guide to creating and managing objects in ForgeRock® Identity Management.



This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.

To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-nd/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

ForgeRock® and ForgeRock Identity Platform™ are trademarks of ForgeRock Inc. or its subsidiaries in the U.S. and in other countries. Trademarks are the property of their respective owners.

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

DejaVu Fonts

Bitstream Vera Fonts Copyright

Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

Arev Fonts Copyright

Copyright (c) 2006 by Tavmjong Bah. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the modifications to the Bitstream Vera Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Tavmjong Bah" or the word "Arev".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Tavmjong Bah Arev" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL TAVMJONG BAH BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the name of Tavmjong Bah shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from Tavmjong Bah. For further information, contact: tavmjong @ free . fr.

FontAwesome Copyright

Copyright (c) 2017 by Dave Gandy, <https://fontawesome.com/>.

This Font Software is licensed under the SIL Open Font License, Version 1.1. See <https://opensource.org/licenses/OFL-1.1>.

# Table of Contents







|  |     |
|--|-----|
| Overview .....   | v   |
| 1. Managed Objects .....                                   | 1   |
| Define the Schema .....                                    | 1   |
| Create and Modify Object Types .....                       | 2   |
| Managed Users .....  | 5   |
| Managed Groups .....                                       | 10  |
| Virtual Properties .....                                   | 11  |
| Run Scripts on Managed Objects .....                       | 13  |
| Track User Metadata .....                                  | 13  |
| 2. Relationships Between Objects .....                     | 17  |
| Define a Relationship Type .....                           | 17  |
| Create a Relationship Between Two Objects .....            | 19  |
| Configure Relationship Change Notification .....           | 21  |
| Validate Relationships Between Objects .....               | 25  |
| Create Bidirectional Relationships .....                   | 26  |
| Grant Relationships Conditionally .....                    | 27  |
| View Relationships Over REST .....                         | 28  |
| View Relationships in Graph Form .....                     | 32  |
| Manage Relationships Through the Admin UI .....            | 33  |
| 3. Roles .....   | 45  |
| IDM Role Types .....                                       | 45  |
| Managed Roles .....  | 46  |
| Manipulate Roles Over REST and in the UI .....             | 47  |
| Use Temporal Constraints to Restrict Effective Roles ..... | 59  |
| Use Assignments to Provision Users .....                   | 63  |
| Effective Roles and Effective Assignments .....            | 69  |
| Roles and Relationship Change Notification .....           | 71  |
| Managed Role Script Hooks .....                            | 72  |
| Use Groups to Control Access to IDM .....                  | 73  |
| 4. Use Policies to Validate Data .....                     | 75  |
| Default Policy for Managed Objects .....                   | 75  |
| Extend the Policy Service .....                            | 84  |
| Disable Policy Enforcement .....                           | 87  |
| Manage Policies Over REST .....                            | 88  |
| 5. Store Managed Objects in the Repository .....           | 94  |
| Repository Configuration Files .....                       | 94  |
| Generic and Explicit Object Mappings .....                 | 102 |
| 6. Access Data Objects .....                               | 119 |
| Access Data Objects By Using Scripts .....                 | 119 |
| Access Data Objects By Using the REST API .....            | 120 |
| Define and Call Data Queries .....                         | 120 |
| Upload Files to the Server .....                           | 139 |
| 7. Import Bulk Data .....                                  | 143 |
| A. Data Models and Objects Reference .....                 | 149 |

|                             |     |
|-----------------------------|-----|
| Managed Objects .....       | 150 |
| Configuration Objects ..... | 166 |
| System Objects .....        | 169 |
| Audit Objects .....         | 169 |
| Links .....                 | 169 |
| IDM Glossary .....          | 170 |

# Overview

IDM provides a default schema for typical managed object types, such as users and roles, but does not control the structure of objects that you store in the repository. In this guide, you will learn how to change and add to the managed object schema, how to establish relationships between objects, and how to use policies to validate objects. You will also learn how to access IDM objects using queries.

## Quick Start

|  |  |  |
|--|--|--|
| <br>Managed Objects<br>Learn about the IDM architecture, component modules, and services. | <br>Relationships<br>Configure relationships between object types.                              | <br>Roles<br>Learn about the role object—a specific relationship type.                |
| <br>Policies<br>Apply validation requirements to objects and properties.                  | <br>Store Objects<br>Configure your IDM repository and map objects to tables in the repository. | <br>Access Objects<br>Access data objects over REST, with scripts, and using queries. |

ForgeRock Identity Platform™ serves as the basis for our simple and comprehensive Identity and Access Management solution. We help our customers deepen their relationships with their customers, and improve the productivity and connectivity of their employees and partners. For more information about ForgeRock and about the platform, see <https://www.forgerock.com>.

The ForgeRock Common REST API works across the platform to provide common ways to access web resources and collections of resources.

## Chapter 1

# Managed Objects

IDM provides a default schema for typical managed object types, such as users and roles, but does not control the structure of objects that you store in the repository. You can modify or extend the schema for the default object types. You can also create new managed object types for any item that can be collected in a data set. For example, with the right schema, you can set up any device associated with the Internet of Things (IoT).

These topics describe how to work with the default managed object types and how to create new object types. For more information about the IDM object model, see "[Data Models and Objects Reference](#)".

- "Define the Schema"
- "Create and Modify Object Types"
- "Managed Users"
- "Managed Groups"
- "Virtual Properties"
- "Run Scripts on Managed Objects"
- "Track User Metadata"

## Define the Schema

Managed objects and their properties are defined in the `conf/managed.json` file. The schema in this file is not a comprehensive list of all the properties that can be stored in the IDM repository. If you use a generic object mapping, you can create a managed object with any arbitrary property, and that property will be stored in the repository. However, if you create an object with properties that are not defined in `conf/managed.json`, those properties are not visible in the UI. In addition, you won't be able to configure the "sub-properties" that are described in the following section.

For explicit object mappings, the schema must be mapped to tables and columns in the JDBC database or to organizational units in DS. For more information about explicit and generic object mappings, see "[Generic and Explicit Object Mappings](#)".

**Important**

The Admin UI depends on the presence of specific core schema elements, such as users, roles, and assignments (and the default properties nested within them). If you remove such schema elements, and you use the Admin UI to configure IDM, you must modify the Admin UI code accordingly. For example, if you remove the entire `assignment` object from `conf/managed.json`, the UI will throw exceptions wherever it queries this schema element.

## Create and Modify Object Types

If the managed object types provided in the default configuration are not sufficient for your deployment, you can create new managed object types. The easiest way to create a new managed object type is to use the Admin UI, as follows:

1. Select Configure > Managed Objects > New Managed Object.
2. On the New Managed Object page, enter a name and readable title for the object, make optional changes, as necessary, and click Save. The readable title specifies what the object will be called in the UI.
3. On the Properties tab, specify the schema for the object type (the properties that make up the object).
4. On the Scripts tab, specify any scripts that will be applied on events associated with that object type. For example, scripts that will be run when an object of that type is created, updated, or deleted.

You can also create a new managed object type by adding its configuration to the `conf/managed.json` file.

+ *Example: 'Phone' object created through the UI*

```
{
  "name": "Phone",
  "schema": {
    "$schema": "http://forgerock.org/json-schema#",
    "type": "object",
    "properties": {
      "brand": {
        "description": "The supplier of the mobile phone",
        "title": "Brand",
        "viewable": true,
        "searchable": true,
        "userEditable": false,
        "policies": [],
        "returnByDefault": false,
        "pattern": "",
        "isVirtual": false,
        "type": [
          "string",
          "null"
        ]
      }
    }
  }
}
```

```

    ],
    "assetNumber": {
      "description": "The asset tag number of the mobile device",
      "title": "Asset Number",
      "viewable": true,
      "searchable": true,
      "userEditable": false,
      "policies": [],
      "returnByDefault": false,
      "pattern": "",
      "isVirtual": false,
      "type": "string"
    },
    "model": {
      "description": "The model number of the mobile device, such as 6 plus, Galaxy S4",
      "title": "Model",
      "viewable": true,
      "searchable": false,
      "userEditable": false,
      "policies": [],
      "returnByDefault": false,
      "pattern": "",
      "isVirtual": false,
      "type": "string"
    }
  },
  "required": [],
  "order": [
    "brand",
    "assetNumber",
    "model"
  ]
}

```

Every managed object type has a **name** and a **schema** that describes the properties associated with that object. The **name** can only include the characters **a-z**, **A-Z**, **0-9**, and **\_** (underscore). You can add any arbitrary properties to the schema.

#### Tip

Avoid using the dash character in property names (like **last-name**) because dashes in names make JavaScript syntax more complex. Rather use "camel case" (**lastName**). If you cannot avoid dash characters, write **source['last-name']** instead of **source.last-name** in your JavaScript.

A property definition typically includes the following fields:

#### **title**

The name of the property, in human-readable language, used to display the property in the UI.

#### **description**

A brief description of the property.

**viewable**

Specifies whether this property is viewable in the object's profile in the UI. Boolean, **true** or **false** (**true** by default).

**searchable**

Specifies whether this property can be searched in the UI. A searchable property is visible within the Managed Object data grid in the End User UI. Note that for a property to be searchable in the UI, it *must be indexed* in the repository configuration. For information on indexing properties in a repository, see "Generic and Explicit Object Mappings".

Boolean, **true** or **false** (**false** by default).

**userEditable**

Specifies whether users can edit the property value in the UI. This property applies in the context of the End User UI, where users are able to edit certain properties of their own accounts. Boolean, **true** or **false** (**false** by default).

**isProtected**

Specifies whether reauthentication is required if the value of this property changes.

For certain properties, such as passwords, changing the value of the property should force an end user to reauthenticate. These properties are referred to as *protected properties*. Depending on how the user authenticates (which authentication module is used), the list of protected properties is added to the user's security context. For example, if a user logs in with the login and password of their managed user entry (**MANAGED\_USER** authentication module), their security context will include this list of protected properties. The list of protected properties is not included in the security context if the user logs in with a module that does not support reauthentication (such as through a social identity provider).

**pattern**

Any specific pattern to which the value of the property must adhere. For example, a property whose value is a date might require a specific date format.

**policies**

Any policy validation that must be applied to the property. For more information on managed object policies, see "Default Policy for Managed Objects".

**required**

Specifies whether the property must be supplied when an object of this type is created. Boolean, **true** or **false**.

**Important**

The **required** policy is assessed only during object creation, not when an object is updated. You can effectively bypass the policy by updating the object and supplying an empty value for that property.

To prevent this inconsistency, set both `required` and `notEmpty` to `true` for required properties. This configuration indicates that the property must exist, and must have a value.

### type

The data type for the property value; can be `string`, `array`, `boolean`, `integer`, `number`, `object`, `Resource Collection`, or `null`.

#### Note

If any user might not have a value for a specific property (such as a `telephoneNumber`), you must include `null` as one of the property `types`. You can set a null property type in the Admin UI (Configure > Managed Objects > User, select the property, and under the Details tab, Advanced Options, set `Nullable` to `true`).

You can also set a null property type directly in your `managed.json` file by setting `"type" : '[ "string", "null" ]'` for that property (where `string` can be any other valid property type. This information is validated by the `policy.js` script, as described in "Validate Managed Object Data Types".

If you're configuring a data `type` of `array` through the Admin UI, you're limited to two values.

### isVirtual

Specifies whether the property takes a static value, or whether its value is calculated "on the fly" as the result of a script. Boolean, `true` or `false`.

### returnByDefault

For non-core attributes (virtual attributes and relationship fields), specifies whether the property will be returned in the results of a query on an object of this type *if it is not explicitly requested*. Virtual attributes and relationship fields are not returned by default. Boolean, `true` or `false`. When the property is in an array within a relationship, always set to `false`.

### relationshipGrantTemporalConstraintsEnforced

For attributes with relationship fields. Specifies whether this relationship should have temporal constraints enforced. Boolean, `true` or `false`. For more information about temporal constraints, see "Use Temporal Constraints to Restrict Effective Roles".

## Managed Users

User objects that are stored in the repository are referred to as *managed users*. For a JDBC repository, IDM stores managed users in the `managedobjects` table. A second table, `managedobjectproperties`, serves as the index table.

IDM provides RESTful access to managed users, at the context path `/openidm/managed/user`. For more information, see "REST Interface Introduction" in the *Installation Guide*.

You can add, change, and delete managed users by using the Admin UI or over the REST interface. To use the Admin UI, select Manage > User. The UI is intuitive as regards user management.

If you are viewing users through the Admin UI, the User List page supports specialized filtering with the Advanced Filter option. This lets you build many of the queries shown in "Define and Call Data Queries".

The following examples show how to add, change, and delete users over the REST interface. For a reference of all managed user endpoints and actions, see "Managed Users" in the *REST API Reference*. You can also use the "*REST API Explorer*" in the *REST API Reference* as a reference to the managed object REST API.

#### Note

Some of the examples in this documentation use client-assigned IDs (such as `bjensen` and `scarter`) when creating objects because it makes the examples easier to read. If you create objects using the Admin UI, they are created with server-assigned IDs (such as `55ef0a75-f261-47e9-a72b-f5c61c32d339`). Generally, immutable server-assigned UUIDs are used in production environments.

#### + Retrieve the IDs of all managed users in the repository

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user?_queryFilter=true&_fields=_id"
{
  "result": [
    {
      "_id": "bjensen",
      "_rev": "0000000079b78ace"
    },
    {
      "_id": "scarter",
      "_rev": "0000000070e587a7"
    },
    ...
  ],
  ...
}
```

#### + Query managed users for a specific user

The `_queryFilter` requires double quotes, or the URL-encoded equivalent (`%22`), around the search term. This example uses the URL-encoded equivalent:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user?_queryFilter=userName+eq+%22scarter%22"
{
  "result": [
    {
      "_id": "scarter",
      "_rev": "0000000070e587a7",
      "userName": "scarter",
      "givenName": "Sam",
      "sn": "Carter",
      "telephoneNumber": "12345678",
      "active": "true",
      "mail": "scarter@example.com",
      "accountStatus": "active",
      "effectiveAssignments": [],
      "effectiveRoles": []
    }
  ],
  ...
}
```

This example uses single quotes around the URL to avoid conflicts with the double quotes around the search term:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=userName+eq+"scarter"'
{
  "result": [
    {
      "_id": "scarter",
      "_rev": "0000000070e587a7",
      "userName": "scarter",
      "givenName": "Sam",
      "sn": "Carter",
      "telephoneNumber": "12345678",
      "active": "true",
      "mail": "scarter@example.com",
      "accountStatus": "active",
      "effectiveAssignments": [],
      "effectiveRoles": []
    }
  ],
  ...
}
```

+ *Retrieve a managed user by their ID*

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/scarter"
{
  "_id": "scarter",
  "_rev": "0000000070e587a7",
  "userName": "scarter",
  "givenName": "Sam",
  "sn": "Carter",
  "telephoneNumber": "12345678",
  "active": "true",
  "mail": "scarter@example.com",
  "accountStatus": "active",
  "effectiveAssignments": [],
  "effectiveRoles": []
}
```

+ Add a user with a specific user ID

```
curl \
--header "Content-Type: application/json" \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "If-None-Match: *" \
--request PUT \
--data '{
  "userName": "bjackson",
  "sn": "Jackson",
  "givenName": "Barbara",
  "mail": "bjackson@example.com",
  "telephoneNumber": "082082082",
  "password": "Passw0rd"
}' \
"http://localhost:8080/openidm/managed/user/bjackson"
{
  "_id": "bjackson",
  "_rev": "0000000055c185c5",
  "userName": "bjackson",
  "sn": "Jackson",
  "givenName": "Barbara",
  "mail": "bjackson@example.com",
  "telephoneNumber": "082082082",
  "accountStatus": "active",
  "effectiveAssignments": [],
  "effectiveRoles": []
}
```

+ Add a user with a system-generated ID

```
curl \
--header "Content-Type: application/json" \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request POST \
--data '{
  "userName": "pjensen",
  "sn": "Jensen",
  "givenName": "Pam",
  "mail": "pjensen@example.com",
  "telephoneNumber": "082082082",
  "password": "Passw0rd"
}' \
"http://localhost:8080/openidm/managed/user?_action=create"
{
  "_id": "9d92cdc8-8b22-4037-a344-df960ea66194",
  "_rev": "00000000a4bf9006",
  "userName": "pjensen",
  "sn": "Jensen",
  "givenName": "Pam",
  "mail": "pjensen@example.com",
  "telephoneNumber": "082082082",
  "accountStatus": "active",
  "effectiveAssignments": [],
  "effectiveRoles": []
}
```

#### + *Update a user*

This example checks whether user **bjensen** exists, then replaces her telephone number with the new data provided in the request body:

```
curl \
--header "Content-Type: application/json" \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request POST \
--data '[
{
  "operation": "replace",
  "field": "/telephoneNumber",
  "value": "0763483726"
}
]' \
"http://localhost:8080/openidm/managed/user?_action=patch&_queryFilter=userName+eq+'bjackson'"
{
  "userName": "bjackson",
  "sn": "Jackson",
  "givenName": "Barbara",
  "mail": "bjackson@example.com",
  "telephoneNumber": "0763483726",
  "accountStatus": "active",
  "effectiveAssignments": [],
  "effectiveRoles": [],
  "_rev": "000000008c0f8617",
  "_id": "bjackson"
}
```

#### + Delete a user

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request DELETE \
"http://localhost:8080/openidm/managed/user/bjackson"
{
  "_id": "bjackson",
  "_rev": "000000008c0f8617",
  "userName": "bjackson",
  "sn": "Jackson",
  "givenName": "Barbara",
  "mail": "bjackson@example.com",
  "telephoneNumber": "0763483726",
  "accountStatus": "active",
  "effectiveAssignments": [],
  "effectiveRoles": []
}
```

## Managed Groups

Managed groups are not provided by default. To use managed groups, add an object similar to the following to your object schema:

```
{  
  "name" : "group"  
}
```

Alternatively, create a new managed object type in the Admin UI.

When you add a managed group object to the schema, you have REST access to managed groups, at `/openidm/managed/group`.

For JDBC repositories, IDM stores managed groups with all other managed objects, in the `managedobjects` table, and uses the `managedobjectproperties` for indexing.

For an example of a deployment that uses managed groups, see "[Synchronize LDAP Groups](#)" in the *Samples Guide*.

## Virtual Properties

Properties can be derived from other properties within an object. This lets computed and composite values be created in the object. Such derived properties are named *virtual properties*. The value of a virtual property can be calculated in two ways:

- Using a script called by the `onRetrieve` script hook. This script then calculates the current value of the virtual property based on the related properties.
- Using `queryConfig` to identify the relationship fields to traverse to reach the managed objects whose state is included in the virtual property, and the fields in these managed objects to include in the value of the virtual property.

### Virtual Properties Using `onRetrieve` Scripts

The `onRetrieve` script hook lets you run a script when the object is retrieved. In the case of virtual properties, this script gets the data from related properties and uses it to calculate a value for the virtual property. For more information about running scripts on managed objects, see "[Run Scripts on Managed Objects](#)".

Prior to IDM version 7.0, using `onRetrieve` scripts was the primary method for calculating virtual properties. This method will continue to work, but is not as performant as using `queryConfig`. There may be some cases involving custom logic where a scripted solution is still the preferred answer. For more information about customizing scripts for role calculation, see [Grant a Role By Using Custom Scripts](#).

### Virtual Properties Using `queryConfig`

Virtual properties can be calculated by IDM based on relationships and relationship notifications. This means that rather than calculating the current state when retrieved, the managed object

containing the virtual property is notified of changes in a related object, and the virtual property recalculated when this notification is received. To configure virtual properties to use relationship notifications, there are two areas that need to be configured:

- The related managed objects need to be configured to use relationship notifications. This lets IDM know where to send notifications of changes in related objects. For more information, see "Configure Relationship Change Notification".
- In order to calculate the value of the virtual property, you need configure what relationships to check, and in what order, when it receives a notification of a change in a related object. This is done using the `queryConfig` property.

The `queryConfig` property tells IDM the sequence of relationship fields it should traverse in order to calculate (or recalculate) a virtual property, and what fields it should return from that related object. This is done using two fields:

- `referencedRelationshipFields` is an array listing a sequence of relationship fields connecting the current object with the related objects you want to calculate the value of the virtual property from. The first field in the array is a relationship field belonging to the same managed object as the virtual property, the second field is a relationship in the managed object referenced by the first field, and so on.

For example, the `referencedRelationshipFields` for `effectiveAssignments` is `["roles", "assignments"]`. The first field refers to the `roles` relationship field in `managed/user`, which references the `managed/role` object. It then refers to the `assignments` relationship in `managed/role`, which references the `managed/assignment` object. Changes to either related object (`managed/role` or `managed/assignment`) will cause the virtual property value to be recalculated, due to the `notify`, `notifySelf`, and `notifyRelationships` configurations on managed user, role, and assignment. These configurations ensure that any changes in the relationships between a user and their roles, or their roles, and their assignments, as well as any relevant changes to the roles or assignments themselves, such as the modification of temporal constraints on roles, or attributes on assignments, will be propagated to connected users, so their `effectiveRoles` and `effectiveAssignments` can be recalculated and potentially synced.

- `referencedObjectFields` is an array of object fields that should be returned as part of the virtual property. If this property is not included, the returned properties will be a reference for the related object. To return the entire related object, use `*`.

Using `queryConfig`, the virtual property is recalculated when it receives a notice that changes occurred in the related objects. This can be significantly more efficient than recalculating whenever an object is retrieved, while still ensuring the state of the virtual property is correct.

#### Note

When making changes to what object fields to return using `referencedObjectFields`, the changes will not be reflected until there is a change in the related object that would trigger the virtual property to be recalculated (as specified by the `notify`, `notifySelf`, and `notifyRelationships` configurations). The calculated state of the virtual

property is still correct, but since a change is necessary for the state to be updated, the returned fields will still be based on the previous configuration.

## Run Scripts on Managed Objects

### Important

Before implementing a script, it's highly recommended that you validate the script using REST in the *Scripting Guide* or the API Explorer in the *REST API Reference*. Use scripts in a test environment before deploying them to a production environment.

A number of *script hooks* let you manipulate managed objects using scripts. Scripts can be triggered during various stages of the lifecycle of the managed object, and are defined in the managed object schema.

You can trigger scripts when a managed object is created (onCreate), updated (onUpdate), retrieved (onRetrieve), deleted (onDelete), validated (onValidate), or stored in the repository (onStore). You can also trigger a script when a change to a managed object triggers an implicit synchronization operation (onSync).

Post-action scripts let you manipulate objects after they are created (postCreate), updated (postUpdate), and deleted (postDelete).

The following sample schema runs a script to check that a role has no members before attempting to delete the role:

```
{
  "name" : "role",
  "onDelete" : {
    "type" : "text/javascript",
    "file" : "roles/onDelete-roles.js"
  },
}
```

## Track User Metadata

Some self-service features, such as progressive profile completion, privacy and consent, and terms and conditions acceptance, rely on user *metadata* that tracks information related to a managed object state. Such data might include when the object was created, or the date of the most recent change, for example. This metadata is not stored within the object itself, but in a separate resource location.

Because object metadata is stored outside the managed object, state change situations (such as the time of an update) are separate from object changes (the update itself). This separation reduces unnecessary synchronization to targets when the only data that has changed is metadata. Metadata is not returned in a query unless it is specifically requested. Therefore, the volume of data that is retrieved when metadata is not required, is reduced.

To specify which metadata you want to track for an object, add a `meta` stanza to the object definition in your schema (in the UI or in `conf/managed.json`). The following default configuration tracks the `createDate` and `lastChanged` date for managed user objects:

```
{
  "objects" : [
    {
      "name" : "user",
      ...
      "schema" : {
        ...
      },
      "meta" : {
        "property" : "_meta",
        "resourceCollection" : "internal/usermeta",
        "trackedProperties" : [
          "createDate",
          "lastChanged"
        ]
      },
      ...
    },
    ...
  ]
}
```

### Important

If you are not using the self-service features that require metadata, you can remove the `meta` stanza from your user object definition in `managed.json`. Preventing the creation and tracking of metadata where it is not required will improve performance in that scenario.

The metadata configuration includes the following properties:

#### property

The property that will be dynamically added to the managed object schema for this object.

#### resourceCollection

The resource location in which the metadata will be stored.

Adjust your repository to match the location you specify here. It's recommended that you use an `internal` object path and define the storage in your `repo.jdbc.json` or `repo.ds.json` file.

For a JDBC repository, metadata is stored in the `metaobjects` table by default. The `metaobjectproperties` table is used for indexing.

For a DS repository, metadata is stored under `ou=usermeta,ou=internal,dc=openidm,dc=forgerock,dc=com` by default.

User objects stored in a DS repository must include the `ou` specified in the preceding `dnTemplate` attribute. For example:

```
dn: ou=usermeta,ou=internal,dc=openidm,dc=forgerock,dc=com
objectclass: organizationalunit
objectclass: top
ou: usermeta
```

### trackedProperties

The properties that will be tracked as metadata for this object. In the previous example, the `createDate` (when the object was created) and the `lastChanged` date (when the object was last modified) are tracked.

You cannot search on metadata and it is not returned in the results of a query unless it is specifically requested. To return all metadata for an object, include `_fields=,_meta/*` in your request. The following example returns a user entry without requesting the metadata:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/bjensen"
{
  "_id": "bjensen",
  "_rev": "000000000444dd1a",
  "mail": "bjensen@example.com",
  "givenName": "Barbara",
  "sn": "Jensen",
  "description": "Created By CSV",
  "userName": "bjensen",
  "telephoneNumber": "1234567",
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

The following example returns the same user entry, with their metadata:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/bjensen?_fields=,_meta/*"
{
  "_id": "bjensen",
  "_rev": "000000000444dd1a",
  "mail": "bjensen@example.com",
  "givenName": "Barbara",
  "sn": "Jensen",
  "description": "Created By CSV",
  "userName": "bjensen",
  "telephoneNumber": "1234567",
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
  "_meta": {
```

```
{
  "_ref": "internal/usermeta/284273ff-5e50-4fa4-9d30-4a3cf4a5f642",
  "_refResourceCollection": "internal/usermeta",
  "_refResourceId": "284273ff-5e50-4fa4-9d30-4a3cf4a5f642",
  "_refProperties": {
    "_id": "30076e2e-8db5-4b4d-ab91-5351d2da4620",
    "_rev": "000000001ad09f00"
  },
  "createDate": "2018-04-12T19:53:19.004Z",
  "lastChanged": {
    "date": "2018-04-12T19:53:19.004Z"
  },
  "loginCount": 0,
  "_rev": "0000000094605ed9",
  "_id": "284273ff-5e50-4fa4-9d30-4a3cf4a5f642"
}
```

### Note

Apart from the `createDate` and `lastChanged` shown previously, the request also returns the `loginCount`. This property is stored by default for all objects, and increments with each login request based on password or social authentication. If the object for which metadata is tracked is not an object that "logs in," this field will remain 0.

The request also returns a `_meta` property that includes relationship information. IDM uses the relationship model to store the metadata. When the `meta` stanza is added to the user object definition, the attribute specified by the `property` ("`property`" : "`_meta`", in this case) is added to the schema as a uni-directional relationship to the resource collection specified by `resourceCollection`. In this example, the user object's `_meta` field is stored as an `internal/usermeta` object. The `_meta/_ref` property shows the full resource path to the internal object where the metadata for this user is stored.

## Chapter 2

# Relationships Between Objects

Relationships are references between managed objects. "*Roles*" are implemented using relationships, but you can create relationships between any managed object type.

- "Define a Relationship Type"
- "Create a Relationship Between Two Objects"
- "Configure Relationship Change Notification"
- "Validate Relationships Between Objects"
- "Create Bidirectional Relationships"
- "Grant Relationships Conditionally"
- "View Relationships Over REST"
- "View Relationships in Graph Form"
- "Manage Relationships Through the Admin UI"

## Define a Relationship Type

Relationships are defined in your managed object schema (in `conf/managed.json` if you are using the file-based configuration). The default configuration includes a relationship named `manager` that lets you configure a management relationship between two managed users. The `manager` relationship is a good example from which to understand how relationships work.

The default `manager` relationship is configured as follows:

```
"manager" : {  
  "type" : "relationship",  
  "validate" : true,  
  "reverseRelationship" : true,  
  "reversePropertyName" : "reports",  
  "description" : "Manager",  
  "title" : "Manager",  
  "viewable" : true,  
  "searchable" : false,  
  "usageDescription" : "",  
  "isPersonal" : false,  
}
```

```

"properties" : {
  "_ref" : {
    "description" : "References a relationship from a managed object",
    "type" : "string"
  },
  "_refProperties" : {
    "description" : "Supports metadata within the relationship",
    "type" : "object",
    "title" : "Manager _refProperties",
    "properties" : {
      "_id" : {
        "description" : "_refProperties object ID",
        "type" : "string"
      }
    }
  }
},
"resourceCollection" : [
  {
    "path" : "managed/user",
    "label" : "User",
    "query" : {
      "queryFilter" : "true",
      "fields" : [
        "userName",
        "givenName",
        "sn"
      ]
    }
  }
],
"userEditable" : false
},

```

Most of these properties apply to any **managed object type**. Relationships have the following specific configurable properties:

#### **type (string)**

The object type. Must be **relationship** for a relationship object.

#### **returnByDefault (boolean true, false)**

Specifies whether the relationship should be returned as part of the response. The **returnByDefault** property is not specific to relationships. This flag applies to all managed object types. However, relationship properties are not returned by default, unless explicitly requested.

#### **reverseRelationship (boolean true, false)**

Specifies whether this is a bidirectional relationship.

#### **reversePropertyName (string)**

The corresponding property name, in the case of a **reverse relationship**. For example, the **manager** property has a **reversePropertyName** of **reports**.

## **`_ref` (JSON object)**

Specifies how the relationship between two managed objects is referenced.

In the relationship definition, the value of this property is `{ "type": "string" }`. In a managed user entry, the value of the `_ref` property is the reference to the other resource. The `_ref` property is described in more detail in "Create a Relationship Between Two Objects".

## **`_refProperties` (JSON object)**

Any required properties from the relationship that should be included in the managed object. The `_refProperties` field includes a unique ID (`_id`) and the revision (`_rev`) of the object. `_refProperties` can also contain arbitrary fields to support metadata within the relationship.

## **`resourceCollection` (JSON object)**

The collection of resources (objects) on which this relationship is based (for example, `managed/user` objects).

# Create a Relationship Between Two Objects

When you have defined a relationship *type*, (such as the `manager` relationship, described in the previous section), you can *reference* one managed user from another, using the `_ref*` relationship properties. Three properties make up a relationship reference:

- `_refResourceCollection` specifies the container of the referenced object (for example, `managed/user`).
- `_refResourceId` specifies the ID of the referenced object. This is generally a system-generated UUID, such as `9dce06d4-2fc1-4830-a92b-bd35c2f6bcbb`. For clarity, this section uses client-assigned IDs such as `bjensen` and `psmith`.
- `_ref` is a derived path that is a combination of `_refResourceCollection` and a URL-encoded `_refResourceId`.

For example, imagine that you are creating a new user, `psmith`, and that `psmith`'s manager will be `bjensen`. You would add `psmith`'s user entry, and *reference* `bjensen`'s entry with the `_ref` property, as follows:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "If-None-Match: *" \
--header "Content-Type: application/json" \
--request PUT \
--data '{
  "sn": "Smith",
  "userName": "psmith",
  "_ref": "managed/user/bjensen"
```

```
"givenName":"Patricia",
"displayName":"Patti Smith",
"description": "psmith - new user",
"mail": "psmith@example.com",
"phoneNumber": "0831245986",
"password": "Passw0rd",
"manager": {"_ref": "managed/user/bjensen"}
}' \
"http://localhost:8080/openidm/managed/user/psmith"
{
  "_id": "psmith",
  "_rev": "00000000ec41097c",
  "sn": "Smith",
  "userName": "psmith",
  "givenName": "Patricia",
  "displayName": "Patti Smith",
  "description": "psmith - new user",
  "mail": "psmith@example.com",
  "phoneNumber": "0831245986",
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

Note that relationship information is not returned by default. To show the relationship in psmith's entry, you must explicitly request her manager entry, as follows:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/psmith?_fields=manager"
{
  "_id": "psmith",
  "_rev": "00000000ec41097c",
  "manager": {
    "_ref": "managed/user/bjensen",
    "_refResourceCollection": "managed/user",
    "_refResourceId": "bjensen",
    "_refProperties": {
      "_id": "ffc6f0f3-93db-4939-b9eb-1f8389a59a52",
      "_rev": "0000000081aa991a"
    }
  }
}
```

If a relationship changes, you can query the updated relationship state when any referenced managed objects are queried. So, after creating user psmith with manager bjensen, a query on bjensen's user entry will show a reference to psmith's entry in her **reports** property (because the **reports** property is configured as the **reversePropertyName** of the **manager** property). The following query shows the updated relationship state for bjensen:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/bjensen?_fields=reports"
{
  "_id": "bjensen",
  "_rev": "0000000057b5fe9d",
  "reports": [
    {
      "_ref": "managed/user/psmith",
      "_refResourceCollection": "managed/user",
      "_refResourceId": "psmith",
      "_refProperties": {
        "_id": "ffc6f0f3-93db-4939-b9eb-1f8389a59a52",
        "_rev": "0000000081aa991a"
      }
    }
  ]
}
```

IDM maintains referential integrity by deleting the relationship reference, if the object referred to by that relationship is deleted. In our example, if bjensen's user entry is deleted, the corresponding reference in psmith's **manager** property is removed.

## Configure Relationship Change Notification

A relationship exists between two managed objects. By default, when a relationship changes (when it is created, updated, or deleted), the managed objects on either side of the relationship are not *notified* of that change. This means that the *state* of each object with respect to that relationship field is not recalculated until the object is read. This default behavior improves performance, especially in the case where many objects are affected by a single relationship change.

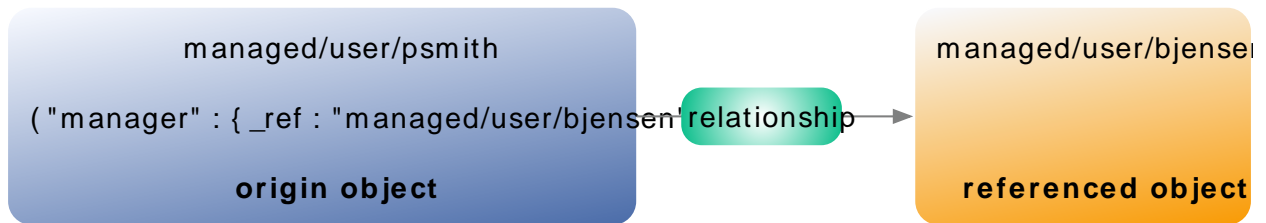
For **roles**, a special kind of relationship, change notification is configured by default. The purpose of this default configuration is to notify managed users when any of the relationships that link users, roles, and assignments are manipulated. For more information about relationship change notification in the specific case of managed roles, see ["Roles and Relationship Change Notification"](#).

To change the default configuration, or to set up notification for other relationship changes, use the **notify\*** properties in the relationship definition, as described in this section.

A relationship exists between an *origin* object and a *referenced* object. These terms reflect which managed object is specified in the URL (for example **managed/user/psmith**), and which object is referenced by the relationship (**\_ref\***) properties. For more information about the relationship properties, see ["Create a Relationship Between Two Objects"](#).

In the previous example, a PUT on **managed/user/psmith** with **"manager" : {\_ref : "managed/user/bjensen"}**, causes **managed/user/psmith** to be the origin object, and **managed/user/bjensen** to be the referenced object for that relationship, as shown in the following illustration:

## Relationship Objects



Note that for the reverse relationship (a PUT on `managed/user/bjensen` with `"reports" : [{_ref = "managed/user/psmith"}]`) `managed/user/bjensen` would be the origin object, and `managed/user/psmith` would be the referenced object.

By default, when a relationship changes, neither the origin object nor the referenced object is *notified* of the change. So, with the PUT on `managed/user/psmith` with `"manager" : {_ref : "managed/user/bjensen"}`, neither `psmith`'s object nor `bjensen`'s object is notified.

### Note

Auditing is not tied to relationship change notification and is always triggered when a *relationship* changes. Therefore, relationship changes are audited, regardless of the `notify` and `notifySelf` properties.

To configure relationship change notification, set the `notify` and `notifySelf` properties in your managed object schema. These properties specify whether objects that reference relationships are notified of a relationship change:

### `notifySelf`

Notifies the origin object of the relationship change.

In our example, if the `manager` definition includes `"notifySelf" : true`, and if the relationship is changed through a URL that references `psmith`, then `psmith`'s object would be notified of the change. For example, for a CREATE, UPDATE or DELETE request on the `psmith/manager`, `psmith` would be notified, but the managed object referenced by this relationship (`bjensen`) would not be notified.

If the relationship were manipulated through a request to `bjensen/reports`, then `bjensen` would only be notified if the `reports` relationship specified `"notifySelf" : true`.

### `notify`

Notifies the referenced object of the relationship change.

Set this property on the `resourceCollection` of the relationship property. In our example, assume that the `manager` definition has a `resourceCollection` with a `path` of `managed/user`, and that this object

specifies `"notify" : true`. If the relationship changes through a CREATE, UPDATE, or DELETE on the URL `psmith/manager`, then the reference object (`managed/user/bjensen`) would be notified of the change to the relationship.

## notifyRelationships

This property controls the propagation of notifications out of a managed object when one of its properties changes through an update or patch, or when that object receives a notification through one of these fields.

The `notifyRelationships` property takes an array of relationships as a value; for example, `"notifyRelationships" : ["relationship1", "relationship2"]`. The relationships specified here are fields defined on the managed object type (which might itself be a relationship).

Notifications are propagated according to the *recipient's* `notifyRelationships` configuration. If a managed object type is notified of a change through one of its relationship fields, the notification is done according to the configuration of the recipient object. To illustrate, look at the `attributes` property in the default `managed/assignment` object:

```
{
  "name" : "assignment",
  "schema" : {
    ...
    "properties" : {
      ...
      "attributes" : {
        "description" : "The attributes operated on by this assignment.",
        "title" : "Assignment Attributes",
        ...
        "notifyRelationships" : ["roles"]
      },
    },
    ...
  }
}
```

This configuration means that if an assignment is updated or patched, and the assignment's `attributes` change in some way, all the `roles` connected to that assignment are notified. Because the `role` managed object has `"notifyRelationships" : ["members"]` defined on its `assignments` field, the notification that originated from the change to the assignment attribute is propagated to the connected `roles`, and then out to the `members` of those roles.

So, the `role` is notified through its `assignments` field because an `attribute` in the assignment changed. This notification is propagated out of the `members` field because the role definition has `"notifyRelationships" : ["members"]` on its `assignments` field.

By default, `roles`, `assignments`, and `members` use relationship change notification to ensure that relationship changes are accurately provisioned.

For example, the default `user` object includes a `roles` property with `notifySelf` set to `true`:

```
{
  "name" : "user",
  ...
  "schema" : {
    ...
    "properties" : {
      ...
      "roles" : {
        "description" : "Provisioning Roles",
        ...
        "items" : {
          "type" : "relationship",
          ...
          "reverseRelationship" : true,
          "reversePropertyName" : "members",
          "notifySelf" : true,
          ...
        }
      }
    }
  }
  ...
}
```

In this case, `notifySelf` indicates the origin or `user` object. If any changes are made to a relationship referencing a role through a URL that includes a user, the user will be notified of the change. For example, if there is a CREATE on `managed/user/psmith/roles` which specifies a set of references to existing roles, user `psmith` will be notified of the change.

Similarly, the `role` object includes a `members` property. That property includes the following schema definition:

```
{
  "name" : "role",
  ...
  "schema" : {
    ...
    "properties" : {
      ...
      "members" : {
        ...
        "items" : {
          "type" : "relationship",
          ...
          "properties" : {
            ...
            "resourceCollection" : [
              {
                "notify" : true,
                "path" : "managed/user",
                "label" : "User",
                ...
              }
            ]
          }
        }
      }
    }
  }
  ...
}
```

Notice the `"notify" : true` setting on the `resourceCollection`. This setting indicates that if the relationship is created, updated, or deleted through a URL that references that role, all objects in

that resource collection (in this case, `managed/user` objects) that are identified as `members` of that role must be notified of the change.

### Important

- To notify an object at the end of a relationship that the relationship has changed (using the `notify` property), the relationship *must* be bidirectional (`"reverseRelationship" : true`).

When an object is notified of a relationship state change (create, delete, or update), part of that notification process involves calculating the changed object state with respect to the changed relationship field. For example, if a managed user is notified that a role has been created, the user object calculates its base state, and the state of its `roles` field, before and after the new role was created. This *before* and *after* state is then reconciled. An object that is referenced by a forward (unidirectional) relationship does not have a field that references that relationship; the object is "pointed-to", but does not "point-back". Because this object cannot calculate its *before* and *after* state with respect to the relationship field, it cannot be notified.

Similarly, relationships that are notified of changes to the objects that reference them *must* be bidirectional relationships.

If you configure relationship change notification on a unidirectional relationship, IDM throws an exception.

- You cannot configure relationship change notification in the Admin UI; you must update the managed object schema in the `conf/managed.json` file directly.

## Validate Relationships Between Objects

Optionally, you can specify that a relationship between two objects must be validated when the relationship is created. For example, you can indicate that a user cannot reference a role, if that role does not exist.

When you create a new relationship type, validation is disabled by default, because it involves an expensive query to the relationship that is not always required. To configure validation of a referenced relationship, set `"validate": true` in the schema. The default `managed.json` files provided with the sample configurations enable validation for the following relationships:

- For user objects – roles, managers, and reports
- For role objects – members and assignments
- For assignment objects – roles

The following configuration of the `manager` relationship enables validation, and prevents a user from referencing a manager that has not already been created:

```
"manager" : {
  "type" : "relationship",
  ...
  "validate" : true,
```

## Create Bidirectional Relationships

In most cases, you define a relationship between two objects *in both directions*. For example, a relationship between a user and his manager might indicate a *reverse relationship* between the manager and her direct report. Reverse relationships are particularly useful for queries. You might want to query `jdoe`'s user entry to discover who his manager is, *or* query `bjensen`'s user entry to discover all the users who report to `bjensen`.

You declare a reverse relationship as part of the relationship definition. Consider the following sample excerpt of the default managed object configuration:

```
"reports" : {
  "description" : "Direct Reports",
  "title" : "Direct Reports",
  ...
  "type" : "array",
  "returnByDefault" : false,
  "items" : {
    "type" : "relationship",
    "reverseRelationship" : true,
    "reversePropertyName" : "manager",
    "validate" : true,
    ...
  }
  ...
}
```

The `reports` property is a `relationship` between users and managers. So, you can *refer* to a managed user's reports by referencing the `reports`. However, the `reports` property is also a reverse relationship (`"reverseRelationship" : true`) which means that you can list all users that reference that report.

You can list all users whose `manager` property is set to the currently queried user.

The reverse relationship includes an optional `resourceCollection` that lets you query a set of objects, based on specific fields:

```
"resourceCollection" : [
  {
    "path" : "managed/user",
    "label" : "User",
    "query" : {
      "queryFilter" : "true",
      "fields" : [
        "userName",
        "givenName",
        "sn"
      ]
    }
  }
]
```

The `path` property of the `resourceCollection` points to the set of objects to be queried. If this path is not in the local repository, the link expansion can incur a significant performance cost. Although the `resourceCollection` is optional, the same performance cost is incurred if the property is absent.

The `query` property indicates how you will query this resource collection to configure the relationship. In this case, `"queryFilter" : "true"`, indicates that you can search on any of the properties listed in the `fields` array when you are assigning a manager to a user or a new report to a manager. To configure these relationships from the Admin UI, see ["Manage Relationships Through the Admin UI"](#).

## Grant Relationships Conditionally

Relationships can be granted dynamically, based on a specified condition. In order to conditionally grant a relationship, the schemas for the resources you are creating a relationship between need to be configured to support conditional association. To do this, three fields in the schema are used:

### `conditionalAssociation`

Boolean. This property is applied to the `resourceCollection` for the grantor of the relationship. For example, the `members` relationship on `managed/role` specifies that there is a conditional association with the `managed/user` resource:

```
"resourceCollection" : [
  {
    "notify" : true,
    "conditionalAssociation" : true,
    "path" : "managed/user",
    "label" : "User",
    "query" : {
      "queryFilter" : "true",
      "fields" : [
        "userName",
        "givenName",
        "sn"
      ]
    }
  }
]
```

### `conditionalAssociationField`

This property is a string, specifying the field used to determine whether a conditional relationship is granted. The field is applied to the `resourceCollection` of the grantee of the relationship. For example, the `roles` relationship on `managed/user` specifies that the conditional association with `managed/role` is defined by the `condition` field in `managed/role`:

```
"resourceCollection" : [
  {
    "path" : "managed/role",
    "label" : "Role",
    "conditionalAssociationField" : "condition",
    "query" : {
      "queryFilter" : "true",
      "fields" : [
        "name"
      ]
    }
  }
]
```

The field name specified will usually be `condition` if you are using default schema, but can be any field that evaluates a condition and has been flagged as `isConditional`.

### `isConditional`

Boolean. This is applied to the field you wish to check to determine whether membership in a relationship is granted. Only one field on a resource can be marked as `isConditional`. For example, in the relationship between `managed/user` and `managed/role`, conditional membership in the relationship is determined by the query filter specified in the `managed/role condition` field:

```
"condition" : {
  "description" : "A conditional filter for this role",
  "title" : "Condition",
  "viewable" : false,
  "searchable" : false,
  "isConditional" : true,
  "type" : "string"
}
```

Conditions can be a powerful tool for dynamically creating relationships between two objects. An example of conditional relationships in use can be seen in [Grant a Role Based on a Condition](#).

## View Relationships Over REST

By default, information about relationships is not returned as the result of a GET request on a managed object. You must explicitly include the relationship property in the request, for example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/psmith?_fields=manager"
{
  "id": "psmith",
  "_rev": "0000000014c0b68d",
  "manager": {
    "_ref": "managed/user/bjensen",
    "_refResourceCollection": "managed/user",
    "_refResourceId": "bjensen",
    "_refProperties": {
      "id": "42418f09-ad6c-4b77-bf80-2a12d0c44678",
      "_rev": "00000000288b921e"
    }
  }
}
```

To obtain more information about the referenced object (psmith's manager, in this case), you can include additional fields from the referenced object in the query, using the syntax `object/property` (for a simple string value) or `object/*/property` (for an array of values).

The following example returns the email address and contact number for psmith's manager:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/psmith?_fields=manager/mail,manager/telephoneNumber"
{
  "id": "psmith",
  "_rev": "0000000014c0b68d",
  "manager": {
    "_rev": "000000005bac8c10",
    "id": "bjensen",
    "telephoneNumber": "12345678",
    "mail": "bjensen@example.com",
    "_ref": "managed/user/bjensen",
    "_refResourceCollection": "managed/user",
    "_refResourceId": "bjensen",
    "_refProperties": {
      "id": "42418f09-ad6c-4b77-bf80-2a12d0c44678",
      "_rev": "00000000288b921e"
    }
  }
}
```

To query all the relationships associated with a managed object, query the reference (`*_ref`) property of that object. For example, the following query shows all the objects that are referenced by psmith's entry:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
```

```
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/psmith?_fields=*_ref"
{
  "_id": "psmith",
  "_rev": "0000000014c0b68d",
  "reports": [],
  "manager": {
    "_ref": "managed/user/bjensen",
    "_refResourceCollection": "managed/user",
    "_refResourceId": "bjensen",
    "_refProperties": {
      "_id": "42418f09-ad6c-4b77-bf80-2a12d0c44678",
      "_rev": "00000000288b921e"
    }
  },
  "roles": [],
  "meta": {
    "_ref": "internal/usermeta/601a3086-8c64-4966-b33c-7a213b13d859",
    "_refResourceCollection": "internal/usermeta",
    "_refResourceId": "601a3086-8c64-4966-b33c-7a213b13d859",
    "_refProperties": {
      "_id": "9de71bd7-1e1b-462e-b565-ac0a7d2f9269",
      "_rev": "0000000037f79a00"
    }
  },
  "authzRoles": [],
  "notifications": [
    {
      "_ref": "internal/notification/3000bb64-4619-490a-8c4b-50ae7ca6b20c",
      "_refResourceCollection": "internal/notification",
      "_refResourceId": "3000bb64-4619-490a-8c4b-50ae7ca6b20c",
      "_refProperties": {
        "_id": "f54b6f84-7d3f-4486-a7c1-676fca03eeab",
        "_rev": "00000000748da107"
      }
    }
  ]
}
```

To expand that query to show all fields within each relationship, add a wildcard as follows:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/psmith?_fields=*_ref/*"
{
  "_id": "psmith",
  "_rev": "0000000014c0b68d",
  "reports": [],
  "manager": {
    "_rev": "000000005bac8c10",
    "_id": "bjensen",
    "userName": "bjensen",
    "givenName": "Babs",
    "sn": "Jensen",
    "telephoneNumber": "12345678",
```

```

    "active": "true",
    "mail": "bjensen@example.com",
    "accountStatus": "active",
    "effectiveAssignments": [],
    "effectiveRoles": [],
    "_ref": "managed/user/bjensen",
    "_refResourceCollection": "managed/user",
    "_refResourceId": "bjensen",
    "_refProperties": {
      "_id": "42418f09-ad6c-4b77-bf80-2a12d0c44678",
      "_rev": "00000000288b921e"
    }
  },
  "roles": [],
  "_meta": {
    "_rev": "0000000079e86d8d",
    "_id": "601a3086-8c64-4966-b33c-7a213b13d859",
    "createDate": "2020-07-29T08:52:20.061794Z",
    "lastChanged": {
      "date": "2020-07-29T11:52:16.424167Z"
    }
  },
  "loginCount": 0,
  "_ref": "internal/usermeta/601a3086-8c64-4966-b33c-7a213b13d859",
  "_refResourceCollection": "internal/usermeta",
  "_refResourceId": "601a3086-8c64-4966-b33c-7a213b13d859",
  "_refProperties": {
    "_id": "9de71bd7-1e1b-462e-b565-ac0a7d2f9269",
    "_rev": "0000000037f79a00"
  }
},
"authzRoles": [],
"notifications": [
  {
    "_rev": "00000000d93a6598",
    "_id": "3000bb64-4619-490a-8c4b-50ae7ca6b20c",
    "notificationType": "info",
    "message": "Your profile has been updated.",
    "createDate": "2020-07-29T11:52:16.517200Z",
    "_ref": "internal/notification/3000bb64-4619-490a-8c4b-50ae7ca6b20c",
    "_refResourceCollection": "internal/notification",
    "_refResourceId": "3000bb64-4619-490a-8c4b-50ae7ca6b20c",
    "_refProperties": {
      "_id": "f54b6f84-7d3f-4486-a7c1-676fca03eeab",
      "_rev": "00000000748da107"
    }
  }
]
}

```

### Note

Metadata is implemented using the relationships mechanism so when you request all relationships for a user (with `_ref/`), you will also get all the metadata for that user, if metadata is being tracked. For more information, see "Track User Metadata".

## View Relationships in Graph Form

The *Identity Relationships* widget gives a visual display of the relationships between objects.

This widget is not displayed on any dashboard by default. You can add it as follows:

1. Log in to the Admin UI.
2. Select Dashboards, and choose the dashboard to which you want to add the widget.

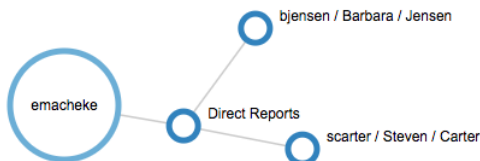
For more information about managing dashboards in the UI, see "Manage Dashboards" in the *Setup Guide*.

3. Select Add Widget.
4. In the Add Widget window, scroll down to the Utilities item, select Identity Relationships, then click Settings.
5. Choose the Widget Size (small, medium, or large).
6. From the Chart Type list, select Collapsible Tree Layout or Radial Layout.

The Collapsible Tree Layout looks something like this:



The Radial Layout looks something like this:



7. Select the object for which you want to display relationships, for example, **User**.

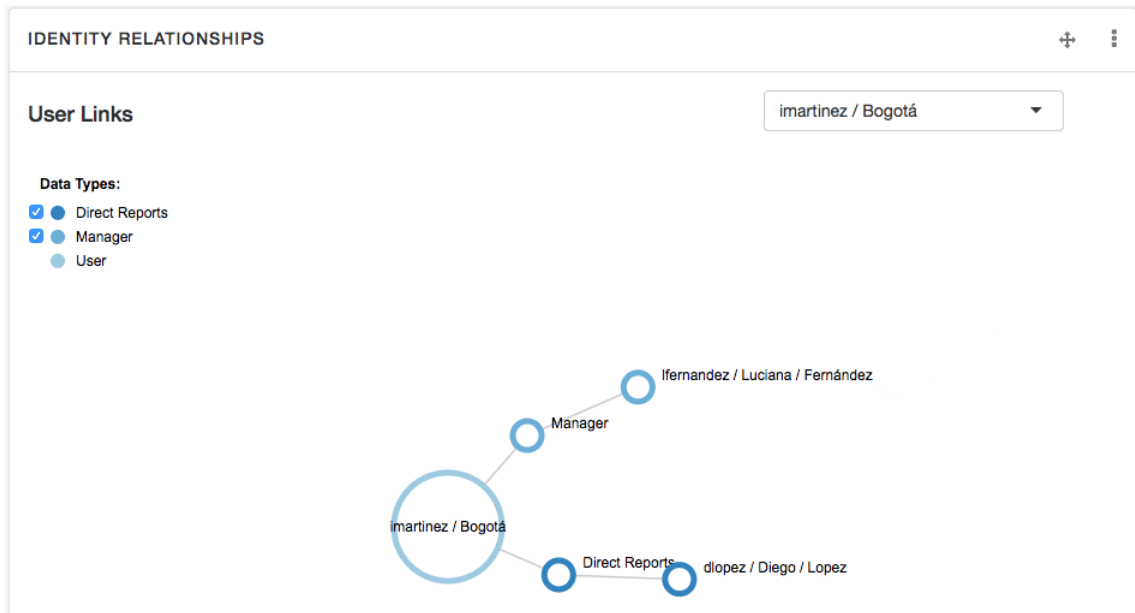
8. Select the property or properties that will be used to search on that object, and that will be displayed in the widget, for example, `userName` and `city`.

Optionally, select Preview for an idea of what the data represented by widget will look like. Select Settings to return to the Add Widget window.

9. Click Add to add the widget to the dashboard.

When you have added the Identity Relationships widget, select the user whose relationships you want to search.

The following graph shows all of imartinez's relationships. The graph shows imartinez's manager and her direct reports.



Select or deselect the Data Types on the left of the screen to control how much information is displayed.

Select and move the graph for a better view. Double-click on any user in the graph to view that user's profile.

## Manage Relationships Through the Admin UI

This section describes how to set up relationships between managed objects by using the Admin UI. You can set up a relationship between any object types. The examples in this section demonstrate how to set up a relationship between users and devices, such as IoT devices.

For illustration purposes, these examples assume that you have started IDM and already have some managed users. If this is not the case, start the server with the sample configuration described in *"Synchronize Data From a CSV File to IDM"* in the *Samples Guide*, and run a reconciliation to populate the managed user repository.

In the following procedures, you will:

- Create a new managed object type named **Device** and add a few devices, each with unique serial numbers (see *"Create a New Device Object Type"*).
- Set up a bidirectional relationship between the Device object and the managed User object (see *"Configure the Relationship Between a Device and a User"*).
- Demonstrate the relationships, assign devices to users, and show relationship validation (see *"Demonstrate the Relationship"*).

### Create a New Device Object Type

This procedure illustrates how to set up a new Device managed object type, adding properties to collect information such as model, manufacturer, and serial number for each device. In the next procedure, you will set up the relationship.

1. Click **Configure > Managed Objects > New Managed Object**.

Give the object an appropriate name and Readable Title. For this procedure, specify **Device** for both these fields.


Enter a description for the object, select an icon that represents the object, and click **Save**.

You should now see three tabs: **Properties**, **Details**, and **Scripts**. Select the **Properties** tab.

2. Click **Add a Property** to set up the schema for the device.

For each property, enter a **Name**, and **Label**, select the data **Type** for the property, and specify whether that property is required for an object of this type.

For the purposes of this example, include the properties shown in the following image: **model**, **serialNumber**, **manufacturer**, **description**, and **category**.



MANAGED OBJECT  
Device

+ New Mapping

Properties
Details
Scripts

+ Add a Property

| PROPERTY NAME | LABEL         | TYPE   | REQUIRED   |   |  |   |
|---------------|---------------|--------|------------|---|--|---|
| model         | Model         | String | ✓ Required | + |  | x |
| serialNumber  | Serial Number | String | ✓ Required | + |  | x |
| manufacturer  | Manufacturer  | String | ✓ Required | + |  | x |
| description   | Description   | String | ✓ Required | + |  | x |
| category      | Category      | String | ✓ Required | + |  | x |


String

☐

Save

When you save the properties for the new managed object type, IDM saves those entries in your project's `conf/managed.json` file.

- Now select Manage > Device > New Device and add a device as shown in the following image:



DEVICE

## New Device

Details

Model

Serial Number

Manufacturer

Description

Category

Save

- Continue adding new devices to the Device object.

When you have finished, select Manage > Device to view the complete list of Devices.

The remaining procedures in this section assume that you have added devices similar to the following:

## Device List

+ New Device

↻ Reload Grid

✕ Clear Filters

🗑 Delete Selected

|                          |                      |                      |                      |                          |             |
|--------------------------|----------------------|----------------------|----------------------|--------------------------|-------------|
| <div>Filter...</div>     | <div>Filter...</div> | <div>Filter...</div> | <div>Filter...</div> | <div>Filter...</div>     |             |
| <input type="checkbox"/> | MODEL                | SERIAL NUMBER        | MANUFACTURER         | DESCRIPTION              | CATEGORY    |
| <input type="checkbox"/> | Generic Phone        | Phone-1              | PhoneCo              | Entry level phone        | Smart Phone |
| <input type="checkbox"/> | Generic Watch        | Watch-1              | WatchCo              | Entry level watch        | Smart Watch |
| <input type="checkbox"/> | Special Phone        | Phone-2              | PhoneCo              | Intermediate level phone | Smart Phone |
| <input type="checkbox"/> | Special Watch        | Watch-2              | WatchCo              | Intermediate level watch | Smart Watch |

«

<

>

»

- (Optional) To change the order in which properties of the Device managed object are displayed, select **Configure > Managed Objects > Device**. Select the property that you want to move and drag it up or down the list.

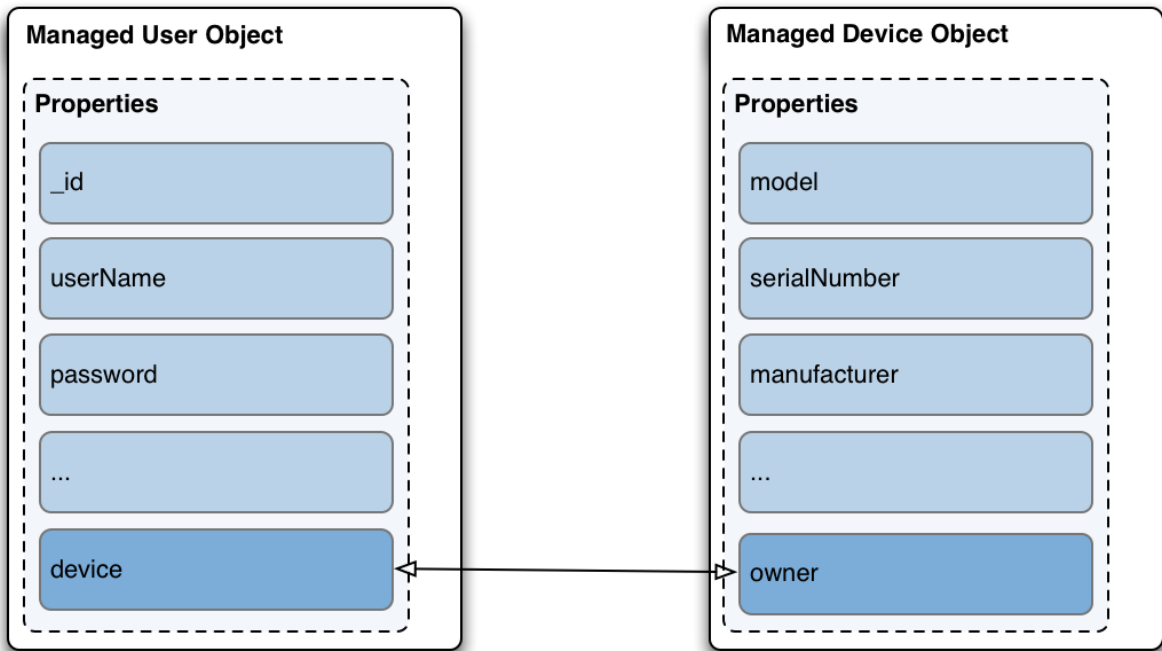
Alternatively, you can make the same changes to this schema (or any managed object schema) in your project's `conf/managed.json` file.

### Configure the Relationship Between a Device and a User

To set up a relationship between the Device object type and the User object type, you must identify the specific property on each object that will form the basis of the relationship. For example, a device must have an *owner* and a user can own one or more *devices*. The property *type* for each of these must be *relationship*.

In this procedure, you will update the managed Device object type to add a new Relationship type property named `owner`. You will then link that property to a new property on the managed User object, named `device`. At the end of the procedure, the updated object types will look as follows:

## Relationship Properties on User and Device Objects



1. Create a new relationship property on the Device object:
  - a. Select Configure > Managed Objects and select the Device object that you created previously.
  - b. On the Properties tab, add a new property named **owner**. Select Relationship as the property Type. Select Required, as all device objects *must* have an owner:

|       |       |              |                                     |      |
|-------|-------|--------------|-------------------------------------|------|
| owner | Owner | Relationship | <input checked="" type="checkbox"/> | Save |
|-------|-------|--------------|-------------------------------------|------|

### Note

You cannot change the Type of a property after it has been created. If you create the property with an incorrect Type, you must delete the property and recreate it.

2. When you have saved the Owner property, select it to show the relationship on the Details tab:

↔

RELATIONSHIP PROPERTY

owner

Details

Validation

Privacy & Encryption

Scripts

Readable Title

Owner

Description

Required

☒

Relationship Configuration

Device

Has one owner

+ Two-way Relationship

+ Related Resource

Relationship Properties

| PROPERTY NAME | LABEL |       |
|---------------|-------|-------|
| Name          | Label | + Add |

Show advanced options

Save

- Click the + Related Resource item and select **user** as the Resource.

This sets up a relationship between the new Device object and the managed User object.

Under Display Properties, select all of the properties of the user object that should be visible when you display a user's devices in the UI. For example, you might want to see the user's name, email address and telephone number.

Note that this list of Display Properties also specifies how you can *search* for user objects when you are assigning a device to a user.

Click Show advanced options. Notice that the Query Filter field is set to **true**. This setting allows you to search on any of the Display Properties that you have selected, when you are assigning a device to a user.

Click Save to continue.

You now have a one-way relationship between a device and a user.

4. Click the + Two-way Relationship item to configure the reverse relationship:
  - a. Select Has Many to indicate that a single user can have more than one device.
  - b. In the Reverse property name field, enter the new property name that will be created in the managed User object type. As shown in "Relationship Properties on User and Device Objects", that property is **device** in this example.
  - c. Under Display Properties, select all of the properties of the device object that should be visible when you display a user in the UI. For example, you might want to see the model and serial number of each device.
  - d. Click Show advanced options. Notice that the Query Filter field is set to **true**. This setting allows you to search on any of the Display Properties that you have selected, when you are assigning a device to a user.
  - e. Select Validate relationship.

This setting ensures that the relationship is valid when a device is assigned to a user. IDM verifies that both the user and device objects exist, and that that specific device has not already been assigned to user.
  - f. Click Save to continue.
5. You should now have the following reverse relationship configured between User objects and Device objects:

↔

RELATIONSHIP PROPERTY

owner

⋮

Details

Validation

Privacy & Encryption

Scripts

Readable Title

Owner

Description

Required

☒

Relationship Configuration

Device

Has one owner

Has many device

user

+ Related Resource

Relationship Properties

| PROPERTY NAME | LABEL |       |
|---------------|-------|-------|
| Name          | Label | + Add |

Show advanced options

Save

Select Configure > Managed Objects > User.

Scroll down to the end of the Properties tab and notice that the **device** property was created automatically when you configured the relationship.

### Demonstrate the Relationship

This procedure demonstrates how devices can be assigned to users, based on the relationship configuration that you set up in the previous two procedures.

1. Select Manage > User, click on a user entry and select the new Device tab.

- Click Add Device and click in the Device field to display the list of devices that you added in the previous procedure.

**Add Device** ✕

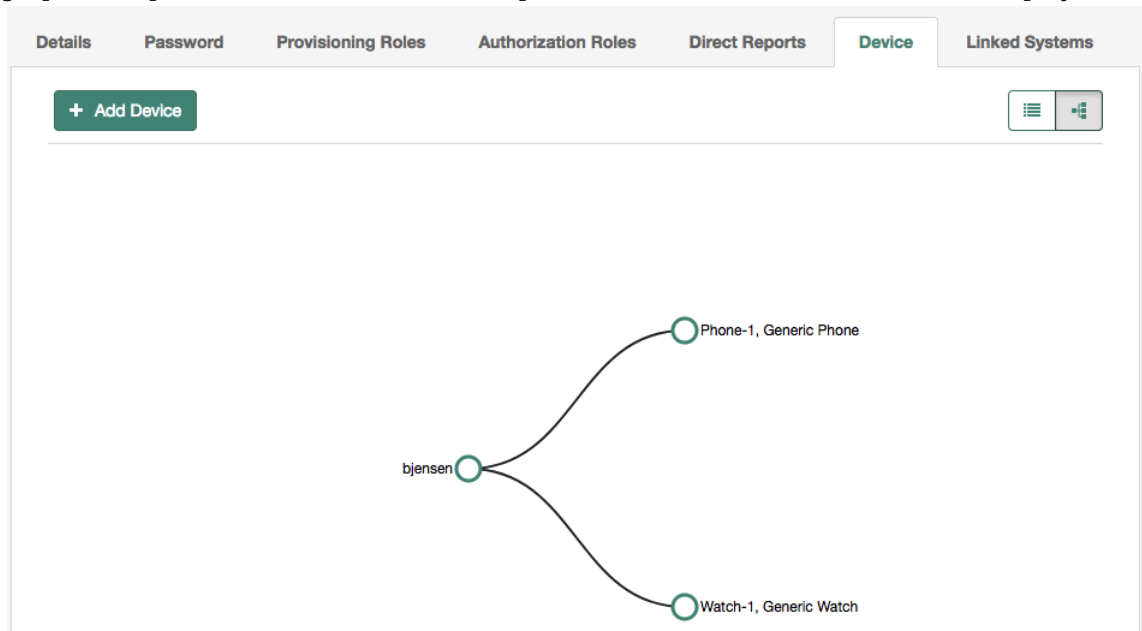
**Device**

- Phone-1, Generic Phone
- Phone-2, Special Phone
- Watch-1, Generic Watch
- Watch-2, Special Watch

Close Add

- Select two devices and click Add.
- On the Device tab, click the Show Chart icon at the top right.

A graphical representation of the relationship between the user and her devices is displayed:



- You can also assign an owner to a device.  
Select Manage > Device, and select one of the devices that you did not assign in the previous step.  
Click Add Owner and search for the user to whom the device should be assigned.
- To demonstrate the relationship validation, try to assign a device that has already been assigned to a different user.

The UI displays the error: **Conflict with Existing Relationship**.

### View the Relationship Configuration in the UI

The *Managed Objects Relationship Diagram* provides a visual display of the relationship configuration between managed objects. Unlike the Identity Relationships widget, described in "View Relationships in Graph Form", this widget does not show the actual relationship data, but rather shows the configured relationship types.

This widget is not displayed on any dashboard by default. You can add it as follows:

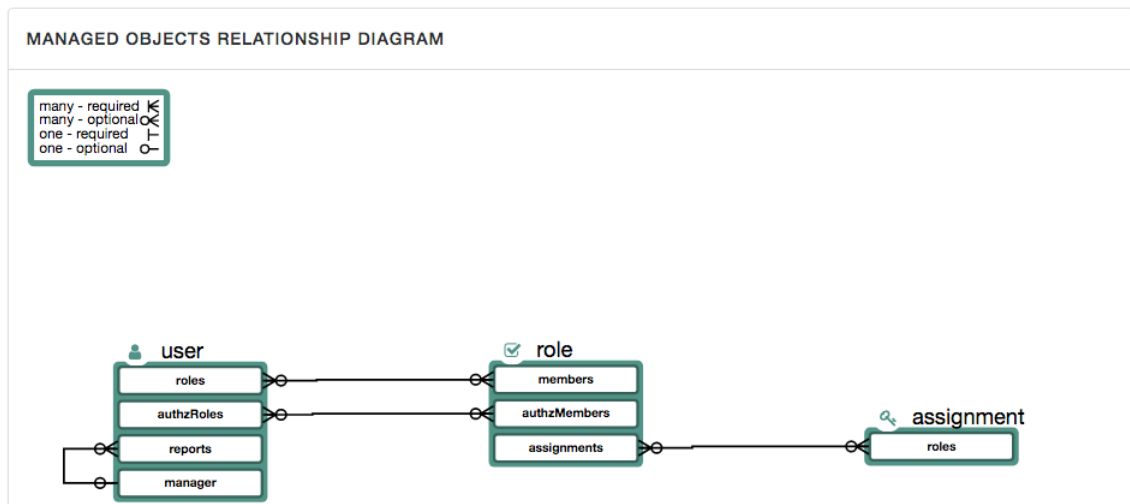
1. Log in to the Admin UI.
2. Select Dashboards, and choose the dashboard to which you want to add the widget.

For more information about managing dashboards in the UI, see "Manage Dashboards" in the *Setup Guide*.

3. Select Add Widget.
4. In the Add Widget window, scroll down to the Utilities item and select Managed Objects Relationship Diagram.

There are no configurable settings for this widget.

5. The Preview button shows the current relationship configuration. The following image shows the relationship configuration for a basic IDM installation with no specific configuration:



The legend indicates which relationships are required, which are optional, and which are one to one or one to many. In the default relationship configuration shown in the previous image, you

can see that a user can have one or more roles and a role can have one or more users. A manager can have one or more reports but a user can have only one manager. There are no mandatory relationships in this default configuration.

## Chapter 3

# Roles

The managed *role* object is a default managed object type that uses the *relationships* mechanism. You should understand how relationships work before you read about IDM roles.

- "IDM Role Types"
- "Managed Roles"
- "Manipulate Roles Over REST and in the UI"
- "Use Temporal Constraints to Restrict Effective Roles"
- "Use Assignments to Provision Users"
- "Effective Roles and Effective Assignments"
- "Roles and Relationship Change Notification"
- "Managed Role Script Hooks"
- "Use Groups to Control Access to IDM"

## IDM Role Types

IDM supports two types of roles:

- *Provisioning roles*: used to specify how objects are provisioned to an external system.

Provisioning roles are created as managed roles, at the context path `openidm/managed/role/role-name`, and are granted to managed users as values of the user's `roles` property.

- *Authorization roles*: used to specify the authorization rights of a managed object internally, within IDM.

Authorization roles are created as internal roles, at the context path `openidm/internal/role/role-name`, and are granted to managed users as values of the user's `authzRoles` property.

Provisioning roles and authorization roles use *relationships* to link the role to the managed object to which it applies. Authorization roles can also be granted statically, during authentication, with the `defaultUserRoles` property. For more information, see "Authentication and Roles" in the *Security Guide*.

## Managed Roles

These sections describe how to create and use provisioning roles. For information about internal authorization roles, and how IDM controls authorization to its own endpoints, see "Authorization and Roles" in the *Security Guide*.

*Managed roles* are defined like any other managed object, and are granted to users through the *relationships* mechanism.

A managed role can be granted manually, as a static value of the user's `roles` attribute, or dynamically, as a result of a condition or script. For example, a user might be granted a role such as `sales-role` dynamically, if that user is in the `sales` organization.

A user's `roles` attribute takes an array of *references* as a value, where the references point to the managed roles. For example, if user `bjensen` has been granted two roles (`employee` and `supervisor`), the value of `bjensen`'s `roles` attribute would look something like the following:

```
"roles": [
  {
    "_ref": "managed/role/employee",
    "_refResourceCollection": "managed/role",
    "_refResourceId": "employee",
    "_refProperties": {
      "_grantType": "",
      "_id": "bb399428-21a9-4b01-8b74-46a7ac43e0be",
      "_rev": "00000000e43e9ba7"
    }
  },
  {
    "_ref": "managed/role/supervisor",
    "_refResourceCollection": "managed/role",
    "_refResourceId": "supervisor",
    "_refProperties": {
      "_grantType": "",
      "_id": "9f7d124b-c7b1-4bcf-9ece-db4900e37c31",
      "_rev": "00000000e9c19d26"
    }
  }
]
```

The `_refResourceCollection` is the container that holds the role. The `_refResourceId` is the ID of the role. The `_ref` property is a resource path that is derived from the `_refResourceCollection` and the URL-encoded `_refResourceId`. `_refProperties` provides more information about the relationship.

### Important

Some of the examples in this documentation set use client-assigned IDs (such as `bjensen` and `scarter`) for the user objects because it makes the examples easier to read. If you create objects using the Admin UI, they are created with server-assigned IDs (such as `55ef0a75-f261-47e9-a72b-f5c61c32d339`). This particular example uses a client-assigned role ID that is the same as the role name. All other examples in this chapter use server-

assigned IDs. Generally, immutable server-assigned UUIDs are used for all managed objects in production environments.

## Manipulate Roles Over REST and in the UI

These sections show the REST calls to create, read, update, and delete managed roles, and to grant roles to users. For information about using roles to provision users to external systems, see "Use Assignments to Provision Users".

Note that the easiest way to work with roles is to use the Admin UI.

### + Create a Role

To create a role, send a PUT or POST request to the `/openidm/managed/role` context path. The following example creates a managed role named `employee`:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '{
  "name": "employee",
  "description": "Role granted to workers on the company payroll"
}' \
"http://localhost:8080/openidm/managed/role?action=create"
{
  "_id": "5790220a-719b-49ad-96a6-6571e63cbaf1",
  "_rev": "0000000079c6644f",
  "name": "employee",
  "description": "Role granted to workers on the company payroll"
}
```

This `employee` role has no corresponding *assignments*. Assignments are what enables the provisioning logic to the external system. Assignments are created and maintained as separate managed objects, and are referred to within role definitions. For more information about assignments, see "Use Assignments to Provision Users".

To create a role through the Admin UI:

1. Select Manage > Role and select New Role on the Role List page.
2. Enter a name and description for the new role and select Save.
3. Optionally, select Temporal Constraint to restrict the role grant to a set time period or Condition to define a query filter that will allow the role to be granted to members dynamically.

For more information, see "Use Temporal Constraints to Restrict Effective Roles" and Grant Roles Dynamically.

## + List Roles

To list all managed roles over REST, query the `openidm/managed/role` endpoint. The following example shows the `employee` role that you created in the previous example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/role?_queryFilter=true"
{
  "result": [
    {
      "_id": "5790220a-719b-49ad-96a6-6571e63cbaf1",
      "_rev": "0000000079c6644f",
      "name": "employee",
      "description": "Role granted to workers on the company payroll"
    }
  ],
  ...
}
```

To display all configured managed roles in the Admin UI, select Manage > Role.

If you have a large number of roles, select Advanced Filter to build a more complex query filter to display only the roles you want.

## + Grant Roles to a User

You grant roles to users through the relationship mechanism. Relationships are essentially references from one managed object to another; in this case from a user object to a role object. For more information about relationships, see "*Relationships Between Objects*".

You can grant roles *statically* or *dynamically*.

To grant a role statically, you must do one of the following:

- Update the value of the user's `roles` property to reference the role.
- Update the value of the role's `members` property to reference the user.

For more information see To Grant Roles Statically.

Dynamic role grants use the result of a condition or script to update a user's list of roles. For more information, see Grant Roles Dynamically.

## + To Grant Roles Statically

Grant a role to a user statically using the REST interface or the Admin UI as follows:

### Over REST

Use one of the following methods to grant a role to a user over REST:

- Update the user's `roles` property to refer to the role.

The following example grants the `employee` role (with ID `5790220a-719b-49ad-96a6-6571e63cbaf1`) to user `scarter`:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request PATCH \
--data '[
{
  "operation": "add",
  "field": "/roles/-",
  "value": {"_ref": "managed/role/5790220a-719b-49ad-96a6-6571e63cbaf1"}
}
]' \
"http://localhost:8080/openidm/managed/user/scarter"
{
  "_id": "scarter",
  "_rev": "000000003be825ce",
  "mail": "scarter@example.com",
  "givenName": "Steven",
  "sn": "Carter",
  "description": "Created By CSV",
  "userName": "scarter",
  "telephoneNumber": "1234567",
  "accountStatus": "active",
  "effectiveRoles": [
    {
      "_ref": "managed/role/5790220a-719b-49ad-96a6-6571e63cbaf1"
    }
  ],
  "effectiveAssignments": []
}
```

Note that `scarter`'s `effectiveRoles` attribute has been updated with a reference to the new role. For more information about effective roles and effective assignments, see "Effective Roles and Effective Assignments".

When you update a user's existing roles array, use the `-` special index to add the new value to the set. For more information, see *Set semantic arrays* in "Patch Operation: Add" in the *REST API Reference*.

- Update the role's `members` property to refer to the user.

The following sample command makes scarter a member of the **employee** role:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request PATCH \
--data '[
{
  "operation": "add",
  "field": "/members/-",
  "value": {"_ref": "managed/user/scarter"}
}
]' \
"http://localhost:8080/openidm/managed/role/5790220a-719b-49ad-96a6-6571e63cbaf1"
{
  "_id": "5790220a-719b-49ad-96a6-6571e63cbaf1",
  "_rev": "0000000079c6644f",
  "name": "employee",
  "description": "Role granted to workers on the company payroll"
}
```

The **members** property of a role is not returned by default in the output. To show all members of a role, you must specifically request the relationship properties (**\*\_ref**) in your query. The following example lists the members of the **employee** role (currently only scarter):

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/role/5790220a-719b-49ad-96a6-6571e63cbaf1?_fields=*_ref,name"
{
  "_id": "5790220a-719b-49ad-96a6-6571e63cbaf1",
  "_rev": "0000000079c6644f",
  "name": "employee",
  "assignments": [],
  "members": [
    {
      "_ref": "managed/user/scarter",
      "_refResourceCollection": "managed/user",
      "_refResourceId": "scarter",
      "_refProperties": {
        "_id": "7ad15a7b-6806-487b-900d-db569927f56d",
        "_rev": "0000000075e09cbf"
      }
    }
  ]
}
```

- You can replace an existing role grant with a new one by using the **replace** operation in your patch request.

The following command replaces scarter's entire `roles` entry (that is, overwrites any existing roles) with a single entry, the reference to the `employee` role (ID `5790220a-719b-49ad-96a6-6571e63cbaf1`):

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request PATCH \
--data '[
{
  "operation": "replace",
  "field": "/roles",
  "value": [
    { "_ref": "managed/role/5790220a-719b-49ad-96a6-6571e63cbaf1" }
  ]
}
]' \
"http://localhost:8080/openidm/managed/user/scarter"
{
  "_id": "scarter",
  "_rev": "00000000da112702",
  "mail": "scarter@example.com",
  "givenName": "Steven",
  "sn": "Carter",
  "description": "Created By CSV",
  "userName": "scarter",
  "telephoneNumber": "1234567",
  "accountStatus": "active",
  "effectiveRoles": [
    {
      "_ref": "managed/role/5790220a-719b-49ad-96a6-6571e63cbaf1"
    }
  ],
  "effectiveAssignments": []
}
```

## Using the Admin UI

Use one of the following UI methods to grant a role to a user:

- Update the user entry:
  1. Select Manage > User and select the user to whom you want to grant the role.
  2. Select the Provisioning Roles tab and select Add Provisioning Roles.
  3. Select the role from the dropdown list and select Add.
- Update the role entry:
  1. Select Manage > Role and select the role that you want to grant.
  2. Select the Role Members tab and select Add Role Members.

3. Select the user from the dropdown list and select Add.

#### + Grant Roles Dynamically

Grant a role *dynamically* by using one of the following methods:

- Use a condition, expressed as a query filter, in the role definition. If the condition is **true** for a particular member, that member is granted the role. Conditions can be used in both managed and internal roles.
- Use a custom script to define a more complex role-granting strategy.

#### + Grant a Role Based on a Condition

A role that is granted based on a defined condition is called a *conditional role*. To create a conditional role, include a query filter in the role definition.

##### Important

Properties that are used as the basis of a conditional role query *must* be configured as **searchable** and must be indexed in the repository configuration. To configure a property as **searchable**, update the schema in your **conf/managed.json** file. For more information, see "Create and Modify Object Types".

To create a conditional role by using the Admin UI, select Condition on the role Details page, then define the query filter that will be used to assess the condition.

To create a conditional role over REST, include the query filter as a value of the **condition** property in the role definition. The following example creates a role, **fr-employee**, that will be granted only to those users who live in France (whose **country** property is set to **FR**):

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '{
  "name": "fr-employee",
  "description": "Role granted to employees resident in France",
  "condition": "/country eq \"FR\""
}' \
"http://localhost:8080/openidm/managed/role?_action=create"
{
  "_id": "eb18a2e2-eele-4cca-83fb-5708a41db94f",
  "_rev": "000000004085704c",
  "name": "fr-employee",
  "description": "Role granted to employees resident in France",
  "condition": "/country eq \"FR\""
}
```

When a conditional role is created or updated, IDM automatically assesses all managed users, and recalculates the value of their `roles` property, if they qualify for that role. When a condition is removed from a role, that is, when the role becomes an unconditional role, all conditional grants are removed. So, users who were granted the role based on the condition, have that role removed from their `roles` property.

#### Caution

When a conditional role is defined in an existing data set, every user entry (including the mapped entries on remote systems) must be updated with the assignments implied by that conditional role. The time that it takes to create a new conditional role is impacted by the following items:

- The number of managed users affected by the condition.
- The number of assignments related to the conditional role.
- The average time required to provision updates to all remote systems affected by those assignments.

In a data set with a very large number of users, creating a new conditional role can therefore incur a significant performance cost when you create it. Ideally, you should set up your conditional roles at the beginning of your deployment to avoid performance issues later.

### + Grant a Role By Using Custom Scripts

The easiest way to grant roles dynamically is to use conditional roles, as described in [Grant a Role Based on a Condition](#). If your deployment requires complex conditional logic that cannot be achieved with a query filter, you can create a custom script to grant the role, as follows:

1. Create a `roles` directory in your project's `script` directory and copy the default effective roles script to that new directory:

```
mkdir project-dir/script/roles/  
cp /path/to/openidm/bin/defaults/script/roles/effectiveRoles.js project-dir/script/roles/
```

The new script will override the default effective roles script.

2. Modify the script to reference additional roles that have not been granted manually, or as the result of a conditional grant. The effective roles script calculates the grants that are in effect when the user is retrieved.

For example, the following addition to the `effectiveRoles.js` script grants the roles `dynamic-role1` and `dynamic-role2` to all active users (managed user objects whose `accountStatus` value is `active`). This example assumes that you have already created the managed roles, `dynamic-role1` (with ID `d2e29d5f-0d74-4d04-bcfe-b1daf508ad7c`) and `dynamic-role2` (with ID `709fed03-897b-4ff0-8a59-6faaa34e3af6`, and their corresponding assignments:

```
// This is the location to expand to dynamic roles,
// project role script return values can then be added via
// effectiveRoles = effectiveRoles.concat(dynamicRolesArray);

if (object.accountStatus === 'active') {
  effectiveRoles = effectiveRoles.concat([
    { "_ref": "managed/role/d2e29d5f-0d74-4d04-bcfe-b1daf508ad7c" },
    { "_ref": "managed/role/709fed03-897b-4ff0-8a59-6faaa34e3af6" }
  ]);
}
```

#### Note

For conditional roles, the user's **roles** property is updated if the user meets the condition. For custom scripted roles, the user's **effectiveRoles** property is calculated when the user is retrieved, and includes the dynamic roles according to the custom script.

If you make any of the following changes to a scripted role grant, you must perform a manual reconciliation of all affected users before assignment changes will take effect on an external system:

- If you create a new scripted role grant.
- If you change the definition of an existing scripted role grant.
- If you change any of the assignment rules for a role that is granted by a custom script.

### + Query a User's Roles

To query user roles over REST, query the user's **roles** property. The following example shows that scarter has been granted two roles—an **employee** role, and an **fr-employee** role:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/scarter/roles?_queryFilter=true&_fields=_ref/*,name"
{
  "result": [
    {
      "_id": "5a023862-654d-4d7f-b9d0-7c151b8dede5",
      "_rev": "00000000bba999c1",
      "_refResourceCollection": "managed/role",
      "_refResourceId": "b8783543-869a-4bd4-907e-9c1d89f826ae",
      "_refResourceRev": "0000000027a959cf",
      "name": "employee",
      "_ref": "managed/role/b8783543-869a-4bd4-907e-9c1d89f826ae",
      "_refProperties": {
```

```
{
  "_id": "5a023862-654d-4d7f-b9d0-7c151b8dede5",
  "_rev": "00000000baa999c1"
},
{
  "_id": "b281ffdf-477e-4211-a112-84476435bab2",
  "_rev": "00000000d612a248",
  "_refResourceCollection": "managed/role",
  "_refResourceId": "01ee6191-75d8-4d4b-9291-13a46592c57a",
  "_refResourceRev": "00000000cb0794d",
  "name": "fr-employee",
  "_ref": "managed/role/01ee6191-75d8-4d4b-9291-13a46592c57a",
  "_refProperties": {
    "_grantType": "conditional",
    "_id": "b281ffdf-477e-4211-a112-84476435bab2",
    "_rev": "00000000d612a248"
  }
},
...
}
```

Note that the `fr-employee` role indicates a `_grantType` of `conditional`. This property indicates *how* the role was granted to the user. If no `_grantType` is listed, the role was granted statically.

Querying a user's roles in this way *does not* return any roles that would be in effect as a result of a custom script, or of any temporal constraint applied to the role. To return a complete list of *all* the roles in effect at a specific time, query the user's `effectiveRoles` property, as follows:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/scarter?_fields=effectiveRoles"
```

Alternatively, to check which roles have been granted to a user, either statically or dynamically, look at the user's entry in the Admin UI:

1. Select Manage > User, then select the user whose roles you want to see.
2. Select the Provisioning Roles tab.
3. If you have a large number of managed roles, use the Advanced Filter option on the Role List page to build a custom query.

## + Delete a User's Roles

To remove a statically granted role from a user entry, do one of the following:

- Update the value of the user's `roles` property to remove the reference to the role.

- Update the value of the role's **members** property to remove the reference to that user.

You can use both of these methods over REST, or use the Admin UI.

### Important

A delegated administrator must use PATCH to add or remove relationships.

Roles that have been granted as the result of a condition can only be removed when the condition is changed or removed, or when the role itself is deleted.

## Over REST

Use one of the following methods to remove a role grant from a user:

- DELETE the role from the user's **roles** property, including the reference ID (the ID of the relationship between the user and the role) in the delete request.

The following example removes the **employee** role from user scarter. The role ID is **b8783543-869a-4bd4-907e-9c1d89f826ae**, but the ID required in the DELETE request is the reference ID (**5a023862-654d-4d7f-b9d0-7c151b8dede5**):

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request DELETE \
"http://localhost:8080/openidm/managed/user/scarter/roles/5a023862-654d-4d7f-b9d0-7c151b8dede5"
{
  "id": "5a023862-654d-4d7f-b9d0-7c151b8dede5",
  "_rev": "00000000b9a999c1",
  "_ref": "managed/role/b8783543-869a-4bd4-907e-9c1d89f826ae",
  "_refResourceCollection": "managed/role",
  "_refResourceId": "b8783543-869a-4bd4-907e-9c1d89f826ae",
  "_refProperties": {
    "id": "5a023862-654d-4d7f-b9d0-7c151b8dede5",
    "_rev": "00000000b9a999c1"
  }
}
```

- PATCH the user entry to remove the role from the array of roles, specifying the *value* of the role object in the JSON payload.

### Caution

When you remove a role in this way, you must include the *entire object* in the value, as shown in the following example:

```
curl \
--header "Content-type: application/json" \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
```

```
--request PATCH \
--data '[
{
  "operation": "remove",
  "field": "/roles",
  "value": {
    "_ref": "managed/role/b8783543-869a-4bd4-907e-9c1d89f826ae",
    "_refResourceCollection": "managed/role",
    "_refResourceId": "b8783543-869a-4bd4-907e-9c1d89f826ae",
    "_refProperties": {
      "_id": "5a023862-654d-4d7f-b9d0-7c151b8dede5",
      "_rev": "00000000baa999c1"
    }
  }
}
]' \
"http://localhost:8080/openidm/managed/user/scarter"
{
  "_id": "scarter",
  "_rev": "000000007b78257d",
  "mail": "scarter@example.com",
  "givenName": "Steven",
  "sn": "Carter",
  "description": "Created By CSV",
  "userName": "scarter",
  "telephoneNumber": "1234567",
  "accountStatus": "active",
  "effectiveRoles": [
    {
      "_ref": "managed/role/01ee6191-75d8-4d4b-9291-13a46592c57a"
    }
  ],
  "effectiveAssignments": [],
  "preferences": {
    "updates": false,
    "marketing": false
  },
  "country": "France"
}
```

- DELETE the user from the role's **members** property, including the reference ID (the ID of the relationship between the user and the role) in the delete request.

The following example first queries the members of the **employee** role, to obtain the ID of the relationship, then removes bjensen's membership from that role:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/role/b8783543-869a-4bd4-907e-9c1d89f826ae/members?
queryFilter=true"
{
  "result": [
    {
      "_id": "a5a4bf94-6425-4458-aae4-bbd6ad094f72",
      "_rev": "00000000c25d994a",

```

```

    "_ref": "managed/user/bjensen",
    "_refResourceCollection": "managed/user",
    "_refResourceId": "bjensen",
    "_refProperties": {
      "_id": "a5a4bf94-6425-4458-aae4-bbd6ad094f72",
      "_rev": "00000000c25d994a"
    }
  },
  ...
}

curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request DELETE \
"http://localhost:8080/openidm/managed/role/b8783543-869a-4bd4-907e-9c1d89f826ae/members/a5a4bf94-6425-4458-aae4-bbd6ad094f72"
{
  "_id": "a5a4bf94-6425-4458-aae4-bbd6ad094f72",
  "_rev": "00000000c25d994a",
  "_ref": "managed/user/bjensen",
  "_refResourceCollection": "managed/user",
  "_refResourceId": "bjensen",
  "_refProperties": {
    "_id": "a5a4bf94-6425-4458-aae4-bbd6ad094f72",
    "_rev": "00000000c25d994a"
  }
}

```

## Using the Admin UI

Use one of the following methods to remove a user's roles:

- Select Manage > User and select the user whose role or roles you want to remove.

Select the Provisioning Roles tab, select the role that you want to remove, then select Remove Selected Provisioning Roles.

- Select Manage > Role, and select the role whose members you want to remove.

Select the Role Members tab, select the member or members that that you want to remove, then select Remove Selected Role Members.

## + Delete a Role Definition

Delete a managed provisioning or authorization role over REST interface or by using the Admin UI.

To delete a role over the REST interface, simply delete that managed object. The following command deletes the **employee** role created in the previous section:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request DELETE \
"http://localhost:8080/openidm/managed/role/b8783543-869a-4bd4-907e-9c1d89f826ae"
{
  "_id": "b8783543-869a-4bd4-907e-9c1d89f826ae",
  "_rev": "0000000027a959cf",
  "privileges": [],
  "name": "employee",
  "description": "All employees"
}
```

### Note

You cannot delete a role that is currently granted to users. If you attempt to delete a role that is granted to a user (either over the REST interface, or by using the Admin UI), IDM returns an error. The following example attempts to remove a role that is still granted to a user:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request DELETE \
"http://localhost:8080/openidm/managed/role/01ee6191-75d8-4d4b-9291-13a46592c57a"
{
  "code": 409,
  "reason": "Conflict",
  "message": "Cannot delete a role that is currently granted"
}
```

To delete a role through the Admin UI, select Manage > Role, select the role you want to remove, then Delete Selected.

## Use Temporal Constraints to Restrict Effective Roles

Temporal constraints restrict the period that a role is effective. You can apply temporal constraints to managed and internal roles, and to role *grants* (for individual users).

For example, you might want a role, **contractors-2020**, to apply to all contract employees for the year 2020. In this case, you would set the temporal constraint on the role. Alternatively, you might want to assign a **contractors** role that applies to an individual user only for the period of their contract of employment.

The following examples show how to set temporal constraints on role definitions, and on individual role grants:

+ *Add a Temporal Constraint to a Role*

When you create a role, you can include a temporal constraint in the role definition that restricts the validity of the role, regardless of how that role is granted. Temporal constraints are expressed as a time interval in ISO 8601 date and time format. For more information on this format, see the [ISO 8601 standard](#).

The following example adds a `contractor` role over the REST interface. The role is effective from March 1st, 2020 to August 31st, 2020:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '{
  "name": "contractor",
  "description": "Role granted to contract workers for 2020",
  "temporalConstraints": [
    {
      "duration": "2020-03-01T00:00:00.000Z/2020-08-31T00:00:00.000Z"
    }
  ]
}' \
"http://localhost:8080/openidm/managed/role?action=create"
{
  "_id": "ed761370-b24f-4e21-8e58-a3230942da67",
  "_rev": "000000007429750e",
  "name": "contractor",
  "description": "Role granted to contract workers for 2020",
  "temporalConstraints": [
    {
      "duration": "2020-03-01T00:00:00.000Z/2020-08-31T00:00:00.000Z"
    }
  ]
}
```

This example specifies the time zone as Coordinated Universal Time (UTC) by appending `Z` to the time. If no time zone information is provided, the time zone is assumed to be local time. To specify a different time zone, include an offset (from UTC) in the format `±hh:mm`. For example, an interval of `2020-03-01T00:00:00.000-07:00/2020-08-31T00:00:00.000-07:00` specifies a time zone that is seven hours behind UTC.

When the period defined by the constraint has ended, the role object remains in the repository, but the effective roles script will not include the role in the list of effective roles for any user.

The following example assumes that user `scarter` has been granted a role `contractor-march`. A temporal constraint has been included in the `contractor-march` role definition, specifying that the role should be applicable only during the month of March 2020. At the end of this period, a query on `scarter`'s entry shows that his `roles` property still includes the `contractor-march` role (with ID `0face495-772d-4d36-a30d-8594618aba0d`), but his `effectiveRoles` property does not:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/scarter?_fields=_id,userName,roles,effectiveRoles"
{
  "_id": "scarter",
  "_rev": "00000000e5fdeb51",
  "userName": "scarter",
  "effectiveRoles": [],
  "roles": [
    {
      "_ref": "managed/role/0face495-772d-4d36-a30d-8594618aba0d",
      "_refResourceCollection": "managed/role",
      "_refResourceId": "0face495-772d-4d36-a30d-8594618aba0d",
      "_refProperties": {
        "_id": "5f41d5a5-19b4-4524-a4b1-445790ff14da",
        "_rev": "00000000cb339810"
      }
    }
  ]
}
```

The role is still in place but is no longer effective.

To restrict the period during which a role is valid by using the Admin UI, select Temporal Constraint on the role Details tab, then select a timezone offset relative to GMT and the start and end dates for the required period.

#### + Add a Temporal Constraint to a Role Grant

To restrict the validity of a role for individual users, apply a temporal constraint at the grant level, rather than as part of the role definition. In this case, the temporal constraint is taken into account per user, when the user's effective roles are calculated. Temporal constraints that are defined at the grant level can be different for each user who is a member of that role.

To apply a temporal constraint to a grant over the REST interface, include the constraint as one of the `_refProperties` of the relationship between the user and the role. The following example assumes a `contractor` role, with ID `ed761370-b24f-4e21-8e58-a3230942da67`. The command adds user `bjensen` as a member of that role, with a temporal constraint that specifies that she be a member of the role for one year only, from January 1st, 2020 to January 1st, 2021:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request PATCH \
--data '[
  {
    "operation": "add",
    "field": "/members/-",
    "value": {
      "_ref": "managed/user/bjensen",
      "_refProperties": {
        "temporalConstraints": [{"duration": "2020-01-01T00:00:00.000Z/2021-01-01T00:00:00.000Z"}]
      }
    }
  }
]' \
"http://localhost:8080/openidm/managed/role/ed761370-b24f-4e21-8e58-a3230942da67"
{
  "_id": "ed761370-b24f-4e21-8e58-a3230942da67",
  "_rev": "000000007429750e",
  "name": "contractor",
  "description": "Role granted to contract workers for 2020",
  "temporalConstraints": [
    {
      "duration": "2020-03-01T00:00:00.000Z/2020-08-31T00:00:00.000Z"
    }
  ]
}
```

A query on bjensen's roles property shows that the temporal constraint has been applied to this grant:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/bjensen/roles?_queryFilter=true"
{
  "result": [
    {
      "_id": "40600260-111d-4695-81f1-450365025784",
      "_rev": "00000000173daedb",
      "_ref": "managed/role/ed761370-b24f-4e21-8e58-a3230942da67",
      "_refResourceCollection": "managed/role",
      "_refResourceId": "ed761370-b24f-4e21-8e58-a3230942da67",
      "_refProperties": {
        "temporalConstraints": [
          {
            "duration": "2020-01-01T00:00:00.000Z/2021-01-01T00:00:00.000Z"
          }
        ]
      },
      "_id": "40600260-111d-4695-81f1-450365025784",
      "_rev": "00000000173daedb"
    }
  ],
  ...
}
```

To restrict the period during which a role grant is valid by using the Admin UI, set a temporal constraint when you add the member to the role.

For example, to specify that bjensen be added to a Contractor role only for the period of her employment contract, select Manage > Role, select the Contractor role, then select Add Role Members. On the Add Role Members screen, select bjensen from the list, then enable the Temporal Constraint, and specify the start and end date of her contract.

## Use Assignments to Provision Users

*Authorization roles* control access to IDM itself. *Provisioning roles* define rules for how attribute values are updated on external systems. These rules are configured through *assignments* that are attached to a provisioning role definition. The purpose of an assignment is to provision an attribute or set of attributes, based on an object's role membership.

The synchronization mapping configuration between two resources provides the basic account provisioning logic (how an account is mapped from a source to a target system). Role assignments provide additional provisioning logic that is not covered in the basic mapping configuration. The attributes and values that are updated by using assignments might include group membership, access to specific external resources, and so on. A group of assignments can collectively represent a *role*.

Assignment objects are created, updated, and deleted like any other managed object, and are attached to a role by using the relationships mechanism, in much the same way as a role is granted

to a user. Assignments are stored in the repository and are accessible at the context path `/openid/managed/assignment`.

This section describes how to manipulate assignments over the REST interface, and by using the Admin UI. When you have created an assignment, and attached it to a role definition, all user objects that reference that role definition will, as a result, reference the corresponding assignment in their `effectiveAssignments` attribute.

### + Create an Assignment

You can create assignments over the REST interface or by using the Admin UI:

#### Over REST

To create a new assignment over REST, send a PUT or POST request to the `/openid/managed/assignment` context path.

The following example creates a new managed assignment named `employee`. The JSON payload in this example shows the following:

- The assignment is applied for the mapping `managedUser_systemLdapAccounts`, so attributes will be updated on the external LDAP system specified in this mapping.
- The name of the attribute on the external system whose value will be set is `employeeType`, and its value will be set to `Employee`.
- When the assignment is applied during a sync operation, the attribute value `Employee` is added to any existing values for that attribute. When the assignment is removed (if the role is deleted, or if the user is no longer a member of that role), the attribute value `Employee` is removed from the values of that attribute.

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '{
  "name": "employee",
  "description": "Assignment for employees.",
  "mapping": "managedUser_systemLdapAccounts",
  "attributes": [
    {
      "name": "employeeType",
      "value": [
        "Employee"
      ],
      "assignmentOperation": "mergeWithTarget",
      "unassignmentOperation": "removeFromTarget"
    }
  ]
}' \
```

```
"http://localhost:8080/openidm/managed/assignment?_action=create"
{
  "_id": "1a6a3af3-024f-4cf1-b4f6-116b98053816",
  "_rev": "00000000b2329649",
  "name": "employee",
  "description": "Assignment for employees.",
  "mapping": "managedUser_systemLdapAccounts",
  "attributes": [
    {
      "name": "employeeType",
      "value": [
        "Employee"
      ],
      "assignmentOperation": "mergeWithTarget",
      "unassignmentOperation": "removeFromTarget"
    }
  ]
}
```

Note that at this stage, the assignment is not linked to any role, so no user can make use of the assignment. You must add the assignment to a role, as described in [Add an Assignment to a Role](#).

### Using the Admin UI

1. Select Manage > Assignment > New Assignment.
2. Enter a name and description for the new assignment.
3. Select the mapping to which the assignment should apply. The mapping indicates the target resource, that is, the resource on which the attributes specified in the assignment will be adjusted.

Select Save to add the assignment.

4. Select the Attributes tab and select the attribute or attributes whose values will be adjusted by this assignment. The attribute you select here will determine what is displayed next:
  - Regular text field—specify what the value of the attribute should be, when this assignment is applied.
  - Item button—specify a managed object type, such as an object, relationship, or string.
  - Properties button—specify additional information, such as an array of role references.
5. Select the assignment operation from the dropdown list:
  - **Merge With Target**: the attribute value will be added to any existing values for that attribute. This operation merges the existing value of the target object attribute with the value(s) from the assignment. If duplicate values are found (for attributes that take a list as a value), each value is included only once in the resulting target. This assignment

operation is used only with complex attribute values like arrays and objects, and does not work with strings or numbers.

- **Replace Target:** the attribute value will overwrite any existing values for that attribute. The value from the assignment becomes the authoritative source for the attribute.

6. Select the unassignment operation from the dropdown list:

- **Remove From Target:** the attribute value is removed from the system object when the user is no longer a member of the role, or when the assignment itself is removed from the role definition.
- **No Operation:** removing the assignment from the user's `effectiveAssignments` has no effect on the current state of the attribute in the system object.

7. (Optional) Select the Events tab to specify any scriptable events associated with this assignment.

The assignment and unassignment operations described in the previous step operate at the *attribute level*. That is, you specify what should happen with each attribute affected by the assignment when the assignment is applied to a user, or removed from a user.

The scriptable *On assignment* and *On unassignment* events operate at the *assignment level*, rather than the attribute level. Define scripts here to apply additional logic or operations that should be performed when a user (or other object) receives or loses an entire assignment. This logic can be anything that is not restricted to an operation on a single attribute.

For information about the variables available to these scripts, see "Variables Available to Role Assignment Scripts" in the *Scripting Guide*.

8. Select the Roles tab to attach this assignment to an existing role definition.

### + Add an Assignment to a Role

After you have created a role, and an assignment, you create a *relationship* between the assignment and the role, in much the same way as a user references a role.

Update a role definition to include one or more assignments over the REST interface, or by using the Admin UI:

#### Over REST

Update the role definition to include a reference to the ID of the assignment in the `assignments` property of the role. The following example adds the `employee` assignment (ID `1a6a3af3-024f-4cf1-b4f6-116b98053816`) to an existing `employee` role (ID `2243f5f8-ed75-4c3b-b4b3-058d5c58fbb4`):

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request PATCH \
--data '[
{
  "operation": "add",
  "field": "/assignments/-",
  "value": { "_ref": "managed/assignment/1a6a3af3-024f-4cf1-b4f6-116b98053816" }
}]' \
"http://localhost:8080/openidm/managed/role/2243f5f8-ed75-4c3b-b4b3-058d5c58fbb4"
{
  "_id": "2243f5f8-ed75-4c3b-b4b3-058d5c58fbb4",
  "_rev": "00000000e85263c7",
  "privileges": [],
  "name": "employee",
  "description": "Roll granted to all permanent employees"
}
```

To check that the assignment was added successfully, query the role's `assignments` property:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/role/2243f5f8-ed75-4c3b-b4b3-058d5c58fbb4/assignments?_queryFilter=true&_fields=_ref/*,name,assignments"
{
  "result": [
    {
      "_id": "d15822f0-05bc-464a-927d-8e5018a234d3",
      "_rev": "0000000010eea343",
      "_refResourceCollection": "managed/assignment",
      "_refResourceId": "1a6a3af3-024f-4cf1-b4f6-116b98053816",
      "_refResourceRev": "00000000b2329649",
      "name": "employee",
      "_ref": "managed/assignment/1a6a3af3-024f-4cf1-b4f6-116b98053816",
      "_refProperties": {
        "_id": "d15822f0-05bc-464a-927d-8e5018a234d3",
        "_rev": "0000000010eea343"
      }
    }
  ],
  ...
}
```

Note that the `assignments` property references the assignment that you created in the previous step.

To remove an assignment from a role definition, remove the reference to the assignment from the role's `assignments` property.

## Using the Admin UI

1. Select Manage > Role and select the role to which you want to add an assignment.
2. Select the Managed Assignments tab and select Add Managed Assignments.
3. Select the assignment that you want to add to the role, then select Add.

## + Delete an Assignment

Delete assignments over the REST interface, or by using the Admin UI:

### Over REST

To delete an assignment over the REST interface, simply delete that object. The following example deletes the **employee** assignment created in the previous example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request DELETE \
"http://localhost:8080/openidm/managed/assignment/1a6a3af3-024f-4cf1-b4f6-116b98053816"
{
  "_id": "1a6a3af3-024f-4cf1-b4f6-116b98053816",
  "_rev": "00000000b2329649",
  "name": "employee",
  "description": "Assignment for employees.",
  "mapping": "managedUser_systemLdapAccounts",
  "attributes": [
    {
      "name": "employeeType",
      "value": [
        "Employee"
      ],
      "assignmentOperation": "mergeWithTarget",
      "unassignmentOperation": "removeFromTarget"
    }
  ]
}
```

### Note

You *can* delete an assignment, even if it is referenced by a managed role. When the assignment is removed, any users to whom the corresponding roles were granted will no longer have that assignment in their list of **effectiveAssignments**. For more information about effective roles and effective assignments, see "Effective Roles and Effective Assignments".

## Using the Admin UI

To delete an assignment by using the Admin UI, select Manage > Assignment.

Select the assignment you want to remove, then select Delete.

### Important

If you have mapped roles and assignments to properties on a target system, and you are preloading the result set into memory, make sure that your `targetQuery` returns the mapped property. For example, if you have mapped a specific role to the `ldapGroups` property on the target system, the target query must include the `ldapGroups` property when it returns the object.

The following mapping excerpt indicates that the target query must return the `_id` of the object as well as its `ldapGroups` property:

```
"targetQuery": {
  "_queryFilter": true,
  "_fields": "_id,ldapGroups"
}
```

For more information about preloading the result set for reconciliation operations, see "Improving Reconciliation Query Performance" in the *Synchronization Guide*.

## Effective Roles and Effective Assignments

*Effective roles* and *effective assignments* are virtual properties of a user object. Their values are calculated by IDM, using relationships between related objects to know when to recalculate when changes occur. The relationships between objects are configured using the `notify`, `notifySelf`, and `notifyRelationships` settings for `managed/user`, `managed/role`, and `managed/assignment`. Which related objects to traverse for calculation is configured using `queryConfig`. Calculation or recalculation is triggered when the roles or assignments for a managed user are added, removed, or changed, including by changes from temporal constraints, and notification of that change is sent to the related objects.

The following excerpt of a `managed.json` file shows how these two virtual properties are constructed for each managed user object:

```
"effectiveRoles" : {
  "type" : "array",
  "title" : "Effective Roles",
  "description" : "Effective Roles",
  "viewable" : false,
  "returnByDefault" : true,
  "isVirtual" : true,
  "queryConfig" : {
    "referencedRelationshipFields" : ["roles"]
  },
  "usageDescription" : "",
  "isPersonal" : false,
  "items" : {
    "type" : "object",
    "title" : "Effective Roles Items"
  }
}
```

```

},
"effectiveAssignments" : {
  "type" : "array",
  "title" : "Effective Assignments",
  "description" : "Effective Assignments",
  "viewable" : false,
  "returnByDefault" : true,
  "isVirtual" : true,
  "queryConfig" : {
    "referencedRelationshipFields" : ["roles", "assignments"],
    "referencedObjectFields" : ["*"]
  },
  "usageDescription" : "",
  "isPersonal" : false,
  "items" : {
    "type" : "object",
    "title" : "Effective Assignments Items"
  }
}
}

```

When a role references an assignment, and a user references the role, that user automatically references the assignment in its list of effective assignments.

`effectiveRoles` uses the `roles` relationship to calculate the grants that are currently in effect, including any qualified by temporal constraints.

`effectiveAssignments` uses the `roles` relationship, and the `assignments` relationship for each role, to calculate the current assignments in effect for that user. The synchronization engine reads the calculated value of the `effectiveAssignments` attribute when it processes the user. The target system is updated according to the configured `assignmentOperation` for each assignment.

When a user's roles or assignments are updated, IDM calculates the `effectiveRoles` and `effectiveAssignments` for that user based on the current value of the user's `roles` property, and the `assignments` property of any roles referenced by the `roles` property. The previous set of examples showed the creation of a role `employee` that referenced an assignment `employee` and was granted to user `bjensen`. Querying that user entry would show the following effective roles and effective assignments:

```

curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/bjensen?
_fields=username,roles,effectiveRoles,effectiveAssignments"
{
  "_id": "ca8855fd-a404-42c7-88b7-02f8a8a825b2",
  "_rev": "00000000081eebela",
  "userName": "bjensen",
  "effectiveRoles": [
    {
      "_ref": "managed/role/2243f5f8-ed75-4c3b-b4b3-058d5c58fbb4"
    }
  ],
  "effectiveAssignments": [
    {

```

```

    "name": "employee",
    "description": "Assignment for employees.",
    "mapping": "managedUser_systemLdapAccounts",
    "attributes": [
      {
        "assignmentOperation": "mergeWithTarget",
        "name": "employeeType",
        "unassignmentOperation": "removeFromTarget",
        "value": [
          "employee"
        ]
      }
    ],
    "_rev": "0000000087d5a9a5",
    "_id": "46befacf-a7ad-4633-864d-d93abfa561e9"
  }
],
"roles": [
  {
    "_ref": "managed/role/2243f5f8-ed75-4c3b-b4b3-058d5c58fbb4",
    "_refResourceCollection": "managed/role",
    "_refResourceId": "2243f5f8-ed75-4c3b-b4b3-058d5c58fbb4",
    "_refProperties": {
      "_id": "93552530-10fa-49a4-865f-c942dff2801",
      "_rev": "0000000081ed9f2b"
    }
  }
]
}

```

In this example, synchronizing the managed/user repository with the external LDAP system defined in the mapping populates user bjensen's `employeeType` attribute in LDAP with the value `employee`.

## Roles and Relationship Change Notification

Before you read this section, see ["Configure Relationship Change Notification"](#) to understand the `notify` and `notifyRelationships` properties, and how change notification works for relationships in general. In the case of roles, the change notification configuration exists to ensure that managed users are notified when any of the relationships that link users, roles, and assignments are manipulated (that is, created, updated, or deleted).

Consider the situation where a user has role `R`. A new assignment `A` is created that references role `R`. Ultimately, we want to notify all users that have role `R` so that their reconciliation state will reflect any attributes in the new assignment `A`. We achieve this notification with the following configuration:

In the managed object schema, the `assignment` object definition has a `roles` property that includes a `resourceCollection`. The `path` of this resource collection is `managed/role` and `"notify": true` for the resource collection:

```
{
  "name" : "assignment",
  "schema" : {
    ...
    "properties" : {
      ...
      "roles" : {
        ...
        "items" : {
          ...
          "resourceCollection" : [
            {
              "notify" : true,
              "path" : "managed/role",
              "label" : "Role",
              "query" : {
                "queryFilter" : "true",
                "fields" : [
                  "name"
                ]
              }
            }
          ]
        }
      }
    }
    ...
  }
}
```

With this configuration, when assignment **A** is created, with a reference to role **R**, role **R** is notified of the change. However, we still need to propagate that notification to any **users** who are **members** of role **R**. To do this, we configure the **role** object as follows:

```
{
  "name" : "role",
  "schema" : {
    ...
    "properties" : {
      ...
      "assignments" : {
        ...
        "notifyRelationships" : ["members"]
      }
    }
    ...
  }
}
```

When role **R** is notified of the creation of a new relationship to assignment **A**, the notification is propagated through the **assignments** property. Because **"notifyRelationships" : ["members"]** is set on the **assignments** property, the notification is propagated across role **R** to all members of role **R**.

## Managed Role Script Hooks

Like any other managed object, you can use script hooks to configure role behavior. The default role definition in **conf/managed.json** includes an **onDelete** hook that calls a script to prevent the role from being deleted if it is currently assigned to users:

```
{
  "name" : "role",
  "onDelete" : {
    "type" : "text/javascript",
    "file" : "roles/onDelete-roles.js"
  },
  ...
}
```

## Use Groups to Control Access to IDM

A user's access to IDM is based on one or more *authorization roles*. Authorization roles are cumulative, and are calculated for a user in the following order:

1. Roles set specifically in the user's `userRoles` property
2. Group roles—based on group membership in an external system

Group roles are controlled with the following properties in the `${authConfig}`:

- `groupMembership`: the property on the external system that represents group membership. In a DS directory server, that property is `ldapGroups` by default. In an Active Directory server, the property is `memberOf` by default. For example:

```
"groupMembership" : "ldapGroups"
```

Note that the value of the `groupMembership` property must be the ICF property name defined in the provisioner file, rather than the property name on the external system.

- `groupRoleMapping`: a mapping between an IDM role and a group on the external system. Setting this property ensures that if a user authenticates through pass-through authentication, they are given specific IDM roles depending on their membership in groups on the external system. In the following example, users who are members of the group `cn=admins,ou=Groups,dc=example,dc=com` are given the internal `openidm-admin` role when they authenticate:

```
"groupRoleMapping" : {
  "internal/role/openidm-admin" : ["cn=admins,ou=Groups,dc=example,dc=com"]
}
```

- `groupComparisonMethod`: the method used to check whether the authenticated user's group membership matches one of the groups mapped to an IDM role (in the `groupRoleMapping` property).

The `groupComparisonMethod` can be one of the following:

- `equals`: a case-sensitive equality check
- `caseInsensitive`: a case-insensitive equality check

- `ldap`: a case-insensitive and whitespace-insensitive equality check. Because LDAP directories do not take case or whitespace into account in group DN's, you must set the `groupComparisonMethod` if you are using pass-through authentication with an LDAP directory.

**Note**

To control access to *external systems*, use *provisioning roles* and assignments, as described in "Use Assignments to Provision Users".

## Chapter 4

# Use Policies to Validate Data

IDM provides an extensible policy service that enables you to apply specific validation requirements to various components and properties. This chapter describes the policy service, and provides instructions on configuring policies for managed objects.

The policy service provides a REST interface for reading policy requirements and validating the properties of components against configured policies. Objects and properties are validated automatically when they are created, updated, or patched. Policies are generally applied to user passwords, but can also be applied to any managed or system object, and to internal user objects.

The policy service enables you to accomplish the following tasks:

- Read the configured policy requirements of a specific component.
- Read the configured policy requirements of all components.
- Validate a component object against the configured policies.
- Validate the properties of a component against the configured policies.

The router service limits policy application to managed, system, and internal user objects. To apply policies to additional objects, such as the audit service, you must modify your project's `conf/router.json` file. For more information about the router service, see "*Router Configuration*" in the *Scripting Guide*.

A default policy applies to all managed objects. You can configure this default policy to suit your requirements, or you can extend the policy service by supplying your own scripted policies.

## Default Policy for Managed Objects

Policies applied to managed objects are configured in two files:

- A policy script file (`openidm/bin/defaults/script/policy.js`) that defines each policy and specifies how policy validation is performed. For more information, see "Policy Script File".
- A managed object policy configuration element, defined in your project's `conf/managed.json` file, that specifies which policies are applicable to each managed resource. For more information, see "Policy Configuration Element".

### Note

The configuration for determining which policies apply to resources *other than managed objects* is defined in your project's `conf/policy.json` file. The default `policy.json` file includes policies that are applied to internal user objects, but you can extend the configuration in this file to apply policies to system objects.

## Policy Script File

The policy script file (`openidm/bin/defaults/script/policy.js`) separates policy configuration into two parts:

- A policy configuration object, which defines each element of the policy. For more information, see "Policy Configuration Objects".
- A policy implementation function, which describes the requirements that are enforced by that policy.

Together, the configuration object and the implementation function determine whether an object is valid in terms of the applied policy. The following excerpt of a policy script file configures a policy that specifies that the value of a property must contain a certain number of capital letters:

```
...
{
  "policyId": "at-least-X-capitals",
  "policyExec": "atLeastXCapitalLetters",
  "clientValidation": true,
  "validateOnlyIfPresent": true,
  "policyRequirements": ["AT_LEAST_X_CAPITAL_LETTERS"]
},
...

policyFunctions.atLeastXCapitalLetters = function(fullObject, value, params, property) {
  var isRequired = _.find(this.failedPolicyRequirements, function (fpr) {
    return fpr.policyRequirement === "REQUIRED";
  }),
  isString = (typeof(value) === "string"),
  valuePassesRegexp = (function (v) {
    var test = isString ? v.match(/[A-Z]/g) : null;
    return test !== null && test.length >= params.numCaps;
  })(value));

  if ((isRequired || isString) && !valuePassesRegexp) {
    return [ { "policyRequirement" : "AT_LEAST_X_CAPITAL_LETTERS", "params" : {"numCaps":
params.numCaps} } ];
  }

  return [];
}
...
```

To enforce user passwords that contain at least one capital letter, the `policyId` from the preceding example is applied to the appropriate resource (`managed/user/*`). The required number of capital letters

is defined in the policy configuration element of the managed object configuration file (see "Policy Configuration Element").

## Policy Configuration Objects

Each element of the policy is defined in a policy configuration object. The structure of a policy configuration object is as follows:

```
{
  "policyId": "minimum-length",
  "policyExec": "minLength",
  "clientValidation": true,
  "validateOnlyIfPresent": true,
  "policyRequirements": ["MIN_LENGTH"]
}
```

- **policyId** - a unique ID that enables the policy to be referenced by component objects.
- **policyExec** - the name of the function that contains the policy implementation. For more information, see "Policy Implementation Functions".
- **clientValidation** - indicates whether the policy decision can be made on the client. When **"clientValidation": true**, the source code for the policy decision function is returned when the client requests the requirements for a property.
- **validateOnlyIfPresent** - notes that the policy is to be validated only if it exists.
- **policyRequirements** - an array containing the policy requirement ID of each requirement that is associated with the policy. Typically, a policy will validate only one requirement, but it can validate more than one.

## Policy Implementation Functions

Each policy ID has a corresponding policy implementation function that performs the validation. Implementation functions take the following form:

```
function <name>(fullObject, value, params, propName) {
  <implementation_logic>
}
```

- **fullObject** is the full resource object that is supplied with the request.
- **value** is the value of the property that is being validated.
- **params** refers to the **params** array that is specified in the property's policy configuration.
- **propName** is the name of the property that is being validated.

The following example shows the implementation function for the **required** policy:

```
function required(fullObject, value, params, propName) {  
  if (value === undefined) {  
    return [ { "policyRequirement" : "REQUIRED" } ];  
  }  
  return [];  
}
```

## Default Policy Reference

IDM includes the following default policies and parameters:

| Policy Id  | Parameters                   |                                 |
|--|------------------------------|---------------------------------|
| <b>required</b><br>The property is required; not optional.                 |                              |                                 |
| <b>not-empty</b><br>The property can't be empty.                           |                              |                                 |
| <b>unique</b><br>The property must be unique.                              |                              |                                 |
| <b>valid-username</b><br>Tests for uniqueness and internal user conflicts. |                              |                                 |
| <b>no-internal-user-conflict</b><br>Tests for internal user conflicts.     |                              |                                 |
| <b>regexMatches</b><br>Matches a regular expression.                       | <b>regex</b><br><b>flags</b> | The regular expression pattern. |
| <b>valid-type</b><br>Tests for the specified types.                        | <b>types</b>                 |                                 |
| <b>valid-query-filter</b><br>Tests for a valid query filter.               |                              |                                 |
| <b>valid-array-items</b><br>Tests for valid array items.                   |                              |                                 |
| <b>valid-date</b><br>Tests for a valid date.                               |                              |                                 |
| <b>valid-email-address-format</b><br>Tests for a valid email address.      |                              |                                 |

| Policy Id   | Parameters                    |   |
|---|-------------------------------|---|
| <code>valid-name-format</code><br>Tests for a valid name format.  |                               |   |
| <code>valid-phone-format</code><br>Tests for a valid phone number format.                                     |                               |   |
| <code>at-least-X-capital</code><br>The property must contain the minimum specified number of capital letters. | <code>numCaps</code>          | Minimum number of capital letters.  |
| <code>at-least-X-numbers</code><br>The property must contain the minimum specified number of numbers.         | <code>numNums</code>          | Minimum number of numbers.  |
| <code>validNumber</code><br>Tests for a valid number.   |                               |   |
| <code>minimumNumber</code><br>The property value must be greater than the <code>minimum</code> .              | <code>minimum</code>          | The minimum value.  |
| <code>maximumNumber</code><br>The property value must be less than the <code>maximum</code> .                 | <code>maximum</code>          | The maximum value.  |
| <code>minimum-length</code><br>The property's minimum string length.  | <code>minLength</code>        | The minimum string length.  |
| <code>maximum-length</code><br>The property's maximum string length.  | <code>maxLength</code>        | The maximum string length.  |
| <code>cannot-contain-others</code><br>The property cannot contain values of the specified fields.             | <code>disallowedFields</code> | A comma-separated list of the fields to check against. For example, the default managed user password policy specifies <code>userName, givenName, sn</code> as disallowed fields. |
| <code>cannot-contain-characters</code><br>The property cannot contain the specified characters.               | <code>forbiddenChars</code>   | A comma-separated list of disallowed characters. For example, the default managed user <code>userName</code> policy specifies <code>/</code> as a disallowed character.           |
| <code>cannot-contain-duplicates</code><br>The property cannot contain duplicate characters.                   |                               |   |
| <code>mapping-exists</code>   |                               |   |

| Policy Id                                   | Parameters |  |
|---|------------|--|
| A sync mapping must exist for the property. |            |  |
| <code>valid-permissions</code>              |            |  |
| Tests for valid permissions.                |            |  |
| <code>valid-accessFlags-object</code>       |            |  |
| Tests for valid access flags.               |            |  |
| <code>valid-privilege-path</code>           |            |  |
| Tests for a valid privilege path.           |            |  |
| <code>valid-temporal-constraints</code>     |            |  |
| Tests for valid temporal constraints.       |            |  |

## Policy Configuration Element

The configuration of a managed object property (in the `managed.json` file) can include a `policies` element that specifies how policy validation should be applied to that property. The following excerpt of the default `managed.json` file shows how policy validation is applied to the `password` and `_id` properties of a managed/user object:

```
{
  "name" : "user",
  "schema" : {
    "id" : "http://jsonschema.net",
    "properties" : {
      "_id" : {
        "description" : "User ID",
        "type" : "string",
        "viewable" : false,
        "searchable" : false,
        "userEditable" : false,
        "usageDescription" : "",
        "isPersonal" : false,
        "policies" : [
          {
            "policyId" : "cannot-contain-characters",
            "params" : {
              "forbiddenChars" : [
                "/"
              ]
            }
          }
        ]
      },
      "password" : {
        "title" : "Password",
        "description" : "Password",
        "type" : "string",
        "viewable" : false,
```

```

    "searchable" : false,
    "userEditable" : true,
    "encryption" : {
      "purpose" : "idm.password.encryption"
    },
    "scope" : "private",
    "isProtected" : true,
    "usageDescription" : "",
    "isPersonal" : false,
    "policies" : [
      {
        "policyId" : "minimum-length",
        "params" : {
          "minLength" : 8
        }
      },
      {
        "policyId" : "at-least-X-capitals",
        "params" : {
          "numCaps" : 1
        }
      },
      {
        "policyId" : "at-least-X-numbers",
        "params" : {
          "numNums" : 1
        }
      },
      {
        "policyId" : "cannot-contain-others",
        "params" : {
          "disallowedFields" : [
            "userName",
            "givenName",
            "sn"
          ]
        }
      }
    ]
  }
}

```

Note that the policy for the `_id` property references the function `cannot-contain-characters`, that is defined in the `policy.js` file. The policy for the `password` property references the functions `minimum-length`, `at-least-X-capitals`, `at-least-X-numbers`, and `cannot-contain-others`, that are defined in the `policy.js` file. The parameters that are passed to these functions (number of capitals required, and so forth) are specified in the same element.

## Validate Managed Object Data Types

The `type` property of a managed object specifies the data type of that property, for example, `array`, `boolean`, `integer`, `number`, `null`, `object`, or `string`. For more information about data types, see the [JSON Schema Primitive Types](#) section of the JSON Schema standard.

The **type** property is subject to policy validation when a managed object is created or updated. Validation fails if data does not match the specified **type**, such as when the data is an **array** instead of a **string**. The **valid-type** policy in the default **policy.js** file enforces the match between property values and the **type** defined in the **managed.json** file.

IDM supports multiple valid property types. For example, you might have a scenario where a managed user can have more than one telephone number, or a **null** telephone number (when the user entry is first created and the telephone number is not yet known). In such a case, you could specify the accepted property type as follows in your **managed.json** file:

```
"telephoneNumber" : {
  "type" : "string",
  "title" : "Telephone Number",
  "description" : "Telephone Number",
  "viewable" : true,
  "userEditable" : true,
  "pattern" : "^\\+?([0-9\\- \\(\\)])*$",
  "usageDescription" : "",
  "isPersonal" : true,
  "policies" : [
    {
      "policyId" : "minimum-length",
      "params" : {
        "minLength" : 1
      }
    },
    {
      "policyId": "maximum-length",
      "params": {
        "maxLength": 255
      }
    }
  ]
}
```

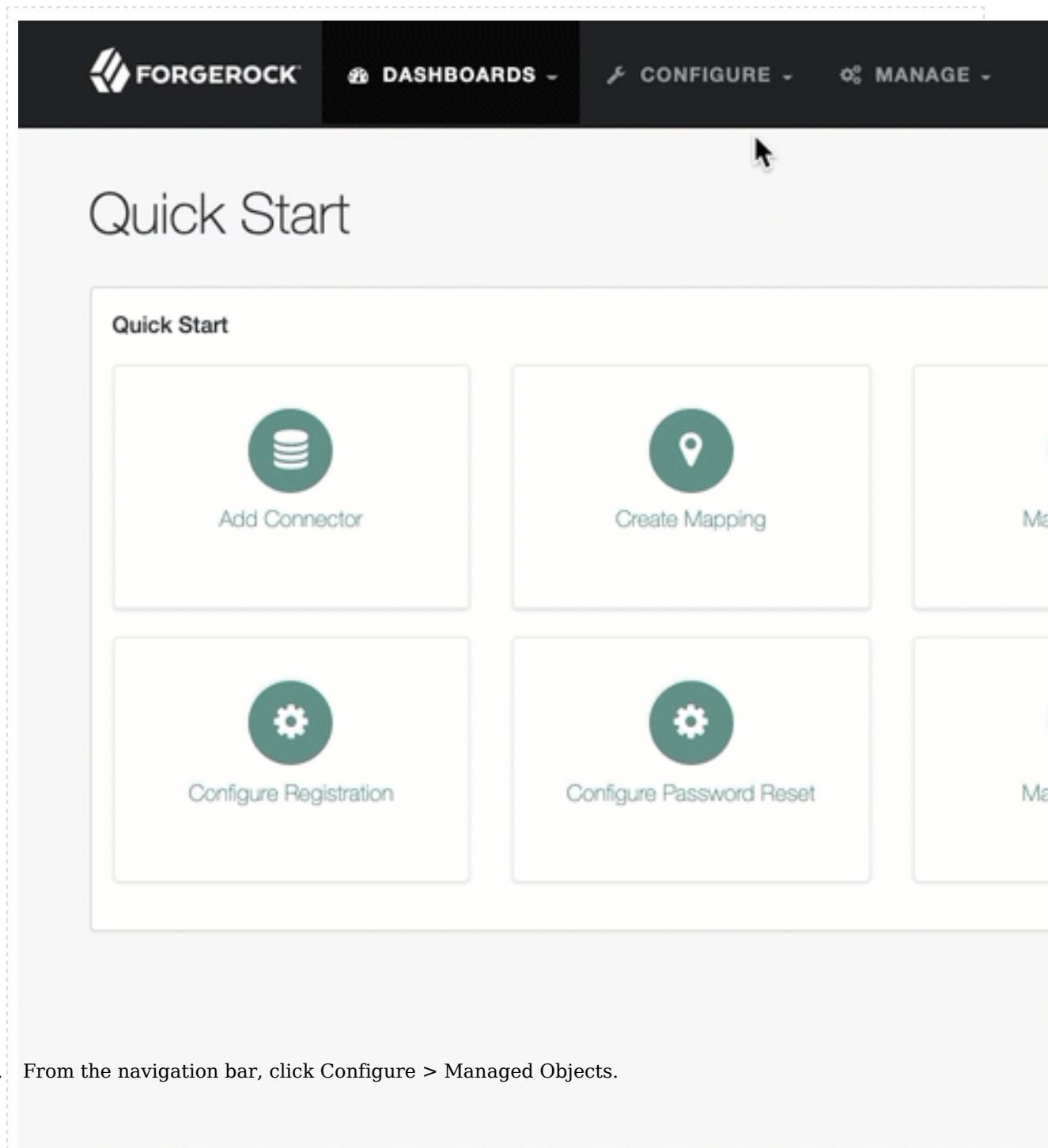
In this case, the **valid-type** policy from the **policy.js** file checks the telephone number for an accepted **type** and **pattern**, either for a real telephone number or a **null** entry.

## Configure Policy Validation Using the Admin UI

To configure policy validation for a managed object type using the Admin UI, update the configuration of the object type—a high-level overview:

1. Go to the managed object, and edit or create a property.
2. Click the Validation tab, and add the policy.

+ *Show Me*



1. From the navigation bar, click Configure > Managed Objects.

2. On the Managed Objects page, edit or create a managed object.
3. On the Managed Object *NAME* page, do one of the following:
  - To edit an existing property, click the property.
  - To create a property, click Add a Property, enter the required information, and click Save.
  - Now click the property.
4. From the Validation tab, click Add Policy.
5. In the Add/Edit Policy window, enter information in the following fields, and click Add or Save:

### Policy Id

Refers to the unique `PolicyId` in the `policy.js` file. For a list of the default policies, see "Default Policy Reference".

### Parameter Name

Refers to the parameters for the `PolicyId`. For a list of the default policy parameters, see "Default Policy Reference".

### Value

The parameter's value to validate.

### Important

Be cautious when using Validation Policies. If a policy relates to an array of relationships, such as between a user and multiple devices, Return by Default should always be set to `false`. You can verify this in your `${managedConfig}`. Any managed object that has `items` of `"type" : "relationship"`, must also have `"returnByDefault" : false`.

## Extend the Policy Service

To extend the policy service, add custom scripted policies, or add policies that are applied only under certain conditions.

- "Add Custom Scripted Policies"
- "Add Conditional Policy Definitions"

## Add Custom Scripted Policies

If your deployment requires additional validation functionality that is not supplied by the default policies, you can add your own policy scripts to your project's `script` directory, and reference them from your project's `conf/policy.json` file.

Do not modify the default policy script file (`openidm/bin/defaults/script/policy.js`) as doing so might result in interoperability issues in a future release. To reference additional policy scripts, set the `additionalFiles` property `conf/policy.json`.

The following example creates a custom policy that rejects properties with null values. The policy is defined in a script named `mypolicy.js`:

```
var policy = {  "policyId" : "notNull",
               "policyExec" : "notNull",
               "policyRequirements" : ["NOT_NULL"]}
}

addPolicy(policy);

function notNull(fullObject, value, params, property) {
  if (value == null) {
    var requireNotNull = [
      {"policyRequirement": "NOT_NULL"}
    ];
    return requireNotNull;
  }
  return [];
}
```

The `mypolicy.js` policy is referenced in the `policy.json` configuration file as follows:

```
{
  "type" : "text/javascript",
  "file" : "policy.js",
  "additionalFiles" : ["script/mypolicy.js"],
  "resources" : [
    {
      ...
    }
  ]
}
```

### Note

In cases where you are using the Admin UI, both `policy.js` and `mypolicy.js` will be run within the client, and then again by the the server. When creating new policies, be aware that these policies may be run in both contexts.

## Add Conditional Policy Definitions

You can extend the policy service to support policies that are applied only under specific conditions. To apply a conditional policy to managed objects, add the policy to your project's `managed.json` file. To apply a conditional policy to other objects, add it to your project's `policy.json` file.

The following excerpt of a `managed.json` file shows a sample conditional policy configuration for the `"password"` property of managed user objects. The policy indicates that sys-admin users have a more lenient password policy than regular employees:

```
{
  "objects" : [
    {
      "name" : "user",
      ...
      "properties" : {
        ...
        "password" : {
          "title" : "Password",
          "type" : "string",
          ...
          "conditionalPolicies" : [
            {
              "condition" : {
                "type" : "text/javascript",
                "source" : "(fullObject.org === 'sys-admin')",
              },
              "dependencies" : [ "org" ],
              "policies" : [
                {
                  "policyId" : "max-age",
                  "params" : {
                    "maxDays" : ["90"]
                  }
                }
              ]
            },
            {
              "condition" : {
                "type" : "text/javascript",
                "source" : "(fullObject.org === 'employees')",
              },
              "dependencies" : [ "org" ],
              "policies" : [
                {
                  "policyId" : "max-age",
                  "params" : {
                    "maxDays" : ["30"]
                  }
                }
              ]
            }
          ],
          "fallbackPolicies" : [
            {
              "policyId" : "max-age",
              "params" : {
                "maxDays" : ["7"]
              }
            }
          ]
        }
      }
    },
    ...
  ]
}
```

To understand how a conditional policy is defined, examine the components of this sample policy. For more information on the policy function, see "Policy Implementation Functions".

There are two distinct scripted conditions (defined in the `condition` elements). The first condition asserts that the user object, contained in the `fullObject` argument, is a member of the `sys-admin` org. If that assertion is true, the `max-age` policy is applied to the `password` attribute of the user object, and the maximum number of days that a password may remain unchanged is set to `90`.

The second condition asserts that the user object is a member of the `employees` org. If that assertion is true, the `max-age` policy is applied to the `password` attribute of the user object, and the maximum number of days that a password may remain unchanged is set to `30`.

In the event that neither condition is met (the user object is not a member of the `sys-admin` org or the `employees` org), an optional fallback policy can be applied. In this example, the fallback policy also references the `max-age` policy and specifies that for such users, their password must be changed after 7 days.

The `dependencies` field prevents the condition scripts from being run at all, if the user object does not include an `org` attribute.

#### Note

This example assumes that a custom `max-age` policy validation function has been defined, as described in "Add Custom Scripted Policies".

#### Tip

These scripted conditions do not apply to progressive profiling. For more information, see "Custom Progressive Profile Conditions" in the *Self-Service Reference*.

## Disable Policy Enforcement

*Policy enforcement* is the automatic validation of data when it is created, updated, or patched. In certain situations you might want to disable policy enforcement temporarily. You might, for example, want to import existing data that does not meet the validation requirements with the intention of cleaning up this data at a later stage.

You can disable policy enforcement by setting `openidm.policy.enforcement.enabled` to `false` in your `resolver/boot.properties` file. This setting disables policy enforcement in the back-end only, and has no impact on direct policy validation calls to the Policy Service (which the UI makes to validate input fields). So, with policy enforcement disabled, data added directly over REST is not subject to validation, but data added with the UI is still subject to validation.

You should not disable policy enforcement permanently, in a production environment.

# Manage Policies Over REST

Manage the policy service over the REST interface at the `openidm/policy` endpoint.

- "List the Defined Policies"
- "Validate Objects and Properties Over REST"

## List the Defined Policies

The following REST call displays a list of all the policies defined in `policy.json` (policies for objects other than managed objects). The policy objects are returned in JSON format, with one object for each defined policy ID:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/policy"
{
  "_id": "",
  "resources": [
    ...
    {
      "resource": "internal/user/*",
      "properties": [
        {
          "name": "_id",
          "policies": [
            {
              "policyId": "cannot-contain-characters",
              "params": {
                "forbiddenChars": [ "/" ]
              },
              "policyFunction": "\nfunction (fullObject, value, params, property) {\n    ...",
              "policyRequirements": [
                "CANNOT_CONTAIN_CHARACTERS"
              ]
            }
          ]
        },
        "policyRequirements": [
          "CANNOT_CONTAIN_CHARACTERS"
        ]
      ]
    }
    ...
  ]
}
```

To display the policies that apply to a specific resource, include the resource name in the URL. For example, the following REST call displays the policies that apply to managed users:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/policy/managed/user/*"
{
  "_id": "*",
  "resource": "managed/user/*",
  "properties": [
    {
      "policyRequirements": [
        "VALID_TYPE",
        "CANNOT_CONTAIN_CHARACTERS"
      ],
      "fallbackPolicies": null,
      "name": "_id",
      "policies": [
        {
          "policyRequirements": [
            "VALID_TYPE"
          ],
          "policyId": "valid-type",
          "params": {
            "types": [
              "string"
            ]
          }
        },
        {
          "policyId": "cannot-contain-characters",
          "params": {
            "forbiddenChars": [ "/" ]
          },
          "policyFunction": "...",
          "policyRequirements": [
            "CANNOT_CONTAIN_CHARACTERS"
          ]
        }
      ],
      "conditionalPolicies": null
    },
    ...
  ]
}
```

## Validate Objects and Properties Over REST

To verify that an object adheres to the requirements of all applied policies, include the `validateObject` action in the request.

The following example verifies that a new managed user object is acceptable, in terms of the policy requirements. Note that the ID in the URL (`test` in this example) is ignored—the action simply validates the object in the JSON payload:

```
curl \
```

```
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '{
  "sn": "Jones",
  "givenName": "Bob",
  "telephoneNumber": "0827878921",
  "passPhrase": null,
  "mail": "bjones@example.com",
  "accountStatus": "active",
  "userName": "bjones@example.com",
  "password": "123"
}' \
"http://localhost:8080/openidm/policy/managed/user/test?_action=validateObject"
{
  "result": false,
  "failedPolicyRequirements": [
    {
      "policyRequirements": [
        {
          "policyRequirement": "MIN_LENGTH",
          "params": {
            "minLength": 8
          }
        }
      ],
      "property": "password"
    },
    {
      "policyRequirements": [
        {
          "policyRequirement": "AT_LEAST_X_CAPITAL_LETTERS",
          "params": {
            "numCaps": 1
          }
        }
      ],
      "property": "password"
    }
  ]
}
```

The result (**false**) indicates that the object is not valid. The unfulfilled policy requirements are provided as part of the response - in this case, the user password does not meet the validation requirements.

Use the **validateProperty** action to verify that a specific property adheres to the requirements of a policy.

The following example checks whether a user's new password (**12345**) is acceptable:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
```

```
--header "Content-Type: application/json" \
--request POST \
--data '{
  "password": "12345"
}' \
"http://localhost:8080/openidm/policy/managed/user/9dce06d4-2fc1-4830-a92b-bd35c2f6bcbb?
_action=validateProperty"
{
  "result": false,
  "failedPolicyRequirements": [
    {
      "policyRequirements": [
        {
          "policyRequirement": "MIN_LENGTH",
          "params": {
            "minLength": 8
          }
        }
      ],
      "property": "password"
    },
    {
      "policyRequirements": [
        {
          "policyRequirement": "AT_LEAST_X_CAPITAL_LETTERS",
          "params": {
            "numCaps": 1
          }
        }
      ],
      "property": "password"
    }
  ]
}
```

The result (**false**) indicates that the password is not valid. The unfulfilled policy requirements are provided as part of the response - in this case, the minimum length and the minimum number of capital letters.

Validating a property that fulfills the policy requirements returns a **true** result, for example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '{
  "password": "1NewPassword"
}' \
"http://localhost:8080/openidm/policy/managed/user/9dce06d4-2fc1-4830-a92b-bd35c2f6bcbb?
_action=validateProperty"
{
  "result": true,
  "failedPolicyRequirements": []
}
```

## Validate Field Removal

To validate field removal, specify the fields to remove when calling the policy `validateProperty` action. You cannot remove fields that:

- Are required in the `required` schema array.
- Have a `required` policy.
- Have a default value.

The following example validates the removal of the fields `description` and `givenName`:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '{
  "remove": [ "description", "givenName" ]
}' \
"http://localhost:8080/openidm/policy/managed/user/ca5a3196-2ed3-4a76-8881-30403dee70e9?
_action=validateProperty"
{
  "result": false,
  "failedPolicyRequirements": [
    {
      "policyRequirements": [
        {
          "policyRequirement": "REQUIRED"
        }
      ],
      "property": "givenName"
    }
  ]
}
```

## Validate Properties to Unknown Resource Paths

To perform a `validateProperty` action to a path that is unknown (\*), such as `managed/user/*` or `managed/user/userDoesntExistYet`, the payload must include:

- An `object` field that contains the object details.
- A `properties` field that contains the properties to be evaluated.

### + Pre-registration Validation Example

A common use case for validating properties for unknown resources is prior to object creation, such as during pre-registration.

1. Always pass the object and properties content in the POST body because IDM has no object to look up.

2. Use any placeholder id in the request URL, as `*` has no special meaning in the API.

This example uses a conditional policy for any object with the description `test1`:

```
"password" : {
  ...
  "conditionalPolicies" : [
    {
      "condition" : {
        "type" : "text/javascript",
        "source" : "(fullObject.description === 'test1')"
      },
      "dependencies" : [ "description" ],
      "policies" : [
        {
          "policyId" : "at-least-X-capitals",
          "params" : {
            "numCaps" : 1
          }
        }
      ]
    }
  ]
},
],
}
```

Using the above conditional policy, you could perform a `validateProperty` action to `managed/user/*` with the request:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '{
  "object": {
    "description": "test1"
  },
  "properties": {
    "password": "passw0rd"
  }
}' \
"http://localhost:8080/openidm/policy/managed/user/*?_action=validateProperty"
{
  "result": false,
  "failedPolicyRequirements": [
    {
      "policyRequirements": [
        {
          "params": {
            "numCaps": 1
          },
          "policyRequirement": "AT_LEAST_X_CAPITAL_LETTERS"
        }
      ],
      "property": "password"
    }
  ]
}
```

## Chapter 5

# Store Managed Objects in the Repository

IDM stores managed objects, internal users, and configuration objects in a repository. By default, the server uses an embedded ForgeRock Directory Services (DS) instance as its repository. In production, you must replace this embedded instance with an external DS instance, or with a JDBC repository, as described in "*Select a Repository*" in the *Installation Guide*.

These topics describe the repository configuration, and how objects are mapped in the repository.

- "Repository Configuration Files"
- "Generic and Explicit Object Mappings"

## Repository Configuration Files

Configuration files for all supported repositories are located in the `/path/to/openidm/db/database/conf` directory. For JDBC repositories, the configuration is defined in two files:

- `datasource.jdbc-default.json` specifies the connection to the database.
- `repo.jdbc.json` specifies the mapping between IDM resources and database tables.

For a DS repository, the `repo.ds.json` file specifies the resource mapping and, in the case of an external repository, the connection details to the LDAP server.

For both DS and JDBC, the `conf/repo.init.json` file specifies IDM's initial internal roles and users in the *Security Guide*.

Copy the configuration files for your specific database type to your project's `conf/` directory.

## JDBC Connection Configuration

The default database connection configuration file for a MySQL database follows:

```
{
  "driverClass" : "com.mysql.jdbc.Driver",
  "jdbcUrl" : "jdbc:mysql://&{openidm.repo.host}&{openidm.repo.port}/openidm?allowMultiQueries=true&characterEncoding=utf8&serverTimezone=UTC",
  "databaseName" : "openidm",
  "username" : "openidm",
  "password" : "openidm",
  "connectionTimeout" : 30000,
  "connectionPool" : {
    "type" : "hikari",
    "minimumIdle" : 20,
    "maximumPoolSize" : 50
  }
}
```

The configuration file includes the following properties:

### driverClass

```
"driverClass" : string
```

To use the JDBC driver manager to acquire a data source, set this property, as well as `jdbcUrl`, `username`, and `password`. The driver class must be the fully-qualified class name of the database driver to use for your database.

Using the JDBC driver manager to acquire a data source is the most likely option, and the only one supported "out of the box". The remaining options in the sample repository configuration file assume that you are using a JDBC driver manager.

Example: `"driverClass" : "com.mysql.jdbc.Driver"`

### jdbcUrl

The connection URL to the JDBC database. The URL should include all of the parameters required by your database. For example, to specify the encoding in MySQL use `'characterEncoding=utf8'`.

Specify the values for `openidm.repo.host` and `openidm.repo.port` in one of the following ways:

- Set the values in `resolver/boot.properties` or your project's `conf/system.properties` file, for example:

```
openidm.repo.host = localhost
openidm.repo.port = 3306
```

- Set the properties in the `OPENIDM_OPTS` environment variable and export that variable before startup. You must include the JVM memory options when you set this variable. For example:

```
export OPENIDM_OPTS="-Xmx1024m -Xms1024m -Dopenidm.repo.host=localhost -Dopenidm.repo.port=3306"
/path/to/openidm/startup.sh
Executing ./startup.sh...
Using OPENIDM_HOME: /path/to/openidm
Using PROJECT_HOME: /path/to/openidm
Using OPENIDM_OPTS: -Xmx1024m -Xms1024m -Dopenidm.repo.host=localhost -Dopenidm.repo.port=3306
Using LOGGING_CONFIG: -Djava.util.logging.config.file=/path/to/openidm/conf/logging.properties
Using boot properties at /path/to/openidm/resolver/boot.properties
-> OpenIDM version "7.0.4"
OpenIDM ready
```

### databaseName

The name of the database to which IDM connects. By default, this is `openidm`.

### username

The username with which to access the JDBC database.

### password

The password with which to access the JDBC database. IDM automatically encrypts clear string passwords. To replace an existing encrypted value, replace the whole `crypto-object` value, including the brackets, with a string of the new password.

### connectionTimeout

The period of time, in milliseconds, after which IDM should consider an attempted connection to the database to have failed. The default period is 30000 milliseconds (30 seconds).

### connectionPool

Database connection pooling configuration. The default connection pool library is HikariCP:

```
"connectionPool" : {
  "type" : "hikari"
}
```

IDM uses the default HikariCP configuration, except for the following parameters. You might need to adjust these parameters, according to your database workload:

- `minimumIdle`

This property controls the minimum number of idle connections that HikariCP maintains in the connection pool. If the number of idle connections drops below this value, HikariCP attempts to add additional connections.

By default, HikariCP runs as a fixed-sized connection pool, that is, this property is not set. The connection configuration files provided with IDM set the minimum number of idle connections to `20`.

- `maximumPoolSize`

This property controls the maximum number of connections to the database, including idle connections and connections that are being used.

By default, HikariCP sets the maximum number of connections to **10**. The connection configuration files provided with IDM set the maximum number of connections to **50**.

For information about the HikariCP configuration parameters, see the [HikariCPCP Project Page](#).

## JDBC Database Table Configuration

An excerpt of a MySQL database table configuration file follows:

```
{
  "dbType" : "MYSQL",
  "useDataSource" : "default",
  "maxBatchSize" : 100,
  "maxTxRetry" : 5,
  "queries" : {...},
  "commands" : {...},
  "resourceMapping" : {...}
}
```

The configuration file includes the following properties:

### **dbType** : string, optional

The type of database. The database type might affect the queries used and other optimizations. Supported database types include the following:

DB2  
SQLSERVER (for Microsoft SQL Server)  
MYSQL  
ORACLE  
POSTGRESQL

### **useDataSource** : string, optional

This option refers to the connection details that are defined in the configuration file, described previously. The default configuration file is named `datasource.jdbc-default.json`. This is the file that is used by default (and the value of the `"useDataSource"` is therefore `"default"`). You might want to specify a different connection configuration file, instead of overwriting the details in the default file. In this case, set your connection configuration file `datasource.jdbc-name.json` and set the value of `"useDataSource"` to whatever *name* you have used.

### **maxBatchSize**

The maximum number of SQL statements that will be batched together. This parameter allows you to optimize the time taken to execute multiple queries. Certain databases do not support batching, or limit how many statements can be batched. A value of **1** disables batching.

### maxTxRetry

The maximum number of times that a specific transaction should be attempted before that transaction is aborted.

### queries

Any custom `queries` that can be referenced from the configuration.

Options supported for query parameters include the following:

- A default string parameter, for example:

```
openidm.query("managed/user", { "_queryId": "for-username", "uid": "jdoe" });
```

For more information about the query function, see `openidm.query(resourceName, params, fields)` in the *Scripting Guide*.

- A list parameter (`${list:propName}`).

Use this parameter to specify a set of indeterminate size as part of your query. For example:

```
WHERE targetObjectId IN (${list:filteredIds})
```

- A boolean parameter (`${bool:propName}`).

Use this parameter to query boolean values in the database.

- Numeric parameters for integers (`${int:propName}`), large integers (`${long:propName}`), and decimal values (`${num:propName}`).

Use these parameters to query numeric values in the database, corresponding to the column data type in your repository.

### commands

Specific commands configured to manage the database over the REST interface. Currently, the following default commands are included in the configuration:

- `purge-by-recon-expired`
- `purge-by-recon-number-of`
- `delete-mapping-links`
- `delete-target-ids-for-recon`

These commands assist with removing stale reconciliation audit information from the repository, and preventing the repository from growing too large. The commands work by executing a query filter, then performing the specified operation on each result set. Currently the only supported operation is `DELETE`, which removes all entries that match the filter.

## resourceMapping

Defines the mapping between IDM resource URIs (for example, `managed/user`) and JDBC tables. The structure of the resource mapping is as follows:

```
"resourceMapping" : {
  "default" : {
    "mainTable" : "genericobjects",
    "propertiesTable" : "genericobjectproperties",
    "searchableDefault" : true
  },
  "genericMapping" : {...},
  "explicitMapping" : {...}
}
```

The default mapping object represents a default generic table in which any resource that does not have a more specific mapping is stored.

The generic and explicit mapping objects are described in the following section.

## DS Repository Configuration

An excerpt of a DS repository configuration file follows:

```
{
  "embedded" : false,
  "maxConnectionAttempts" : 5,
  "security" : {...},
  "ldapConnectionFactory" : {...},
  "queries" : {...},
  "commands" : {...},
  "rest2LdapOptions" : {...},
  "indices" : {...},
  "schemaProviders" : {...},
  "resourceMapping" : {...}
}
```

The configuration file includes the following properties:

### **embedded** : boolean

Specifies an embedded or external DS instance.

IDM uses an embedded DS instance by default. The embedded instance is not supported in production.

### **maxConnectionAttempts** : integer

Specifies the number of times IDM should attempt to connect to the DS instance. On startup, IDM will attempt to connect to DS indefinitely. The `maxConnectionAttempts` parameter controls the number of reconnection attempts in the event of a failure during normal operation, for example, if an attempt to access the DS repository times out.

By default, IDM will attempt to reconnect to the DS instance 5 times.

## security

Specifies the keystore and truststore for secure connections to DS.

```
"security": {
  "trustManager": "file",
  "fileBasedTrustManagerType": "JKS",
  "fileBasedTrustManagerFile": "&{idm.install.dir}/security/truststore",
  "fileBasedTrustManagerPasswordFile": "&{idm.install.dir}/security/storepass"
}
```

In the default case, where DS servers use TLS key pairs generated using a deployment key and password, you must import the deployment key-based CA certificate into the IDM truststore. For more information, see ["External DS Repository"](#) in the *Installation Guide*.

Note that the **security** settings have no effect for an embedded DS repository. Embedded DS is not supported in production, and is meant for evaluation or testing purposes only.

## ldapConnectionFactories

For an external DS repository, configures the connection to the DS instance. For example:

```
"ldapConnectionFactories": {
  "bind": {
    "connectionSecurity": "startTLS",
    "heartBeatIntervalSeconds": 60,
    "heartBeatTimeoutMilliseconds": 10000,
    "primaryLdapServers": [
      {
        "hostname": "localhost",
        "port": 31389
      }
    ],
    "secondaryLdapServers": []
  },
  "root": {
    "inheritFrom": "bind",
    "authentication": {
      "simple": { "bindDn": "uid=admin", "bindPassword": "password" }
    }
  }
}
```

The connection to the DS repository uses the DS *REST2LDAP* gateway and the **ldapConnectionFactories** property sets the gateway configuration. For example, the **secondaryLdapServers** property specifies an array of LDAP servers that the gateway can contact if the primary LDAP servers cannot be contacted.

For information on all the gateway configuration properties, see [Gateway Configuration](#) in the *DS REST API Guide*.

## queries

Predefined queries that can be referenced from the configuration. For a DS repository, all predefined queries are really filtered queries (using the **\_queryFilter** parameter), for example:

```
"query-all-ids": {
  "_queryFilter": "true",
  "_fields": "_id,_rev"
}
```

The queries are divided between those for **generic** mappings and those for **explicit** mappings, but the queries themselves are the same for both mapping types.

#### commands

Specific commands configured to manage the repository over the REST interface. Currently, only two commands are included by default:

- **delete-mapping-links**
- **delete-target-ids-for-recon**

Both of these commands assist with removing stale reconciliation audit information from the repository, and preventing the repository from growing too large.

#### rest2LdapOptions

Specifies the configuration for accessing the LDAP data stored in DS. For more information, see [Gateway REST2LDAP Configuration in the DS REST API Guide](#).

#### indices

For generic mappings, enables you to set up LDAP indices on custom object properties. For more information, see ["Improving Generic Mapping Search Performance \(DS\)"](#).

#### schemaProviders

For generic mappings, enables you to list custom objects whose properties should be indexed. For more information, see ["Improving Generic Mapping Search Performance \(DS\)"](#).

#### resourceMapping

Defines the mapping between IDM resource URIs (for example, **managed/user**) and the DS directory tree. The structure of the resource mapping object is as follows:

```
{
  "resourceMapping" : {
    "defaultMapping" : {
      "dnTemplate": "ou=generic,dc=openidm,dc=forgerock,dc=com"
    },
    "explicitMapping" : {...},
    "genericMapping" : {...}
  }
}
```

The default mapping object represents a default generic organizational unit (**ou**) in which any resource that does not have a more specific mapping is stored.

The generic and explicit mapping objects are described in "Generic and Explicit Object Mappings" .

## Generic and Explicit Object Mappings

There are two ways to map IDM objects to the tables in a JDBC database or to organizational units in DS:

- *Generic mapping*, which allows you to store arbitrary objects without special configuration or administration.
- *Explicit mapping*, which maps specific objects and properties to tables and columns in the JDBC database or to organizational units in DS.

By default, IDM uses a generic mapping for user-definable objects, for both a JDBC and a DS repository. A generic mapping speeds up initial deployment, and can make system maintenance more flexible by providing a stable database structure. In a test environment, generic tables enable you to modify the user and object model easily, without database access, and without the need to constantly add and drop table columns. However, generic mapping does not take full advantage of the underlying database facilities, such as validation within the database and flexible indexing. Using an explicit mapping generally results in a *substantial* performance improvement. It is therefore strongly advised that you change to an explicit mapping before deploying in a production environment. If you are integrating IDM with AM, and using a shared DS repository, you *must* use an explicit schema mapping.

These two mapping strategies are discussed in the following sections, for JDBC repositories and for DS repositories:

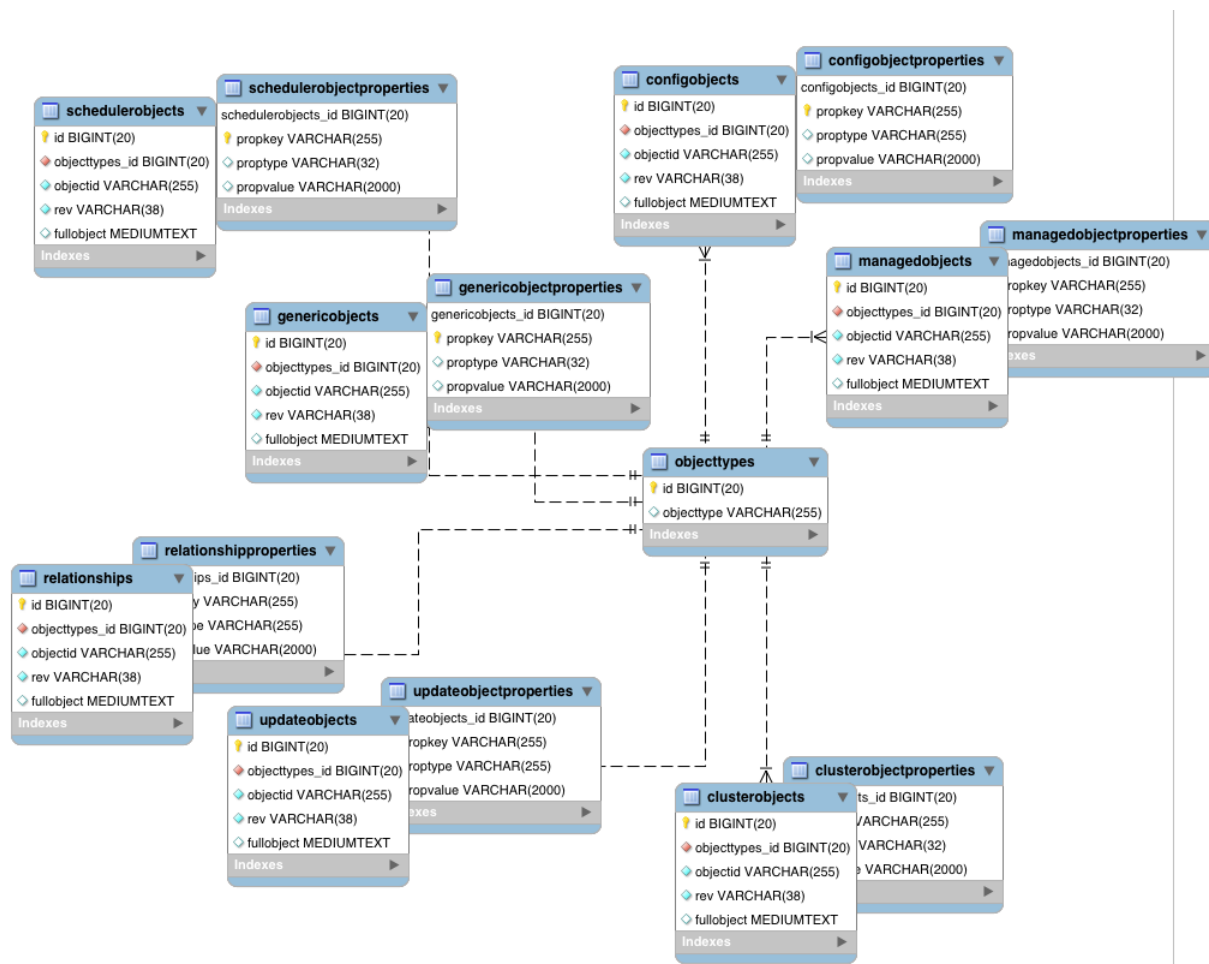
### Generic and Explicit Mappings With a JDBC Repository

#### Generic Mappings With a JDBC Repository

Generic mapping speeds up development, and can make system maintenance more flexible by providing a stable database structure. However, generic mapping can have a performance impact and does not take full advantage of the database facilities (such as validation within the database and flexible indexing). In addition, queries can be more difficult to set up.

In a generic table, the entire object content is stored in a single large-character field named `fullobject` in the `mainTable` for the object. To search on specific fields, you can read them by referring to them in the corresponding `properties table` for that object. The disadvantage of generic objects is that, because every property you might like to filter by is stored in a separate table, you must join to that table each time you need to filter by anything.

The following diagram shows a pared down database structure for the default generic table, when using a MySQL repository. The diagram indicates the relationship between the main table and the corresponding properties table for each object.



```
SELECT obj.objectid, obj.rev, obj.fullobject FROM ${_dbSchema}.${_mainTable} obj
INNER JOIN ${_dbSchema}.${_propTable} prop ON obj.id = prop.${_mainTable}_id
INNER JOIN ${_dbSchema}.objecttypes objtype ON objtype.id = obj.objecttypes_id
WHERE prop.propkey='userName' AND prop.propvalue = ${uid} AND objtype.objecttype = ${ resource}";
```

```
SELECT obj.objectid, obj.rev, obj.fullobject FROM ${_dbSchema}.${_mainTable} obj
```

2. Join to the properties table and locate the object with the corresponding ID:

```
INNER JOIN ${_dbSchema}.${_propTable} prop ON obj.id = prop.${_mainTable}_id
```

3. Join to the object types table to restrict returned entries to objects of a specific type. For example, you might want to restrict returned entries to **managed/user** objects, or **managed/role** objects:

```
INNER JOIN ${_dbSchema}.objecttypes objtype ON objtype.id = obj.objecttypes_id
```

4. Filter records by the **userName** property, where the userName is equal to the specified **uid** and the object type is the specified type (in this case, managed/user objects):

```
WHERE prop.propkey='/userName'
      AND prop.propvalue = ${uid}
      AND objtype.objtype = ${_resource}",
```

The value of the **uid** field is provided as part of the query call, for example:

```
openidm.query("managed/user", { "_queryId": "for-userName", "uid": "jdoe" });
```

Tables for user definable objects use a generic mapping by default.

The following sample generic mapping object illustrates how **managed/** objects are stored in a generic table:

```
"genericMapping" : {
  "managed/*" : {
    "mainTable" : "managedobjects",
    "propertiesTable" : "managedobjectproperties",
    "searchableDefault" : true,
    "properties" : {
      "/picture" : {
        "searchable" : false
      }
    }
  }
}
```

### **mainTable** (string, mandatory)

Indicates the main table in which data is stored for this resource.

The complete object is stored in the **fullobject** column of this table. The table includes an **objecttypes** foreign key that is used to distinguish the different objects stored within the table. In addition, the revision of each stored object is tracked, in the **rev** column of the table, enabling multiversion concurrency control (MVCC). For more information, see "Manipulating Managed Objects Programmatically".

### **propertiesTable** (string, mandatory)

Indicates the properties table, used for searches.

#### Note

PostgreSQL repositories do not use these properties tables to access specific properties. Instead, the PostgreSQL `json_extract_path_text()` function achieves this functionality.

The contents of the properties table is a defined subset of the properties, copied from the character large object (CLOB) that is stored in the `fullobject` column of the main table. The properties are stored in a one-to-many style separate table. The set of properties stored here is determined by the properties that are defined as `searchable`.

The stored set of searchable properties makes these values available as discrete rows that can be accessed with SQL queries, specifically, with `WHERE` clauses. It is not otherwise possible to query specific properties of the full object.

The properties table includes the following columns:

- `${mainTable}_id` corresponds to the `id` of the full object in the main table, for example, `manageobjects_id`, or `genericobjects_id`.
- `propkey` is the name of the searchable property, stored in JSON pointer format (for example `/mail`).
- `proptype` is the data type of the property, for example `java.lang.String`. The property type is obtained from the Class associated with the value.
- `propvalue` is the value of property, extracted from the full object that is stored in the main table.

Regardless of the property data type, this value is stored as a string, so queries against it should treat it as such.

#### `searchableDefault` (boolean, optional)

Specifies whether all properties of the resource should be searchable by default. Properties that are searchable are stored and indexed. You can override the default for individual properties in the `properties` element of the mapping. The preceding example indicates that all properties are searchable, with the exception of the `picture` property.

For large, complex objects, having all properties searchable implies a substantial performance impact. In such a case, a separate insert statement is made in the properties table for each element in the object, every time the object is updated. Also, because these are indexed fields, the recreation of these properties incurs a cost in the maintenance of the index. You should therefore enable `searchable` only for those properties that must be used as part of a `WHERE` clause in a query.

#### Note

PostgreSQL repositories do not use the `searchableDefault` property.

### properties

Lists any individual properties for which the searchable default should be overridden.

Note that if an object was originally created with a subset of `searchable` properties, changing this subset (by adding a new `searchable` property in the configuration, for example) will not cause the existing values to be updated in the properties table for that object. To add the new property to the properties table for that object, you must update or recreate the object.

## Improving Generic Mapping Search Performance (JDBC)

All properties in a generic mapping are searchable by default. In other words, the value of the `searchableDefault` property is `true` unless you explicitly set it to false. Although there are no individual indexes in a generic mapping, you can improve search performance by setting only those properties that you need to search as `searchable`. Properties that are searchable are created within the corresponding properties table. The properties table exists only for searches or look-ups, and has a composite index, based on the resource, then the property name.

The sample JDBC repository configuration files (`db/database/conf/repo.jdbc.json`) restrict searches to specific properties by setting the `searchableDefault` to `false` for `managed/user` mappings. You must explicitly set `searchable` to true for each property that should be searched. The following sample extract from `repo.jdbc.json` indicates searches restricted to the `userName` property:

```
"genericMapping" : {
  "managed/user" : {
    "mainTable" : "manageduserobjects",
    "propertiesTable" : "manageduserobjectproperties",
    "searchableDefault" : false,
    "properties" : {
      "/userName" : {
        "searchable" : true
      }
    }
  }
}
```

With this configuration, IDM creates entries in the properties table only for `userName` properties of managed user objects.

If the global `searchableDefault` is set to false, properties that do not have a searchable attribute explicitly set to true are not written in the properties table.

## Explicit Mappings With a JDBC Repository

Explicit mapping is more difficult to set up and maintain, but can take complete advantage of the native database facilities.

An explicit table offers better performance and simpler queries. There is less work in the reading and writing of data, because the data is all in a single row of a single table. In addition, it is easier to create different types of indexes that apply to only specific fields in an explicit table. The disadvantage of explicit tables is the additional work required in creating the table in the schema. Also, because rows in a table are inherently more simple, it is more difficult to deal with complex objects. Any non-simple key:value pair in an object associated with an explicit table is converted to a JSON string and stored in the cell in that format. This makes the value difficult to use, from the perspective of a query attempting to search within it.

You can have a generic mapping configuration for most managed objects, *and* an explicit mapping that overrides the default generic mapping in certain cases.

IDM provides a sample configuration, for each JDBC repository, that sets up an explicit mapping for the managed *user* object and a generic mapping for all other managed objects. This configuration is defined in the files named `/path/to/openidm/db/repository/conf/repo.jdbc-repository-explicit-managed-user.json`. To use this configuration, copy the file that corresponds to your repository to your project's `conf/` directory and rename it `repo.jdbc.json`. Run the `sample-explicit-managed-user.sql` data definition script (in the `path/to/openidm/db/repository/scripts` directory) to set up the corresponding tables when you configure your JDBC repository.

IDM uses explicit mapping for internal system tables, such as the tables used for auditing.

Depending on the types of usage your system is supporting, you might find that an explicit mapping performs better than a generic mapping. Operations such as sorting and searching (such as those performed in the default UI) tend to be faster with explicitly-mapped objects, for example.

The following sample explicit mapping object illustrates how `internal/user` objects are stored in an explicit table:

```
"explicitMapping" : {
  "internal/user" : {
    "table" : "internaluser",
    "objectToColumn" : {
      "_id" : "objectid",
      "_rev" : { "column" : "rev", "isNotNull" : true },
      "password" : "pwd"
    }
  },
  ...
}
```

#### **<resource-uri> (string, mandatory)**

Indicates the URI for the resources to which this mapping applies, for example, `internal/user`.

#### **table (string, mandatory)**

The name of the database table in which the object (in this case internal users) is stored.

#### **objectToColumn (string, mandatory)**

The way in which specific managed object properties are mapped to columns in the table.

The mapping can be a simple one to one mapping, for example `"userName": "userName"`, or a more complex JSON map or list. When a column is mapped to a JSON map or list, the syntax is as shown in the following examples:

```
"messageDetail" : { "column" : "messagedetail", "type" : "JSON_MAP" }
```

or

```
"roles" : { "column" : "roles", "type" : "JSON_LIST" }
```

Available column data types you can specify are `STRING` (the default), `NUMBER`, `JSON_MAP`, `JSON_LIST`, and `FULLOBJECT`.

You can also prevent a column from accepting a `NULL` value, by setting the property `isNotNull` to `true`. This property is optional; if the property is omitted, it will default to `false`. Specifying which columns do not allow a null value can improve performance when sorting and paginating large queries. The syntax is similar to when specifying a column type:

```
"createDate" : { "column" : "createDate", "isNotNull" : true }
```

## Caution

Pay particular attention to the following caveats when you map properties to explicit columns in your database:

- Support for data types in columns is restricted to numeric values (`NUMBER`), strings (`STRING`), and boolean values (`BOOLEAN`). Although you can specify other data types, IDM handles all other data types as strings. Your database will need to convert these types from a string to the alternative data type. This conversion is *not guaranteed to work*.

If the conversion does work, the format might not be the same when the data is read from the database as it was when it was saved. For example, your database might parse a date in the format `12/12/2012` and return the date in the format `2012-12-12` when the property is read.

- Passwords are encrypted before they are stored in the repository. The length of the password column must be long enough to store the encrypted password value, which can vary depending on how it is encrypted and whether it is also hashed.

The `sample-explicit-managed-user.sql` file referenced in this section sets the password column to a length of 511 characters (`VARCHAR(511)`) to account for the additional space an encrypted password requires. For more information about IDM encryption and an example encrypted password value, see `"encrypt"` in the *Setup Guide* and `"Encoding Attribute Values"` in the *Security Guide*.

- If your data objects include *virtual properties*, you must include columns in which to store these properties. If you don't explicitly map the virtual properties, you will see errors similar to the following when you attempt to create the corresponding object:

```
{
  "code":400,
  "reason":"Bad Request",
  "message":"Unmapped fields [/property-name/0] for type managed/user and table
  openidm.managed_user"
}
```

When virtual properties are returned in the result of a query, the query previously persisted values of the requested virtual properties. To recalculate virtual property values in a query, you must set `executeOnRetrieve` to `true` in the query request parameters. For more information, see "Property Storage Triggers".

## Generic and Explicit Mappings With a DS Repository

For both generic and explicit mappings, IDM maps object types using a `dnTemplate` property. The `dnTemplate` is effectively a pointer to where the object is stored in DS. For example, the following excerpt of the default `repo.ds.json` file shows how configuration objects are stored under the DN `ou=config,dc=openidm,dc=forgerock,dc=com`:

```
"config": {
  "dnTemplate": "ou=config,dc=openidm,dc=forgerock,dc=com"
}
```

## Generic Mappings With a DS Repository

By default, IDM uses a generic mapping for all objects *except* the following:

- Internal users, roles, and privileges
- Links
- Clustered reconciliation target IDs

### Note

Clustered reconciliation is not currently supported with a DS repository.

- Locks
- Objects related to queued synchronization

With a generic mapping, all the properties of an object are stored as a single JSON blob in the `fr-idm-json` attribute. To create a new generic mapping, you need only specify the `dnTemplate`, that is, where the object will be stored in the directory tree.

You can specify a wildcard mapping, that stores all nested URIs under a particular branch of the directory tree, for example:

```
"managed/*": {
  "dnTemplate": "ou=managed,dc=openidm,dc=forgerock,dc=com"
}
```

With this mapping, all objects under `managed/`, such as `managed/user` and `managed/device`, will be stored in the branch `ou=managed,dc=openidm,dc=forgerock,dc=com`. You do not have to specify separate mappings

for each of these objects. The mapping creates a new `ou` for each object. So, for example, `managed/user` objects will be stored under the DN `ou=user,ou=managed,dc=openidm,dc=forgerock,dc=com` and `managed/device` objects will be stored under the DN `ou=device,ou=managed,dc=openidm,dc=forgerock,dc=com`.

## Improving Generic Mapping Search Performance (DS)

By default, all generic objects are instances of the `fr-idm-generic-obj` object class and their properties are stored as a single JSON blob in the `fr-idm-json` attribute. The `fr-idm-json` attribute is indexed by default, which results in *all* attributes of a generic object being indexed. JDBC repositories behave in a similar way, with all generic objects being searchable by default.

To optimize search performance on specific generic resources, you can set up your own schema providers and indices as described in this section. For a detailed explanation of how indexes improve LDAP search performance, see [Indexes](#) in the *DS Configuration Guide*.

For the embedded DS repository, or an external DS repository installed as described in "External DS Repository" in the *Installation Guide*, the following managed user properties are indexed by default:

- `userName` (cn)
- `givenName`
- `sn`
- `mail`
- `accountStatus`

You can configure managed user indexes in the repository configuration (`repo.ds.json`) by adding `indices` and `schemaProviders` objects, as follows:

```
"indices" : {
  ...
  "fr-idm-managed-user-json" : {
    "type" : [ "EQUALITY" ]
  },
  ...
},
"schemaProviders" : {
  "IDM managed/user Json Schema" : {
    "matchingRuleName" : "caseIgnoreJsonQueryMatchManagedUser",
    "matchingRuleOid" : "1.3.6.1.4.1.36733.2....",
    "caseSensitiveStrings" : false,
    "fields" : [ "accountStatus", "givenName", "mail", "sn", "userName" ]
  },
  ...
}
```

The indexed properties are listed in the array of `fields` for that managed object. To index additional managed user properties, add the property names to this array of `fields`.

To set up indexes on generic objects other than the managed user object, you must do the following:

- Add the object to the DS schema.

The schema for an embedded DS repository is stored in the `/path/to/openidm/db/openidm/opensj/db/schema/60-repo-schema.ldif` file.

You can use the managed user object as an example of the schema syntax:

```
###
# Managed User
###
attributeTypes: ( 1.3.6.1.4.1.36733.2.3.1.13
  NAME 'fr-idm-managed-user-json'
  SYNTAX 1.3.6.1.4.1.36733.2.1.3.1
  EQUALITY caseIgnoreJsonQueryMatchManagedUser
  ORDERING caseIgnoreOrderingMatch
  SINGLE-VALUE
  X-ORIGIN 'OpenIDM DSRepoService')
objectClasses: ( 1.3.6.1.4.1.36733.2.3.2.6
  NAME 'fr-idm-managed-user'
  SUP top
  STRUCTURAL
  MUST ( fr-idm-managed-user-json )
  X-ORIGIN 'OpenIDM DSRepoService' )
```

For information about adding JSON objects to the DS schema, see [Schema and JSON in the DS Configuration Guide](#).

#### Warning

If you delete the `db/openidm` directory, any additions you have made to the schema will be lost. If you have customized the schema, be sure to back up the `60-repo-schema.ldif` file.

- Add the object to the `indices` property in the `conf/repo.ds.json` file.

The following example sets up an equality index for a managed devices object:

```
"indices" : {
  ...
  "fr-idm-managed-devices-json" : {
    "type" : [ "EQUALITY" ]
  },
  ...
}
```

- Add the object to the `schemaProviders` property in the `conf/repo.ds.json` file and list the properties that should be indexed.

The following example sets up indexes for the `deviceName`, `brand`, and `assetNumber` properties of the managed device object:

```
"schemaProviders" : {  
  "Managed Device Json" : {  
    "matchingRuleName" : "caseIgnoreJsonQueryMatchManagedDevice",  
    "matchingRuleId" : "1.3.6.1.4.1.36733.2.....",  
    "caseSensitiveStrings" : false,  
    "fields" : [ "deviceName", "brand", "assetNumber" ]  
  }  
}
```

For more information about indexing JSON attributes, see [JSON Query Matching Rule Index](#) in the *DS Configuration Guide*.

#### Note

The OIDs shown in this section are reserved for ForgeRock internal use. If you set up additional objects and attributes, or if you change the default schema, you must specify your own OIDs here.

## Explicit Mappings With a DS Repository

The default configuration uses a generic mapping for managed user objects. To use an explicit mapping for managed user objects, change the repository configuration *before you start IDM for the first time*.

To set up an explicit mapping:

1. Copy the `repo.ds-explicit-managed-user.json` file to your project's `conf` directory, and rename that file `repo.ds.json`:

```
cp /path/to/openidm/db/ds/conf/repo.ds-explicit-managed-user.json project-dir/conf/repo.ds.json
```

#### Important

This file is configured for an embedded DS repository by default. To set up an explicit mapping for an external DS repository, change the value of the `embedded` property to `false` and add the following properties:

```
"security": {
  "trustManager": "file",
  "fileBasedTrustManagerType": "JKS",
  "fileBasedTrustManagerFile": "&{idm.install.dir}/security/truststore",
  "fileBasedTrustManagerPasswordFile": "&{idm.install.dir}/security/storepass"
},
"ldapConnectionFactories": {
  "bind": {
    "connectionSecurity": "startTLS",
    "heartBeatIntervalSeconds": 60,
    "heartBeatTimeoutMilliseconds": 10000,
    "primaryLdapServers": [
      {
        "hostname": "localhost",
        "port": 31389
      }
    ],
    "secondaryLdapServers": []
  },
  "root": {
    "inheritFrom": "bind",
    "authentication": {
      "simple": {
        "bindDn": "uid=admin",
        "bindPassword": "password"
      }
    }
  }
}
}
```

For more information on these properties, see "DS Repository Configuration".

## 2. Start IDM.

IDM uses the DS REST to LDAP gateway to map JSON objects to LDAP objects stored in the directory. To create additional explicit mappings, you must specify the LDAP **objectClasses** to which the object is mapped, and how each property maps to its corresponding LDAP attributes. Specify at least the property **type** and the corresponding **ldapAttribute**. For relationships between objects, you must explicitly define those objects in the repository configuration.

The following excerpt shows an example of an explicit managed user object mapping:

```
"managed/user" : {
  "dnTemplate": "ou=user,ou=managed,dc=openidm,dc=forgerock,dc=com",
  "objectClasses": [
    "person",
    "organizationalPerson",
    "inetOrgPerson",
    "fr-idm-managed-user-explicit",
    "inetuser"
  ],
  "properties": {
    "_id": {
      "type": "simple", "ldapAttribute": "uid", "isRequired": true, "writability": "createOnly"
    },
  },
}
```

```

    "userName": {
      "type": "simple", "ldapAttribute": "cn"
    },
    "password": {
      "type": "json", "ldapAttribute": "fr-idm-password"
    },
    "accountStatus": {
      "type": "simple", "ldapAttribute": "fr-idm-accountStatus"
    },
    "roles": {
      "type": "json", "ldapAttribute": "fr-idm-role", "isMultiValued": true
    },
    "effectiveRoles": {
      "type": "json", "ldapAttribute": "fr-idm-effectiveRole", "isMultiValued": true
    },
    "effectiveAssignments": {
      "type": "json", "ldapAttribute": "fr-idm-effectiveAssignment", "isMultiValued": true
    },
    ...
  }
}

```

You do not need to map the `_rev` (revision) property of an object as this property is implicit in all objects and maps to the DS `etag` operational attribute.

If your data objects include *virtual properties*, you must include property mappings for these properties. If you don't explicitly map the virtual properties, you will see errors similar to the following when you attempt to create the corresponding object:

```

{
  "code": 400,
  "reason": "Bad Request",
  "message": "Unmapped fields..."
}

```

For more information about the REST to LDAP property mappings, see [Mapping Configuration](#) in the *DS REST API Guide*.

For performance reasons, the DS repository does not apply unique constraints to links. This behavior is different to the JDBC repositories, where uniqueness on link objects is enforced.

### Important

DS currently has a default index entry limit of 4000. Therefore, you cannot query more than 4000 records unless you create a Virtual List View (VLV) index. A VLV index is designed to help DS respond to client applications that need to browse through a long list of objects.

You cannot create a VLV index on a JSON attribute. For generic mappings, IDM avoids this restriction by using client-side sorting and searching. However, for explicit mappings you *must* create a VLV index for any filtered

or sorted results, such as results displayed in a UI grid. To configure a VLV index, use the **dsconfig** command described in Virtual List View Index in the *DS Configuration Guide*.

## Specifying How IDM IDs Map to LDAP Entry Names

The DS REST2LDAP configuration lets you set a **namingStrategy** that specifies how LDAP entry names are mapped to JSON resources. When IDM stores its objects in a DS repository, this **namingStrategy** determines how the IDM **\_id** value maps to the Relative Distinguished Name (RDN) of the corresponding DS object.

The **namingStrategy** is specified as part of the **explicitMapping** of an object in the **repo.ds.json** file. The following example shows a naming strategy configuration for an explicit managed user mapping:

```
"resourceMapping": {
  "defaultMapping": {
    "dnTemplate": "ou=generic,dc=openidm,dc=forgerock,dc=com"
  },
  ...
  "explicitMapping": {
    "managed/user": {
      "dnTemplate": "ou=user,ou=managed,dc=openidm,dc=forgerock,dc=com",
      "objectClasses": [
        "person",
        "organizationalPerson",
        "inetOrgPerson",
        "fr-idm-managed-user-explicit"
      ],
      "namingStrategy": {
        "type": "clientDnNaming",
        "dnAttribute": "uid"
      },
      ...
    }
  }
}
```

The **namingStrategy** can be one of the following:

- **clientDnNaming** - IDM provides an **\_id** to DS and that **\_id** is used to generate the DS RDN. In the following example, the IDM **\_id** maps to the LDAP **uid** attribute:

```
{
  "namingStrategy": {
    "type": "clientDnNaming",
    "dnAttribute": "uid"
  }
}
```

With this *default* configuration, entries are stored in DS with a DN similar to the following:

```
"uid=idm-uuid,ou=user,ou=managed,dc=openidm,dc=forgerock,dc=com"
```

### Note

If these default DNs are suitable in your deployment, you do not have to change anything with regard to the naming strategy.

- **clientNaming** - IDM provides an **\_id** to DS but the DS RDN is derived from a different user attribute in the LDAP entry. In the following example, the RDN is the **cn** attribute. The **\_id** that IDM provides for the object maps to the LDAP **uid** attribute:

```
{
  "namingStrategy": {
    "type": "clientNaming",
    "dnAttribute": "cn",
    "idAttribute": "uid"
  }
}
```

With this configuration, entries are stored in DS with a DN similar to the following:

```
"cn=username,ou=user,ou=managed,dc=openidm,dc=forgerock,dc=com"
```

Specifying a **namingStrategy** is optional. If you do not specify a strategy, the default is **clientDnNaming** with the following configuration:

```
{
  "namingStrategy" : {
    "type" : "clientDnNaming",
    "dnAttribute" : "uid"
  },
  "properties" : {
    "_id": {
      "type": "simple",
      "ldapAttribute": "uid",
      "isRequired": true,
      "writability": "createOnly"
    },
    ...
  }
}
```

### Note

If you do not set a **dnAttribute** as part of the naming strategy, the value of the **dnAttribute** is taken from the value of the **ldapAttribute** on the **\_id** property.

## Relationship Properties in a DS Repository

The IDM object model lets you define relationships between objects. In a DS repository, relationships are implemented using the **reference** and **reverseReference** REST to LDAP property types. For more

information about the `reference` and `reverseReference` property types, read the JSON property mapping section of the *DS HTTP User Guide*.

Relationship properties must be defined in the repository configuration (`repo.ds.json`), for both generic and explicit object mappings.

The following property definitions for a `managed/user` object show how the relationship between a `manager` and their `reports` is defined in the repository configuration:

```
"managed/user" : {
  "dnTemplate" : "ou=user,ou=managed,dc=openidm,dc=forgerock,dc=com",
  ...
  "properties" : {
    ...
    "reports" : {
      "type" : "reverseReference",
      "resourcePath" : "managed/user",
      "propertyName" : "manager",
      "isMultiValued" : true
    },
    "manager" : {
      "type" : "reference",
      "ldapAttribute" : "fr-idm-managed-user-manager",
      "primaryKey" : "uid",
      "resourcePath" : "managed/user",
      "isMultiValued" : false
    },
    ...
  }
}
```

This configuration sets the `reports` property as a `reverseReference`, or reverse *relationship* of the `manager` property. This means that if you add a `manager` to a user, the user automatically becomes one of the `reports` of that manager.

Note the `ldapAttribute` defined in the relationship object (`fr-idm-managed-user-manager` in this case). Your DS schema must include this attribute, and an object class that contains this attribute. Relationship attributes in the DS schema must use the *Name and Optional* JSON syntax.

The following example shows the DS schema definition for the IDM `manager` property:

```
attributeTypes: ( 1.3.6.1.4.1.36733.2.3.1.69
  NAME 'fr-idm-managed-user-manager'
  DESC 'Reference to a users manager'
  SINGLE-VALUE
  SYNTAX 1.3.6.1.4.1.36733.2.1.3.12
  EQUALITY nameAndOptionalCaseIgnoreJsonIdEqualityMatch
  X-STABILITY 'Internal' )
```

## Important

If you define a relationship in the schema (`managed.json`) and you do not define that relationship as a reference or reverse reference in the repository configuration (`repo.ds.json`), you will be able to query the relationships, but filtering and sorting on those queries will not work. This is the case when you define relationship objects

in the Admin UI—the relationship is defined only in the managed object schema and not in the repository configuration.

In this case, queries such as the following are not supported:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/_id/managedOrgs?
_pageSize=50&_sortKeys=_id&_totalPagedResultsPolicy=ESTIMATE&_queryFilter=true"
```

This restriction includes delegated admin privilege filters.

## Chapter 6

# Access Data Objects

You can access data objects by using scripts (through the Resource API) or by using direct HTTP calls (through the REST API).

These sections describe these two methods of accessing objects, and provide information on constructing and calling data queries:

- "Access Data Objects By Using Scripts"
- "Access Data Objects By Using the REST API"
- "Define and Call Data Queries"
- "Upload Files to the Server"

## Access Data Objects By Using Scripts

IDM's uniform programming model means that all objects are queried and manipulated in the same way, using the Resource API. The URL or URI that is used to identify the target object for an operation depends on the object type. For an explanation of object types, see "*Data Models and Objects Reference*". For more information about scripts and the objects available to scripts, see "*Scripting Function Reference*" in the *Scripting Guide*.

You can use the Resource API to obtain managed, system, configuration, and repository objects, as follows:

```
val = openidm.read("managed/organization/mysampleorg")
val = openidm.read("system/mysystem/account")
val = openidm.read("config/custom/mylookuptable")
val = openidm.read("repo/custom/mylookuptable")
```

For information about constructing an object ID, see "URI Scheme" in the *REST API Reference*.

You can update entire objects with the `update()` function, as follows:

```
openidm.update("managed/organization/mysampleorg", rev, object)
openidm.update("system/mysystem/account", rev, object)
```

You can apply a partial update to a managed or system object by using the `patch()` function:

```
openidm.patch("managed/organization/mysampleorg", rev, value)
```

The `create()`, `delete()`, and `query()` functions work the same way.

## Access Data Objects By Using the REST API

IDM provides RESTful access to data objects through the ForgeRock Common REST API. To access objects over REST, you can use a browser-based REST client, such as the *Simple REST Client* for Chrome, or *RESTClient* for Firefox. Alternatively you can use the `curl` command-line utility.

For a comprehensive overview of the REST API, see the [REST API Reference](#).

To obtain a managed object through the REST API, depending on your security settings and authentication configuration, perform an HTTP GET on the corresponding URL, for example `http://localhost:8080/openidm/managed/organization/mysampleorg`.

By default, the HTTP GET returns a JSON representation of the object.

In general, you can map any HTTP request to the corresponding `openidm.method` call. The following example shows how the parameters provided in an `openidm.query` request correspond with the key-value pairs that you would include in a similar HTTP GET request:

Reading an object using the Resource API:

```
openidm.query("managed/user", { "_queryFilter": "true" }, ["userName", "sn"])
```

Reading an object using the REST API:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user?_queryFilter=true&_fields=userName,sn"
```

## Define and Call Data Queries

An advanced query model enables you to define queries and to call them over the REST or Resource API. Three types of queries are supported, on both managed, and system objects:

- Common filter expressions
- Parameterized, or predefined queries
- Native query expressions

Each of these mechanisms is discussed in the following sections.

### Tip

For limits on queries in progressive profiling, see "Custom Progressive Profile Conditions" in the *Self-Service Reference*.

## Common Filter Expressions

The ForgeRock REST API defines common filter expressions that enable you to form arbitrary queries using a number of supported filter operations. This query capability is the standard way to query data if no predefined query exists, and is supported for all managed and system objects.

Common filter expressions are useful in that they do not require knowledge of how the object is stored and do not require additions to the repository configuration.

Common filter expressions are called with the `_queryFilter` keyword. The following example uses a common filter expression to retrieve managed user objects whose user name is Smith:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
'http://localhost:8080/openidm/managed/user?_queryFilter=userName+eq+"smith"'
```

The filter is URL encoded in this example. The corresponding filter using the resource API would be:

```
openidm.query("managed/user", { "_queryFilter" : '/userName eq "smith"' });
```

Note that, this JavaScript invocation is internal and is not subject to the same URL-encoding requirements that a GET request would be. Also, because JavaScript supports the use of single quotes, it is not necessary to escape the double quotes in this example.

For a list of supported filter operations, see "Construct Queries".

Note that using common filter expressions to retrieve values from arrays is currently not supported. If you need to search within an array, you should set up a predefined (parameterized) in your repository configuration. For more information, see "Parameterized Queries".

## Parameterized Queries

You can access managed objects in *JDBC repositories* using custom parameterized queries. Define these queries in your JDBC repository configuration, (`repo.*.json`), and call them by their `_queryId`.

### Important

- Parameterized queries are not supported for system objects, or for DS repositories.

- All internal queries are filtered queries. Internal queries that reference a `queryId` are translated to filtered queries.

A typical query definition is as follows:

```
"query-all-ids" : "SELECT objectid FROM ${_dbSchema}.${_table} LIMIT ${int:_pageSize} OFFSET
${int:_pagedResultsOffset}",
```

To call this query, you would reference its ID, as follows:

```
?_queryId=query-all-ids
```

The following example calls `query-all-ids` over the REST interface:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
"http://localhost:8080/openidm/managed/user?_queryId=query-all-ids"
```

### Note

In `repo.jdbc.json`, the `queries` configuration object has a property, `validInRelationshipQuery`, which is an array specifying the IDs of queries that use relationships. If you define parameterized queries that you expect to use as part of a relationship query, you must add the query ID to this array. If no query IDs are specified or if the property is absent, relationship information is not returned in query results, even if requested. For more information about relationships, see *"Relationships Between Objects"*.

## Native Query Expressions

Native query expressions are supported for system objects only, and can be called directly.

You should only use native queries in situations where common query filters or parameterized queries are insufficient. For example, native queries are useful if the query needs to be generated dynamically.

The query expression is specific to the target resource and uses the native query language of that system resource.

Native queries are made using the `_queryExpression` keyword.

## Construct Queries

The `openidm.query` function lets you query managed and system objects. The query syntax is `openidm.query(id, params)`, where `id` specifies the object on which the query should be performed, and `params` provides the parameters that are passed to the query (the `_queryFilter`). For example:

```
var params = {
  '_queryFilter' : 'givenName co ' + sourceCriteria + ' or ' + 'sn co ' + sourceCriteria + '
};
var results = openidm.query("system/ScriptedSQL/account", params)
```

Over the REST interface, the query filter is specified as `_queryFilter=filter`, for example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=userName+eq+"Smith"'
```

### Note

In `_queryFilter` expressions, string values *must* use double-quotes. Numeric and boolean expressions should not use quotes.

When called over REST, you must URL encode the filter expression. The following examples show the filter expressions using the resource API and the REST API, but do not show the URL encoding, to make them easier to read.

For generic mappings, any fields that are included in the query filter (for example `userName` in the previous query), must be explicitly defined as *searchable*, if you have set the global `searchableDefault` to false. For more information, see "Improving Generic Mapping Search Performance (JDBC)".

The *filter* expression is constructed from the building blocks shown in this section. In these expressions the simplest *json-pointer* is a field of the JSON resource, such as `userName` or `id`. A JSON pointer can, however, point to nested elements.

### Note

You can also use the negation operator (!) in query construction. For example, a `_queryFilter=!(userName+eq+"jdoe")` query would return every `userName` except for `jdoe`.

## Comparison Expressions

You can set up query filters with the following expression types:

### + Objects That Equal a Specified Value

This is the associated JSON comparison expression: `json-pointer eq json-value`.

Consider the following example:

```
"_queryFilter" : '/givenName eq "Dan"'
```

The following REST call returns the user name and given name of all managed users whose first name (**givenName**) is "Dan":

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=givenName+eq
+ "Dan"&_fields=username,givenName'
{
  "result": [
    {
      "givenName": "Dan",
      "userName": "dlangdon"
    },
    {
      "givenName": "Dan",
      "userName": "dcope"
    },
    {
      "givenName": "Dan",
      "userName": "dlanoway"
    }
  ],
  "resultCount": 3,
  "pagedResultsCookie": null,
  "remainingPagedResults": -1
}
```

#### + Objects That Contain a Specified Value

This is the associated JSON comparison expression: **json-pointer co json-value**.

Consider the following example:

```
"_queryFilter" : '/givenName co "Da"'
```

The following REST call returns the user name and given name of all managed users whose first name (**givenName**) contains "Da":

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=givenName+co+"Da"&_fields=username,givenName'
{
  "result": [
    {
      "givenName": "Dave",
      "userName": "djensen"
    },
    {
      "givenName": "David",
```

```

    "userName": "dakers"
  },
  {
    "givenName": "Dan",
    "userName": "dlangdon"
  },
  {
    "givenName": "Dan",
    "userName": "dcope"
  },
  {
    "givenName": "Dan",
    "userName": "dlanoway"
  },
  {
    "givenName": "Daniel",
    "userName": "dsmith"
  },
  ...
],
"resultCount": 10,
"pagedResultsCookie": null,
"remainingPagedResults": -1
}

```

#### + Objects That Start With a Specified Value

This is the associated JSON comparison expression: `json-pointer sw json-value`.

Consider the following example:

```
"_queryFilter" : '/sn sw "Jen"'
```

The following REST call returns the user names of all managed users whose last name (`sn`) starts with "Jen":

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=sn+sw+"Jen"&_fields=userName'
{
  "result": [
    {
      "userName": "bjensen"
    },
    {
      "userName": "djensen"
    },
    {
      "userName": "cjenkins"
    },
    {
      "userName": "mjennings"
    }
  ],
  "resultCount": 4,
  "pagedResultsCookie": null,
  "remainingPagedResults": -1
}
```

#### + Objects That Are Less Than a Specified Value

This is the associated JSON comparison expression: `json-pointer lt json-value`.

Consider the following example:

```
"_queryFilter" : '/employeeNumber lt 5000'
```

The following REST call returns the user names of all managed users whose `employeeNumber` is lower than 5000:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=employeeNumber+lt+5000&_fields=userName,employeeNumber'
{
  "result": [
    {
      "employeeNumber": 4907,
      "userName": "jnorris"
    },
    {
      "employeeNumber": 4905,
      "userName": "afrancis"
    },
    {

```

```

    "employeeNumber": 3095,
    "userName": "twhite"
  },
  {
    "employeeNumber": 3921,
    "userName": "abasson"
  },
  {
    "employeeNumber": 2892,
    "userName": "dcarter"
  },
  ...
],
"resultCount": 4999,
"pagedResultsCookie": null,
"remainingPagedResults": -1
}

```

### + Objects That Are Less Than or Equal to a Specified Value

This is the associated JSON comparison expression: `json-pointer le json-value`.

Consider the following example:

```
"_queryFilter" : '/employeeNumber le 5000'
```

The following REST call returns the user names of all managed users whose `employeeNumber` is 5000 or less:

```

curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=employeeNumber+le+5000&_fields=username,employeeNumber'
{
  "result": [
    {
      "employeeNumber": 4907,
      "userName": "jnorris"
    },
    {
      "employeeNumber": 4905,
      "userName": "afrancis"
    },
    {
      "employeeNumber": 3095,
      "userName": "twhite"
    },
    {
      "employeeNumber": 3921,
      "userName": "abasson"
    }
  ],
  "resultCount": 4,
  "remainingPagedResults": -1
}

```

```

    "employeeNumber": 2892,
    "userName": "dcarter"
  },
  ...
],
"resultCount": 5000,
"pagedResultsCookie": null,
"remainingPagedResults": -1
}

```

### + Objects That Are Greater Than a Specified Value

This is the associated JSON comparison expression: *json-pointer gt json-value*

Consider the following example:

```

"_queryFilter" : '/employeeNumber gt 5000'

```

The following REST call returns the user names of all managed users whose *employeeNumber* is higher than 5000:

```

curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=employeeNumber+gt+5000&_fields=username,employeeNumber'
{
  "result": [
    {
      "employeeNumber": 5003,
      "userName": "agilder"
    },
    {
      "employeeNumber": 5011,
      "userName": "bsmith"
    },
    {
      "employeeNumber": 5034,
      "userName": "bjensen"
    },
    {
      "employeeNumber": 5027,
      "userName": "cclarke"
    },
    {
      "employeeNumber": 5033,
      "userName": "scarter"
    },
    ...
  ],
  "resultCount": 1458,
  "pagedResultsCookie": null,
  "remainingPagedResults": -1
}

```

```
}

```

### + Objects That Are Greater Than or Equal to a Specified Value

This is the associated JSON comparison expression: `json-pointer ge json-value`.

Consider the following example:

```
"_queryFilter" : '/employeeNumber ge 5000'
```

The following REST call returns the user names of all managed users whose `employeeNumber` is 5000 or greater:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=employeeNumber+ge+5000&_fields=username,employeeNumber'
{
  "result": [
    {
      "employeeNumber": 5000,
      "userName": "agilder"
    },
    {
      "employeeNumber": 5011,
      "userName": "bsmith"
    },
    {
      "employeeNumber": 5034,
      "userName": "bjensen"
    },
    {
      "employeeNumber": 5027,
      "userName": "cclarke"
    },
    {
      "employeeNumber": 5033,
      "userName": "scarter"
    },
    ...
  ],
  "resultCount": 1457,
  "pagedResultsCookie": null,
  "remainingPagedResults": -1
}
```

```
}
```

### Note

Certain system endpoints also support `EndsWith` and `ContainsAllValues` queries. However, such queries are *not supported* for managed objects and have not been tested with all supported ICF connectors.

## Presence Expressions

The following examples show how you can build filters using a presence expression, shown as `pr`. The presence expression is a filter that returns all records with a given attribute.

A presence expression filter evaluates to `true` when a *json-pointer* `pr` matches any object in which the *json-pointer* is present, and contains a non-null value. Consider the following expression:

```
"_queryFilter" : '/mail pr'
```

The following REST call uses that expression to return the mail addresses for all managed users with a `mail` property:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=mail+pr&_fields=mail'
{
  "result": [
    {
      "mail": "jdoe@exampleAD.com"
    },
    {
      "mail": "bjensen@example.com"
    }
  ],
  "resultCount": 2,
  "pagedResultsCookie": null,
  "remainingPagedResults": -1
}
```

Depending on the connector, you can apply the presence filter on system objects. The following query returns the email address of all users in a CSV file who have the `email` attribute in their entries:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/system/csvfile/account?_queryFilter=email+pr&_fields=email'
{
  "result": [
    {
      "_id": "bjensen",
      "email": "bjensen@example.com"
    },
    {
      "_id": "scarter",
      "email": "scarter@example.com"
    }
  ],
  "resultCount": 2,
  "pagedResultsCookie": "MA%3D%3D",
  "totalPagedResultsPolicy": "NONE",
  "totalPagedResults": -1,
  "remainingPagedResults": -1
}
```

### Note

Not all connectors support the presence filter. In most cases, you can replicate the behavior of the presence filter with an "equals" (eq) query such as `_queryFilter=email+eq"`

## Literal Expressions

A literal expression is a boolean:

- `true` matches any object in the resource.
- `false` matches no object in the resource.

For example, you can list the `_id` of all managed objects as follows:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=true&_fields=_id'
{
  "result": [
    {
      "_id": "d2e29d5f-0d74-4d04-bcfe-b1daf508ad7c"
    },
    {
      "_id": "709fed03-897b-4ff0-8a59-6faaa34e3af6"
    }
  ],
  "resultCount": 2,
  "pagedResultsCookie": null,
  "remainingPagedResults": -1
}
```

## In Expression Clause

IDM provides limited support for the **in** expression clause. You can use this clause for queries on singleton string properties, not arrays. **in** query expressions are not supported through the Admin UI.

The **in** operator is shorthand for multiple **OR** conditions.

### Note

The following example command includes escaped characters. For readability, the non-escaped URL syntax is:

```
http://localhost:8080/openidm/managed/user?_pageSize=1000&_fields=username&_queryFilter=/username+in+['user3a','user4a']'
```

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user?_pageSize=1000&_fields=username&_queryFilter=username%20in%20'%5B%22user4a%22%2C%22user3a%22%5D'"
{
  "result": [
    {
      "_id": "e32f9a3d-0039-4cb0-82d7-347cb808672e",
      "_rev": "000000000ae18357",
      "userName": "user3a"
    },
    {
      "_id": "120625c5-cfe7-48e7-b66a-6a0a0f9d2901",
      "_rev": "000000005ad98467",
      "userName": "user4a"
    }
  ],
  "resultCount": 2,
  "pagedResultsCookie": null,
  "totalPagedResultsPolicy": "NONE",
  "totalPagedResults": -1,
  "remainingPagedResults": -1
}
```

## Filter Expanded Relationships

You can use `_queryFilter` to directly filter expanded relationships from a collection, such as `authzRoles`. The following example queries the `manager-int` authorization role of a user:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user/b70293db-8743-45a7-9215-1ca8fd8a0073/authzRoles?_queryFilter=name+eq+'manager-int'&_fields=*"
{
  "result": [
    {
      "_id": "b1d78144-7029-4135-8e73-85efe0a40b6b",
      "_rev": "00000000d4b8ab97",
      "_ref": "internal/role/c0a38233-c0f2-477d-8f18-f5485b7d002f",
      "_refResourceCollection": "internal/role",
      "_refResourceId": "c0a38233-c0f2-477d-8f18-f5485b7d002f",
      "_refProperties": {
        "_grantType": "",
        "_id": "b1d78144-7029-4135-8e73-85efe0a40b6b",
        "_rev": "00000000d4b8ab97"
      }
    }
  ],
}
```

```

    "name": "manager-int",
    "description": "manager-int-desc",
    "temporalConstraints": null,
    "condition": null,
    "privileges": null
  }
],
"resultCount": 1,
"pagedResultsCookie": null,
"totalPagedResultsPolicy": "NONE",
"totalPagedResults": -1,
"remainingPagedResults": -1
}

```

## Complex Expressions

You can combine expressions using the boolean operators **and**, **or**, and **!** (not). The following example queries managed user objects located in London, with last name Jensen:

```

curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user/?_queryFilter=city+eq+"London"+and+sn+eq
+"Jensen"&_fields=username,givenName,sn'
{
  "result": [
    {
      "sn": "Jensen",
      "givenName": "Clive",
      "userName": "cjensen"
    },
    {
      "sn": "Jensen",
      "givenName": "Dave",
      "userName": "djensen"
    },
    {
      "sn": "Jensen",
      "givenName": "Margaret",
      "userName": "mjensen"
    }
  ],
  "resultCount": 3,
  "pagedResultsCookie": null,
  "remainingPagedResults": -1
}

```

## Page Query Results

The common filter query mechanism supports paged query results for managed objects, and for some system objects, depending on the system resource. There are two ways to page objects in a query:

- Using a cookie based on the value of a specified sort key.

- Using an offset that specifies how many records should be skipped before the first result is returned.

These methods are implemented with the following query parameters:

#### `_pagedResultsCookie`

Opaque cookie used by the server to keep track of the position in the search results. The format of the cookie is a base-64 encoded version of the value of the unique sort key property. The value of the returned cookie is URL-encoded to prevent values such as `+` from being incorrectly translated.

You cannot page results without sorting them (using the `_sortKeys` parameter). If you do not specify a sort key, the `_id` of the record is used as the default sort key. At least one of the specified sort key properties must be a unique value property, such as `_id`.

#### Tip

For paged searches on generic mappings with the default DS repository, you should sort on the `_id` property, as this is the only property that is stored outside of the JSON blob. If you sort on something other than `_id`, the search will incur a performance hit because IDM effectively has to pull the entire result set, and then sort it.

The server provides the cookie value on the first request. You should then supply the cookie value in subsequent requests until the server returns a null cookie, meaning that the final page of results has been returned.

The `_pagedResultsCookie` parameter is supported only for filtered queries, that is, when used with the `_queryFilter` parameter. You cannot use the `_pagedResultsCookie` with a `_queryId`.

The `_pagedResultsCookie` and `_pagedResultsOffset` parameters are mutually exclusive, and cannot be used together.

Paged results are enabled only if the `_pageSize` is a non-zero integer.

#### `_pagedResultsOffset`

Specifies the index within the result set of the number of records to be skipped before the first result is returned. The format of the `_pagedResultsOffset` is an integer value. When the value of `_pagedResultsOffset` is greater than or equal to 1, the server returns pages, starting after the specified index.

This request assumes that the `_pageSize` is set, and not equal to zero.

For example, if the result set includes 10 records, the `_pageSize` is 2, and the `_pagedResultsOffset` is 6, the server skips the first 6 records, then returns 2 records, 7 and 8. The `_remainingPagedResults` value would be 2, the last two records (9 and 10) that have not yet been returned.

If the offset points to a page beyond the last of the search results, the result set returned is empty.

## `_pageSize`

An optional parameter indicating that query results should be returned in pages of the specified size. For all paged result requests other than the initial request, a cookie should be provided with the query request.

The default behavior is not to return paged query results. If set, this parameter should be an integer value, greater than zero.

When a `_pageSize` is specified, and non-zero, the server calculates the `totalPagedResults`, in accordance with the `totalPagedResultsPolicy`, and provides the value as part of the response. If a count policy is specified (`_totalPagedResultsPolicy=EXACT`, The `totalPagedResults` returns the total result count. If no count policy is specified in the query, or if `_totalPagedResultsPolicy=NONE`, result counting is disabled, and the server returns a value of -1 for `totalPagedResults`. The following example shows a query that requests two results with a `totalPagedResultsPolicy` of `EXACT`:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/managed/user?
_queryFilter=true&_pageSize=2&_totalPagedResultsPolicy=EXACT"
{
  "result": [
    {
      "_id": "adonnelly",
      "_rev": "0",
      "userName": "adonnelly",
      "givenName": "Abigail",
      "sn": "Donnelly",
      "telephoneNumber": "12345678",
      "active": "true",
      "mail": "adonnelly@example.com",
      "accountStatus": "active",
      "effectiveRoles": [],
      "effectiveAssignments": []
    },
    {
      "_id": "bjensen",
      "_rev": "0",
      "userName": "bjensen",
      "givenName": "Babs",
      "sn": "Jensen",
      "telephoneNumber": "12345678",
      "active": "true",
      "mail": "bjensen@example.com",
      "accountStatus": "active",
      "effectiveRoles": [],
      "effectiveAssignments": []
    }
  ],
  "resultCount": 2,
  "pagedResultsCookie": "eyJvX2lkIjoiYm11cnJheSJ9",
  "totalPagedResultsPolicy": "EXACT",
  "totalPagedResults": 22,
```

```
"remainingPagedResults": -1  
}
```

The `totalPagedResults` and `_remainingPagedResults` parameters are not supported for all queries. Where they are not supported, their returned value is always `-1`. In addition, counting query results using these parameters is not currently supported for a ForgeRock Directory Services (DS) repository.

Requesting the total result count (with `_totalPagedResultsPolicy=EXACT`) incurs a performance cost on the query.

Queries that return large data sets will have a significant impact on heap requirements, particularly if they are run in parallel with other large data requests. To avoid out of memory errors, analyze your data requirements, set the heap configuration appropriately, and modify access controls to restrict requests on large data sets.

## Sort Query Results

For common filter query expressions, you can sort the results of a query using the `_sortKeys` parameter. This parameter takes a comma-separated list as a value and orders the way in which the JSON result is returned, based on this list.

The `_sortKeys` parameter is not supported for predefined queries.

### Note

When using DS as a repo, pagination using `_pageSize` is recommended if you intend to use `_sortKeys`. If you do not plan to paginate your query, the data you are querying must at least be indexed in DS. For more information about how to set up indexes in DS, see *Indexes in the DS Configuration Guide*.

The following query returns all users with the `givenName` Dan, and sorts the results alphabetically, according to surname (`sn`):

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/system/ldap/account?_queryFilter=givenName+eq+
+\"Dan\"&_fields=givenName,sn&_sortKeys=sn'
{
  "result": [
    {
      "sn": "Cope",
      "givenName": "Dan"
    },
    {
      "sn": "Langdon",
      "givenName": "Dan"
    },
    {
      "sn": "Lanoway",
      "givenName": "Dan"
    }
  ],
  "resultCount": 3,
  "pagedResultsCookie": null,
  "remainingPagedResults": -1
}
```

### Note

When you query a relationship field, fields that belong to the related object are not available as `_sortKeys`. For example, if you query a list of a manager's reports, you cannot sort by the reports' last names. This is because the available `_sortKeys` are based on the object being queried, which, in the case of relationships, is actually a list of references to other objects, not the objects themselves.

## Recalculate Virtual Property Values in Queries

For managed objects IDM includes an `onRetrieve` script hook that enables you to recalculate property values when an object is retrieved as the result of a query. To use the `onRetrieve` trigger, the query must include the `executeOnRetrieve` parameter, for example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=sn+eq+\"Jensen\"&executeOnRetrieve=true'
```

If a query includes `executeOnRetrieve`, the query recalculates virtual property values, based on the current state of the system. The result of the query will be the same as a `read` on a specific object, because reads always recalculate virtual property values.

If a query does not include `executeOnRetrieve`, the query returns the virtual properties of an object, based on the value that is persisted in the repository. Virtual property values are not recalculated.

For performance reasons, `executeOnRetrieve` is `false` by default.

#### Note

Virtual properties that use `queryConfig` for calculation instead of an `onRetrieve` script are not recalculated by `executeOnRetrieve`. These properties are recalculated only when there is a change (such as adding or removing a role affecting `effectiveRoles`, or a temporal constraint being triggered or changed).

## Upload Files to the Server

IDM provides a generic file upload service that enables you to upload and save files either to the filesystem or to the repository. The service uses the `multipart/form-data` Content-Type to accept file content, store it, and return that content when it is called over the REST interface.

To configure the file upload service, add one or more `file-description.json` files to your project's `conf` directory, where `description` provides an indication of the purpose of the upload service. For example, you might create a `file-images.json` configuration file to handle uploading image files. Each file upload configuration file sets up a separate instance of the upload service. The `description` in the filename also specifies the endpoint at which the file service will be accessible over REST. In the previous example, `file-images.json`, the service would be accessible at the endpoint `openidm/file/images`.

A sample file upload service configuration file is available in the `/path/to/openidm/samples/example-configurations/conf` directory. The configuration is as follows:

```
{
  "enabled" : true,
  "fileHandler" : {
    "type" : file handler type,
    "root" : directory
  }
}
```

The service supports two *file handlers*—`file` and `repo`. The file handlers are configured as follows:

- `"type" : "file"` specifies that the uploaded content will be stored in the filesystem. If you use the `file` type, you must specify a `root` property to indicate the directory (relative to the IDM installation directory) in which uploaded content is stored. In the following example, uploaded content is stored in the `/path/to/openidm/images` directory:

```
{
  "enabled" : true,
  "fileHandler" : {
    "type" : "file",
    "root" : "images"
  }
}
```

You cannot use the file upload service to access any files outside the configured `root` directory.

**Warning**

If `root` is configured to be an empty string, do not grant access to the file upload service to end users. When `type` is configured as `file`, ensure that `root` is configured to be a directory.

- `"type" : "repo"` specifies that the uploaded content will be stored in the repository. The `root` property does not apply to the repository file handler so the configuration is as follows:

```
{
  "enabled" : true,
  "fileHandler" : {
    "type" : "repo"
  }
}
```

The file upload service performs a multi-part CREATE operation. Each upload request includes two `--form` options. The first option indicates that the uploaded file content will be converted to a base 64-encoded string and inserted into the JSON object as a field named `content` with the following structure:

```
{
  "content" : {
    "$ref" : "cid:filename#content"
  }
}
```

The second `--form` option specifies the file to be uploaded, and the file type. The request loads the entire file into memory, so file size will be constrained by available memory.

You can upload any mime type using this service, however, you must specify a safelist of mime types that can be *retrieved* over REST. If you specify a mime type that is not in the safelist during retrieval of the file, the response content defaults to `application/json`. To configure the list of supported mime types, specify a comma-separated list as the value of the `org.forgerock.json.resource.http.safemimetypes` property in the `conf/system.properties` file. For example:

```
org.forgerock.json.resource.http.safemimetypes=application/json,application/pkix-cert,application/x-pem-file
```

You can only select from the following list:

- `image/*`
- `text/plain`
- `text/css`
- `application/json`
- `application/pkix-cert`

- `application/x-pem-file`

The following request uploads an image (PNG) file named `test.png` to the filesystem. The file handler configuration file provides the REST endpoint. In this case `openidm/file/images` references the configuration in the `file-images.json` file:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--form 'json={"content" : {"$ref" : "cid:test#content"}};type=application/json' \
--form 'test=@test.png;type=image/png' \
--request PUT \
"http://localhost:8080/openidm/file/images/test.png"
{
  "_id": "test.png",
  "content": "aW1hZ2UvcG5n"
}
```

Note that the resource ID is derived directly from the upload filename—system-generated IDs are not supported.

The following request uploads a stylesheet (css) file named `test.css` to the same location on the filesystem as the previous request:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--form 'json={"content" : {"$ref" : "cid:test#content"}};type=application/json' \
--form '@test.css;type=text/css' \
--request PUT \
"http://localhost:8080/openidm/file/images/test.css"
{
  "_id": "test.css",
  "content": "aW1hZ2UvY3N2"
}
```

Files uploaded to the repository are stored as JSON objects in the `openidm.files` table. The following request uploads the same image (PNG) file (`test.png`) to the repository:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--form 'json={"content" : {"$ref" : "cid:test#content"}};type=application/json' \
--form 'test=@test.png;type=image/png' \
--request PUT \
"http://localhost:8080/openidm/file/repo/test.png"
{
  "_id": "test.png",
  "_rev": "00000000970b4454",
  "content": "aW1hZ2UvcG5n"
}
```

Note that the preceding example assumes the following file upload service configuration (in `file-repo.json`):

```
{
  "enabled" : true,
  "fileHandler" : {
    "type" : "repo"
  }
}
```

The file type is not stored with the file. By default, a READ on uploaded file content returns the content as a base 64-encoded string within the JSON object. For example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/file/repo/test.png"
{
  "_id": "test.png",
  "_rev": "00000000970b4454",
  "content": "aW1hZ2UvcG5n"
}
```

Your client can retrieve the file in the correct format by specifying the `content` and `mimeType` parameters in the read request. For example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/file/repo/test.css?_fields=content&_mimeType=text/css"
```

To delete uploaded content, send a DELETE request as follows:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request DELETE \
"http://localhost:8080/openidm/file/repo/test.png"
{
  "_id": "test.png",
  "_rev": "00000000970b4454",
  "content": "aW1hZ2UvcG5n"
}
```

## Chapter 7

# Import Bulk Data

The bulk import facility lets you import large numbers of external entries over REST. You import entries from a comma-separated values (CSV) file, to a specified managed object type in the IDM repository. Bulk import works as follows:

- Loads bulk CSV entries and stores them temporarily (in the IDM repository) as JSON objects
- Creates a temporary mapping between those entries and the managed object store in the repository
- Performs a reconciliation between the JSON objects and the objects in the repository

### Note

The bulk import mechanism assumes that the CSV file is the *authoritative* data source. If you run an import more than once, the import overwrites all of the properties of the managed object (including timestamps) with the values in the CSV file.

To import bulk CSV entries into the repository, using the REST API, follow these steps:

#### + *Generate a CSV Template*

The first time you upload entries, you must generate a CSV template. The template is essentially an empty CSV file with one header row that matches the managed object type to which you are importing. In most cases, you will be importing data that fits the **managed/user** object model, but you can import any managed object type, such as roles and assignments.

To generate the CSV template, send a GET request to the **openidm/csv/template** endpoint. The following request generates a CSV template for the managed user object type:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
"http://localhost:8080/openidm/csv/template?resourceCollection=managed/
user&_fields=header&_mimeType='text/plain'"
{
  "id": "template",
  "header": "\"userName\\",\"givenName\\",\"sn\\",\"mail\\",\"description\\",\"accountStatus\\",
\"telephoneNumber\\",
\"postalAddress\\",\"city\\",\"postalCode\\",\"country\\",\"stateProvince\\",\"preferences/updates\\",
\"preferences/marketing\""
}
```

The template is generated based on the specified `resourceCollection`, and includes a single header row. The names of each header column are derived from the schema of the managed object type. The template includes only a subset of managed user properties that can be represented by CSV fields.

Only the following managed object properties are included in the header row:

- Properties of type `string`, `boolean`, and `number`
- Properties that do *not* start with an underscore (such as `_id` or `_rev`)

If you are importing entries to `managed/user`, the bulk import facility assumes that self-service password reset is enabled. This is because the import does not support upload of hashed passwords.

- Properties whose `scope` is not `private`

Set the parameters `_fields=header` and `_mimeType=text/csv` to download the template as a CSV file.

When you have generated the template, export your external data to CSV format, using the headers in the generated template.

## + Upload a CSV File

The default maximum file size for bulk import is 50MBytes. If you need to import a number of records that exceeds this size, divide the data into chunks and import each file separately. You can also increase the maximum file size by changing the value of the `maxRequestSizeInMegabytes` property in your `conf/servletfilter-upload.json` file.

When you have a CSV file, with the structure of the template generated in the previous example, upload the file to the IDM repository with the following request:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--form upload=@/path/to/example-users.csv \
--request POST \
"http://localhost:8080/upload/csv/managed/user?uniqueProperty=userName"
{
  "importUUIDs": [
    "3ebd514f-bdd7-491f-928f-21b72f44e381"
  ]
}
```

### --form (-F)

This option causes `curl` to POST data using the Content-Type `multipart/form-data`, which lets you upload binary files. To indicate that the form content is a file, prefix the file name with an `@` sign.

To import more than one file at once, specify multiple `--form` options, for example:

```
--form upload=@/path/to/example-users-a-j.csv \  
--form upload=@/path/to/example-users-k-z.csv \  

```

### **uniqueProperty (required)**

This parameter lets you correlate existing entries, based on a unique value field. This is useful if you need to upload the same file a number of times (for example, if data in the file changes, or if some entries in the file contained errors). You can specify any unique value property here. You can also correlate on more than one property by specifying multiple, comma-delimited unique properties.

A successful upload generates an array of `importUUIDs`. You need these UUIDs to perform other operations on the import records.

### **Important**

Note that the endpoint (`upload/csv`) is not an IDM endpoint.

## + Query Bulk Imports

A query on the `csv/metadata` endpoint returns the import ID, the data structure (header fields in the CSV file), a recon ID, and a number of fields indicating the status of the import:

```
curl \  
--header "X-OpenIDM-Username: openidm-admin" \  
--header "X-OpenIDM-Password: openidm-admin" \  
--header "Accept-API-Version: resource=1.0" \  
--request GET \  
"http://localhost:8080/openidm/csv/metadata/?_queryFilter=true"  
{  
  "result": [  
    {  
      "_id": "3ebd514f-bdd7-491f-928f-21b72f44e381",  
      "_rev": "0000000003e8ef4f7",  
      "header": [  
        "userName",  
        "givenName",  
        "sn",  
        "mail",  
        "description",  
        "accountStatus",  
        "country"  
      ],  
      "reconId": "2e2cf41a-c4b8-4dda-9d92-6e0af65a15fe-6528",  
      "filename": "example-users.csv",  
      "resourcePath": "managed/user",  
      "total": 1000,  
      "success": 1000,  
      "failure": 0,  
    }  
  ]  
}
```

```

    "created": 1000,
    "updated": 0,
    "unchanged": 0,
    "begin": "2020-04-17T16:31:02.955Z",
    "end": "2020-04-17T16:31:09.861Z",
    "cancelled": false,
    "importDeleted": false,
    "tempRecords": 0,
    "purgedTempRecords": true,
    "purgedErrorRecords": false,
    "authId": "openidm-admin",
    "authzComponent": "internal/user"
  },
  {
    "_rev": "00000000d4392fc8"
  }
],
...
}

```

### + Query Imports To a Specific Object Type

Use a query filter to restrict your query to imports to a specific managed object type. The following example queries uploads to the managed user object:

```

curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request GET \
'http://localhost:8080/openidm/csv/metadata/?_queryFilter=/resourcePath+eq+"managed/user"'
{
  "result": [
    {
      "_id": "82d9a643-8b03-4cec-86fc-3e09c4c2f01c",
      "_rev": "000000009b3ff60b",
      "header": [
        "userName",
        "givenName",
        "sn",
        "mail",
        "description",
        "accountStatus",
        "country"
      ],
      "reconId": "417dae3b-c939-4191-acbf-6eb1b9e802af-53335",
      "filename": "example-users.csv",
      "resourcePath": "managed/user",
      "total": 1001,
      "success": 1000,
      "failure": 1,
      "created": 0,
      "updated": 0,
      "unchanged": 1000,
      "begin": "2020-04-20T13:12:03.028Z",
      "end": "2020-04-20T13:12:05.222Z",
    }
  ]
}

```

```
"cancelled": false,
"importDeleted": false,
"tempRecords": 0,
"purgedTempRecords": true,
"purgedErrorRecords": false,
"authId": "openidm-admin",
"authzComponent": "internal/user"
}
},
...
}
```

### + Handle Failed Import Records

The previous example showed the statistics that are returned when you query bulk imports. One of these fields is `"failure": 0`. If the import was unsuccessful for any records, this `failure` field will have a positive value. You can then download the failed records, examine the failures and correct them in the CSV file, then run the import again.

To download failed records, send a GET request to the endpoint `export/csvImportFailures/importUUID`:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--request GET \
--header "Accept-API-Version: resource=1.0" \
"http://localhost:8080/export/csvImportFailures/82d9a643-8b03-4cec-86fc-3e09c4c2f01c"

userName, givenName, sn, mail, ..., _importError
emacheke, Edward, Macheke, emacheke, ..., "{code=403, reason=Forbidden, message=Policy
validation
failed, detail={result=false, failedPolicyRequirements=[{policyRequirements=[
{policyRequirement=VALID_EMAIL_ADDRESS_FORMAT}], property=mail}}}"
```

The output indicates the failed record or records, and the reason for the failure, in the `_importError` field. In this example, the import failed because of a policy validation error—the email address is not the correct format.

#### Warning

IDM does not scan for possible CSV injection attacks on uploaded files. *Do not* edit the downloaded CSV file with Microsoft Excel, as this can expose your data to CSV injection.

### + Cancel an Import in Progress

Cancel an import that is in progress by sending a POST request to the `openidm/csv/metadata/importUUID` endpoint, with the `cancel` action. You might want to cancel an import if the import is taking too long, or if you have noticed problems with the import data, for example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request POST \
"http://localhost:8080/openidm/csv/metadata/92971c92-67bb-4ae7-b41b-96d249b0b2aa/?_action=cancel"
{
  "status": "OK"
}
```

#### + HTTP Request Timeout

By default, the timeout for the bulk import servlets is 30 seconds (or 30000 milliseconds). This parameter is set in your `resolver/boot.properties` file, as follows:

```
openidm.servlet.timeoutMillis=30000
```

If you are importing a very large number of records, you might need to increase the HTTP request timeout to prevent requests timing out.

In test environments, you can set this parameter to 0 to disable the request timeout. You should *not* disable the timeout in a production environment because no timeout can lead to DDoS attacks where thousands of slow HTTP connections are made.

For a list of all REST endpoints related to bulk import, see "Bulk Import" in the *REST API Reference*.

# Appendix A. Data Models and Objects Reference

You can customize a variety of objects that can be addressed via a URL or URI. IDM can perform a common set of functions on these objects, such as CRUDPAQ (create, read, update, delete, patch, action, and query).

Depending on how you intend to use them, different object types are appropriate.

## *Object Types*

| Object Type           | Intended Use   | Special Functionality  |
|-----------------------|--|--|
| Managed objects       | Serve as targets and sources for synchronization, and to build virtual identities.   | Provide appropriate auditing, script hooks, declarative mappings and so forth in addition to the REST interface. |
| Configuration objects | Ideal for look-up tables or other custom configuration, which can be configured externally like any other system configuration.              | Adds file view, REST interface, and so forth   |
| Repository objects    | The equivalent of arbitrary database table access. Appropriate for managing data purely through the underlying data store or repository API. | Persistence and API access   |
| System objects        | Representation of target resource objects, such as accounts, but also resource objects such as groups.                                       |  |
| Audit objects         | Houses audit data in the repository.   |  |

| Object Type | Intended Use                            | Special Functionality |
|-------------|---|-----------------------|
| Links       | Defines a relation between two objects. |                       |

## Managed Objects

A *managed object* is an object that represents the identity-related data managed by IDM. Managed objects are stored in the IDM repository. All managed objects are JSON-based data structures.

### Managed Object Schema

IDM provides a default schema for typical managed object types, such as users and roles, but does not control the structure of objects that you store in the repository. You can modify or extend the schema for the default object types, and you can set up a new managed object type for any item that can be collected in a data set.

The `_rev` property of a managed object is reserved for internal use, and is not explicitly part of its schema. This property specifies the revision of the object in the repository. This is the same value that is exposed as the object's ETag through the REST API. The content of this attribute is not defined. No consumer should make any assumptions of its content beyond equivalence comparison. This attribute may be provided by the underlying data store.

Schema validation is performed by the `policy` service and can be configured according to the requirements of your deployment.

Properties can be defined to be strictly derived from other properties within the object. This allows computed and composite values to be created in the object. Such properties are named *virtual properties*. The value of a virtual property is computed only when that property is retrieved.

### Data Consistency

Single-object operations are consistent within the scope of the operation performed, limited by the capabilities of the underlying data store. Bulk operations have no consistency guarantees. IDM does not expose any transactional semantics in the managed object access API.

For information on conditional header access through the REST API, see "Conditional Operations" in the *REST API Reference*.

### Managed Object Triggers

*Triggers* are user-definable functions that validate or modify object or property state.

## State Triggers

Managed objects are resource-oriented. A set of triggers is defined to intercept the supported request methods on managed objects. Such triggers are intended to perform authorization, redact, or modify objects before the action is performed. The object being operated on is in scope for each trigger, meaning that the object is retrieved by the data store before the trigger is fired.

If retrieval of the object fails, the failure occurs before any trigger is called. Triggers are executed before any optimistic concurrency mechanisms are invoked. The reason for this is to prevent a potential attacker from getting information about an object (including its presence in the data store) before authorization is applied.

### **onCreate**

Called upon a request to create a new object. Throwing an exception causes the create to fail.

### **postCreate**

Called after the creation of a new object is complete.

### **onRead**

Called upon a request to retrieve a whole object or portion of an object. Throwing an exception causes the object to not be included in the result. This method is also called when lists of objects are retrieved via requests to its container object; in this case, only the requested properties are included in the object. Allows for uniform access control for retrieval of objects, regardless of the method in which they were requested.

### **onUpdate**

Called upon a request to store an object. The `oldObject` and `newObject` variables are in-scope for the trigger. The `oldObject` represents a complete object, as retrieved from the data store. The trigger can elect to change `newObject` properties. If, as a result of the trigger, the values of the `oldObject` and `newObject` are identical (that is, update is reverted), the update ends prematurely, but successfully. Throwing an exception causes the update to fail.

### **postUpdate**

Called after an update request is complete.

### **onDelete**

Called upon a request to delete an object. Throwing an exception causes the deletion to fail.

### **postDelete**

Called after an object is deleted.

### **onSync**

Called when a managed object is changed, and the change triggers an implicit synchronization operation. The implicit synchronization operation is triggered by calling the sync service, which

attempts to go through all the configured managed-system mappings. The sync service returns either a response or an error. For both the response and the error, the script that is referenced by the `onSync` hook is called.

You can use this hook to inject business logic when the sync service either fails or succeeds to synchronize all applicable mappings. For an example of how the `onSync` hook is used to revert partial successful synchronization operations, see "Synchronization Failure Compensation" in the *Synchronization Guide*.

## Object Storage Triggers

An object-scoped trigger applies to an entire object. Unless otherwise specified, the object itself is in scope for the trigger.

### **onValidate**

Validates an object prior to its storage in the data store. If an exception is thrown, the validation fails and the object is not stored.

### **onStore**

Called just prior to when an object is stored in the data store. Typically used to transform an object just prior to its storage (for example, encryption).

## Property Storage Triggers

A property-scoped trigger applies to a specific property within an object. Only the property itself is in scope for the trigger. No other properties in the object should be accessed during execution of the trigger. Unless otherwise specified, the order of execution of property-scoped triggers is intentionally left undefined.

### **onValidate**

Validates a given property value after its retrieval from and prior to its storage in the data store. If an exception is thrown, the validation fails and the property is not stored.

### **onRetrieve**

Called on all requests that return a single object: read, create, update, patch, and delete.

`onRetrieve` is called on queries only if `executeOnRetrieve` is set to `true` in the query request parameters. If `executeOnRetrieve` is not passed, or if it is `false`, the query returns previously persisted values of the requested fields. This behavior avoids performance problems when executing the script on all results of a query.

### **onStore**

Called before an object is stored in the data store. Typically used to transform a given property before its object is stored.

## Storage Trigger Sequences

Triggers are executed in the following order:

### *Object Retrieval Sequence*

1. Retrieve the raw object from the data store
2. The `executeOnRetrieve` boolean is used to check whether property values should be recalculated. The sequence continues if the boolean is set to `true`.
3. Call object `onRetrieve` trigger
4. Per-property within the object, call property `onRetrieve` trigger

### *Object Storage Sequence*

1. Per-property within the object:
  - Call property `onValidate` trigger
  - Call object `onValidate` trigger
2. Per-property trigger within the object:
  - Call property `onStore` trigger
  - Call object `onStore` trigger
  - Store the object with any resulting changes to the data store

## Managed Object Encryption

Sensitive object properties can be encrypted prior to storage, typically through the property `onStore` trigger. The trigger has access to configuration data, which can include arbitrary attributes that you define, such as a symmetric encryption key. Such attributes can be decrypted during retrieval from the data store through the property `onRetrieve` trigger.

## Managed Object Configuration

Configuration of managed objects is provided through an array of managed object configuration objects.

```
{
  "objects": [ managed-object-config object, ... ]
}
```

## objects

array of managed-object-config objects, required

Specifies the objects that the managed object service manages.

### *Managed-Object-Config Object Properties*

Specifies the configuration of each managed object.

```
{
  "name"      : string,
  "actions"   : script object,
  "onCreate"  : script object,
  "onDelete"  : script object,
  "onRead"    : script object,
  "onRetrieve": script object,
  "onStore"   : script object,
  "onSync"    : script object,
  "onUpdate"  : script object,
  "onValidate": script object,
  "postCreate": script object,
  "postDelete": script object,
  "postUpdate": script object,
  "schema"    : {
    "id"       : urn,
    "icon"     : string,
    "mat-icon" : string,
    "order"    : [ list of properties ],
    "properties": { property-configuration objects },
    "$schema"  : "http://json-schema.org/draft-03/schema",
    "title"    : "User",
    "viewable" : true
  }
}
```

## name

string, required

The name of the managed object. Used to identify the managed object in URIs and identifiers.

## actions

script object, optional

A custom script that initiates an action on the managed object. For more information, see *"Register Custom Scripted Actions"* in the *Scripting Guide*.

## onCreate

script object, optional

A script object to trigger when the creation of an object is being requested. The object to be created is provided in the root scope as an **object** property. The script can change the object. If an exception is thrown, the create aborts with an exception.

## onDelete

script object, optional

A script object to trigger when the deletion of an object is being requested. The object being deleted is provided in the root scope as an **object** property. If an exception is thrown, the deletion aborts with an exception.

## onRead

script object, optional

A script object to trigger when the read of an object is being requested. The object being read is provided in the root scope as an **object** property. The script can change the object. If an exception is thrown, the read aborts with an exception.

## onRetrieve

script object, optional

A script object to trigger when an object is retrieved from the repository. The object that was retrieved is provided in the root scope as an **object** property. The script can change the object. If an exception is thrown, then object retrieval fails.

## onStore

script object, optional

A script object to trigger when an object is about to be stored in the repository. The object to be stored is provided in the root scope as an **object** property. The script can change the object. If an exception is thrown, then object storage fails.

## onSync

script object, optional

A script object to trigger when a change to a managed object triggers an implicit synchronization operation. The script has access to the **syncResults** object, the **request** object, the state of the object before the change (**oldObject**) and the state of the object after the change (**newObject**). The script can change the object.

## onUpdate

script object, optional

A script object to trigger when an update to an object is requested. The old value of the object being updated is provided in the root scope as an **oldObject** property. The new value of the object being updated is provided in the root scope as a **newObject** property. The script can change the **newObject**. If an exception is thrown, the update aborts with an exception.

## **onValidate**

script object, optional

A script object to trigger when the object requires validation. The object to be validated is provided in the root scope as an **object** property. If an exception is thrown, the validation fails.

## **postCreate**

script object, optional

A script object to trigger after an object is created, but before any targets are synchronized.

## **postDelete**

script object, optional

A script object to trigger after a delete of an object is complete, but before any further synchronization. The value of the deleted object is provided in the root scope as an **oldObject** property.

## **postUpdate**

script object, optional

A script object to trigger after an update to an object is complete, but before any targets are synchronized. The value of the object before the update is provided in the root scope as an **oldObject** property. The value of the object after the update is provided in the root scope as a **newObject** property.

## **schema**

json-schema object, optional

The schema to use to validate the structure and content of the managed object, and how the object is displayed in the UI. The schema-object format is defined by the JSON Schema specification.

The **schema** property includes the following additional elements:

### **icon**

string, optional

The name of the Font Awesome icon to display for this object in the UI. Only applies to standalone IDM.

### **mat-icon**

string, optional

The name of the [Material Design Icon](#) to display for this object in the UI. Only applies to IDM as part of the ForgeRock Identity Platform.

**id**

urn, optional

The URN of the managed object, for example,  
`urn:jsonschema:org:forgerock:openidm:managed:api:Role`.

**order**

list of properties, optional

The order in which properties of this managed object are displayed in the UI.

**properties**

list of property configuration objects, optional

A list of property specifications. For more information, see [Property Configuration Properties](#).

**\$schema**

url, optional

Link to the JSON schema specification.

**title**

string, optional

The title of this managed object in the UI.

**viewable**

boolean, optional

Whether this object is visible in the UI.

### *Property Configuration Properties*

Each managed object property, identified by its `property-name`, can have the following configurable properties:

```
"property-name" : {  
  "description"   : string,  
  "encryption"    : property-encryption object,  
  "isPersonal"    : boolean true/false,
```

```
"isProtected"      : boolean true/false,
"isVirtual"        : boolean true/false,
"items"           : {
  "id"              : urn,
  "properties"      : property-config object,
  "resourceCollection" : property-config object,
  "reversePropertyName" : string,
  "reverseRelationship" : boolean true/false,
  "title"           : string,
  "type"            : string,
  "validate"        : boolean true/false,
},
"onRetrieve"       : script object,
"onStore"          : script object,
"onValidate"       : script object,
"pattern"          : string,
"policies"         : policy object,
"required"         : boolean true/false,
"returnByDefault"  : boolean true/false,
"scope"            : string,
"searchable"       : boolean true/false,
"secureHash"       : property-hash object,
"title"            : string,
"type"             : data type,
"usageDescription" : string,
"userEditable"     : boolean true/false,
"viewable"         : boolean true/false,
}
```

## description

string, optional

A brief description of the property.

## encryption

property-encryption object, optional

Specifies the configuration for encryption of the property in the repository. If omitted or null, the property is not encrypted.

## isPersonal

boolean, true/false

Designed to highlight personally identifying information. By default, `isPersonal` is set to `true` for `userName` and `postalAddress`.

## isProtected

boolean, true/false

Specifies whether reauthentication is required if the value of this property changes.

## isVirtual

boolean, true/false

Specifies whether the property takes a static value, or whether its value is calculated dynamically as the result of a script.

The most recently calculated value of a virtual property is persisted by default. The persistence of virtual property values allows IDM to compare the new value of the property against the last calculated value, and therefore to detect change events during synchronization.

Virtual property values are not persisted by default if you are using an explicit mapping.

## items

property-configuration object, optional

For `array` type properties, defines the elements in the array. `items` can include the following sub-properties:

### id

urn, optional

The URN of the property, for example,

`urn:jsonschema:org:forgerock:openidm:managed:api:Role:members:items.`

## properties

property configuration object, optional

A list of properties, and their configuration, that make up this items array. For example, for a relationship type property:

```
"properties" : {
  "_ref" : {
    "description" : "References a relationship from a managed object",
    "type" : "string"
  },
  "_refProperties" : {
    "description" : "Supports metadata within the relationship",
    ...
  }
}
```

## resourceCollection

property configuration object, optional

The collection of resources (objects) on which this relationship is based (for example, `managed/user` objects).

**reversePropertyName**

string, optional

For **relationship** type properties, specifies the corresponding property name in the case of a reverse relationship. For example, a **roles** property might have a **reversePropertyName** of **members**.

**reverseRelationship**

boolean, true or false.

For **relationship** type properties, specifies whether the relationship exists in both directions.

**title**

string, optional

The title of array items, as displayed in the UI, for example **Role Members Items**.

**type**

string, optional

The array type, for example **relationship**.

**validate**

boolean, true/false

For reverse relationships, specifies whether the relationship should be validated.

**onRetrieve**

script object, optional

A script object to trigger once a property is retrieved from the repository. That property may be one of two related variables: **property** and **propertyName**. The property that was retrieved is provided in the root scope as the **propertyName** variable; its value is provided as the **property** variable. If an exception is thrown, then object retrieval fails.

**onStore**

script object, optional

A script object to trigger when a property is about to be stored in the repository. That property may be one of two related variables: **property** and **propertyName**. The property that was retrieved is provided in the root scope as the **propertyName** variable; its value is provided as the **property** variable. If an exception is thrown, then object storage fails.

**onValidate**

script object, optional

A script object to trigger when the property requires validation. The value of the property to be validated is provided in the root scope as the `property` property. If an exception is thrown, validation fails.

### **pattern**

string, optional

Any specific pattern to which the value of the property must adhere. For example, a property whose value is a date might require a specific date format. Patterns specified here must follow regular expression syntax.

### **policies**

policy object, optional

Any policy validation that must be applied to the property.

### **required**

boolean, true/false

Specifies whether the property must be supplied when an object of this type is created.

### **returnByDefault**

boolean, true/false

For virtual properties, specifies whether the property will be returned in the results of a query on an object of this type if it is not explicitly requested. Virtual attributes are not returned by default.

### **scope**

string, optional

Specifies whether the property should be filtered from HTTP/external calls. The value can be either `"public"` or `"private"`. `"private"` indicates that the property should be filtered, `"public"` indicates no filtering. If no value is set, the property is assumed to be public and thus not filtered.

### **searchable**

boolean, true/false

Specifies whether this property can be used in a search query on the managed object. A searchable property is visible in the End User UI. False by default.

### **secureHash**

property-hash object, optional

Specifies the configuration for hashing of the property value in the repository. If omitted or null, the property is not hashed.

**title**

string, required

A human-readable string, used to display the property in the UI.

**type**

data type, required

The data type for the property value; can be String, Array, Boolean, Number, Object, or Resource Collection.

**usageDescription**

string, optional

Designed to help end users understand the sensitivity of a property such as a telephone number.

**userEditable**

boolean, true/false

Specifies whether users can edit the property value in the UI. This property applies in the context of the End User UI, in which users are able to edit certain properties of their own accounts. False by default.

**viewable**

boolean, true/false

Specifies whether this property is viewable in the object's profile in the UI. True by default.

***Script Object Properties***

```
{  
  "type" : "text/javascript",  
  "source": string  
}
```

**type**

string, required

Specifies the type of script to be executed. Supported types include `"text/javascript"` and `"groovy"`.

**source, file**

string, required (only one, source or file is required)

Specifies the source code of the script to be executed (if the keyword is "source"), or a pointer to the file that contains the script (if the keyword is "file").

## Property Encryption Object

```
{
  "cipher": string,
  "key"   : string
}
```

### **cipher**

string, optional

The cipher transformation used to encrypt the property. If omitted or null, the default cipher of `"AES/CBC/PKCS5Padding"` is used.

### **key**

string, required

The alias of the key in the IDM cryptography service keystore used to encrypt the property.

## Property Hash Object

```
{
  "algorithm" : string,
  "type"      : string
}
```

### **algorithm**

string, required

The algorithm that should be used to hash the value. For a list of supported hash algorithms, see ["Encoding Attribute Values by Using Salted Hash Algorithms"](#) in the *Security Guide*.

### **type**

string, optional

The type of hashing. Currently only salted hash is supported. If this property is omitted or null, the default `"salted-hash"` is used.

## Custom Managed Objects

Managed objects are inherently fully user definable and customizable. Like all objects, managed objects can maintain relationships to each other in the form of links. Managed objects are intended for use as targets and sources for synchronization operations to represent domain objects, and to build up virtual identities. The name *managed objects* comes from the intention that IDM stores and manages these objects, as opposed to system objects that are present in external systems.

IDM can synchronize and map directly between external systems (system objects), without storing intermediate managed objects. Managed objects are appropriate, however, as a way to cache the data—for example, when mapping to multiple target systems, or when decoupling the availability of systems—to more fully report and audit on all object changes during reconciliation, and to build up views that are different from the original source, such as transformed and combined or virtual views. Managed objects can also be allowed to act as an authoritative source if no other appropriate source is available.

Other object types exist for other settings that should be available to a script, such as configuration or look-up tables that do not need audit logging.

## Setting Up a Managed Object Type

To set up a managed object, you declare the object in your project's `conf/managed.json` file. The following example adds a simple `foobar` object declaration after the user object type.

```
{
  "objects": [
    {
      "name": "user"
    },
    {
      "name": "foobar"
    }
  ]
}
```

## Manipulating Managed Objects Declaratively

By mapping an object to another object, either an external system object or another internal managed object, you automatically tie the object life cycle and property settings to the other object. For more information, see *"Mapping Data Between Resources"* in the *Synchronization Guide*.

## Manipulating Managed Objects Programmatically

You can address managed objects as resources using URLs or URIs with the `managed/` prefix. This works whether you address the managed object internally as a script running in IDM or externally through the REST interface.

You can use all resource API functions in script objects for create, read, update, delete operations, and also for arbitrary queries on the object set, but not currently for arbitrary actions. For more information, see *"Scripting Function Reference"* in the *Scripting Guide*.

IDM supports concurrency through a multi version concurrency control (MVCC) mechanism. Each time an object changes, IDM assigns it a new revision.

Objects can be arbitrarily complex as long as they use supported types, such as maps, lists, numbers, strings, and booleans as defined in JSON.

## Creating Objects

The following script example creates an object type.

```
openidm.create("managed/foobar", "myidentifier", mymap)
```

## Updating Objects

The following script example updates an object type.

```
var expectedRev = origMap._rev
openidm.update("managed/foobar/myidentifier", expectedRev, mymap)
```

The MVCC mechanism requires that `expectedRev` be set to the expected revision of the object to update. You obtain the revision from the object's `_rev` property. If something else changes the object concurrently, IDM rejects the update, and you must either retry or inspect the concurrent modification.

## Patching Objects

You can partially update a managed or system object using the patch method, which changes only the specified properties of the object.

The following script example updates an object type.

```
openidm.patch("managed/foobar/myidentifier", rev, value)
```

The patch method supports a revision of `"null"`, which effectively disables the MVCC mechanism, that is, changes are applied, regardless of revision. In the REST interface, this matches the `If-Match: *` condition supported by patch. Alternatively, you can omit the `"If-Match: *`" header.

For managed objects, the API supports patch by query, so the caller does not need to know the identifier of the object to change.

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '[
{
  "operation": "replace",
  "field": "/password",
  "value": "Passw0rd"
}
]' \
"http://localhost:8080/openidm/managed/user?_action=patch&_queryFilter=username+eq+'DDOE'"
```

## Deleting Objects

The following script example deletes an object type.

```
var expectedRev = origMap._rev
openidm.delete("managed/foobar/myidentifier", expectedRev)
```

The MVCC mechanism requires that `expectedRev` be set to the expected revision of the object to update. You obtain the revision from the object's `_rev` property. If something else changes the object concurrently, IDM rejects deletion, and you must either retry or inspect the concurrent modification.

## Reading Objects

The following script example reads an object type.

```
val = openidm.read("managed/foobar/myidentifier")
```

## Querying Object Sets

You can query managed objects using common query filter syntax. The following script example queries managed user objects whose `userName` is smith.

```
var qry = {
  "_queryFilter" : "/userName eq \"smith\""
};
val = openidm.query("managed/user", qry);
```

For more information, see "Define and Call Data Queries".

## Accessing Managed Objects Through the REST API

IDM exposes all managed object functionality through the REST API unless you configure a policy to prevent such access. In addition to the common REST functionality of create, read, update, delete, patch, and query, the REST API also supports patch by query. For more information, see the [REST API Reference](#).

IDM requires authentication to access the REST API. The authentication configuration is specified in your project's `conf/authentication.json` file. The default authorization filter script is `openidm/bin/defaults/script/router-authz.js`. For more information, see "Secure Authentication" in the *Security Guide*.

## Configuration Objects

IDM provides an extensible configuration to allow you to leverage regular configuration mechanisms.

Unlike native the IDM configuration, which is interpreted automatically and can start new services, IDM stores custom configuration objects and makes them available to your code through the API.

For an introduction to the standard configuration objects, see "Configure the Server" in the *Setup Guide*.

## When To Use Custom Configuration Objects

Configuration objects are ideal for metadata and settings that need not be included in the data to reconcile. Use configuration objects for data that does not require audit log, and does not serve directly as a target or source for mappings.

Although you can set and manipulate configuration objects programmatically and manually, configuration objects are expected to change slowly, through both manual file updates and programmatic updates. To store temporary values that can change frequently and that you do not expect to be updated by configuration file changes, custom repository objects might be more appropriate.

## Custom Configuration Object Naming Conventions

By convention custom configuration objects are added under the reserved context, `config/custom`.

You can choose any name under `config/context`. Be sure, however, to choose a value for `context` that does not clash with future IDM configuration names.

## Mapping Configuration Objects To Configuration Files

If you have not disabled the file based view for configuration, you can view and edit all configuration including custom configuration in `openidm/conf/*.json` files. The configuration maps to a file named `context-config-name.json`, where `context` for custom configuration objects is `custom` by convention, and `config-name` is the configuration object name. A configuration object named `escalation` thus maps to a file named `conf/custom-escalation.json`.

IDM detects and automatically picks up changes to the file.

IDM also applies changes made through APIs to the file.

By default, IDM stores configuration objects in the repository. The file view is an added convenience aimed to help you in the development phase of your project.

## Configuration Objects File and REST Payload Formats

By default, IDM maps configuration objects to JSON representations.

IDM represents objects internally in plain, native types like maps, lists, strings, numbers, booleans, null. The object model is restricted to simple types so that mapping objects to external representations is easy.

The following example shows a representation of a configuration object with a look-up map.

```
{
  "CODE123" : "ALERT",
  "CODE889" : "IGNORE"
}
```

In the JSON representation, maps are represented with braces (`{ }`), and lists are represented with brackets (`[ ]`). Objects can be arbitrarily complex, as in the following example.

```
{
  "CODE123" : {
    "email" : ["sample@sample.com", "john.doe@somedomain.com"],
    "sms" : ["555666777"]
  }
  "CODE889" : "IGNORE"
}
```

## Accessing Configuration Objects Through the REST API

You can list all available configuration objects, including system and custom configurations, using an HTTP GET on `/openidm/config`.

The `_id` property in the configuration object provides the link to the configuration details with an HTTP GET on `/openidm/config/id-value`. By convention, the *id-value* for a custom configuration object called `escalation` is `custom/escalation`.

IDM supports REST mappings for create, read, update, delete, patch, and query of configuration objects.

## Accessing Configuration Objects Programmatically

You can address configuration objects as resources using the URL or URI `config/` prefix both internally and also through the REST interface. The resource API provides script object functions for create, read, update, query, and delete operations.

IDM supports concurrency through a multi version concurrency control mechanism. Each time an object changes, IDM assigns it a new revision.

Objects can be arbitrarily complex as long as they use supported types, such as maps, lists, numbers, strings, and booleans.

## Creating Objects

The following script example creates an object type.

```
openidm.create("config/custom", "myconfig", mymap)
```

## Updating Objects

The following script example updates a custom configuration object type.

```
openidm.update("config/custom/myconfig", mymap)
```

## Deleting Objects

The following script example deletes a custom configuration object type.

```
openidm.delete("config/custom/myconfig")
```

## Reading Objects

The following script example reads an object type.

```
val = openidm.read("config/custom/myconfig")
```

## System Objects

*System objects* are pluggable representations of objects on external systems. They follow the same RESTful resource based design principles as managed objects. There is a default implementation for the ICF framework, which allows any connector object to be represented as a system object.

## Audit Objects

Audit objects contain audit data selected for local storage in repository.

## Links

Link objects define relations between source objects and target objects, usually relations between managed objects and system objects. The link relationship is established by provisioning activity that either results in a new account on a target system, or a reconciliation or synchronization scenario that takes a **LINK** action.

# IDM Glossary

|                    |  |
|--------------------|--|
| correlation query  | A correlation query specifies an expression that matches existing entries in a source repository to one or more entries in a target repository. A correlation query might be built with a script, but it is not the same as a correlation script. For more information, see <i>"Correlating Source Objects With Existing Target Objects"</i> in the <i>Synchronization Guide</i> . |
| correlation script | A correlation script matches existing entries in a source repository, and returns the IDs of one or more matching entries on a target repository. While it skips the intermediate step associated with a <b>correlation query</b> , a correlation script can be relatively complex, based on the operations of the script.   |
| entitlement        | An entitlement is a collection of attributes that can be added to a user entry via roles. As such, it is a specialized type of <b>assignment</b> . A user or device with an entitlement gets access rights to specified resources. An entitlement is a property of a managed object.   |
| JCE                | Java Cryptographic Extension, which is part of the Java Cryptography Architecture, provides a framework for encryption, key generation, and digital signatures.  |
| JSON               | JavaScript Object Notation, a lightweight data interchange format based on a subset of JavaScript syntax. For more information, see the <a href="#">JSON site</a> .  |
| JSON Pointer       | A JSON Pointer defines a string syntax for identifying a specific value within a JSON document. For information about JSON Pointer syntax, see the <a href="#">JSON Pointer RFC</a> .  |

|                 |   |
|-----------------|---|
| JWT             | JSON Web Token. As noted in the <a href="#">JSON Web Token draft IETF Memo</a> , "JSON Web Token (JWT) is a compact URL-safe means of representing claims to be transferred between two parties." For IDM, the JWT is associated with the <a href="#">JWT_SESSION</a> authentication module.                                |
| managed object  | An object that represents the identity-related data managed by IDM. Managed objects are configurable, JSON-based data structures that IDM stores in its pluggable repository. The default configuration of a managed object is that of a user, but you can define any kind of managed object, for example, groups or roles. |
| mapping         | A policy that is defined between a source object and a target object during reconciliation or synchronization. A mapping can also define a trigger for validation, customization, filtering, and transformation of source and target objects.   |
| OSGi            | A module system and service platform for the Java programming language that implements a complete and dynamic component model. For more information, see <a href="#">What is OSGi?</a> Currently, only the <a href="#">Apache Felix</a> container is supported.   |
| reconciliation  | During reconciliation, comparisons are made between managed objects and objects on source or target systems. Reconciliation can result in one or more specified actions, including, but not limited to, synchronization.  |
| resource        | An external system, database, directory server, or other source of identity data to be managed and audited by the identity management system.   |
| REST            | Representational State Transfer. A software architecture style for exposing resources, using the technologies and protocols of the World Wide Web. REST describes how distributed data objects, or resources, can be defined and addressed.   |
| role            | IDM distinguishes between two distinct role types - provisioning roles and authorization roles. For more information, see " <a href="#">Managed Roles</a> ".  |
| source object   | In the context of reconciliation, a source object is a data object on the source system, that IDM scans before attempting to find a corresponding object on the target system. Depending on the defined mapping, IDM then adjusts the object on the target system (target object).  |
| synchronization | The synchronization process creates, updates, or deletes objects on a target system, based on the defined mappings from the source system. Synchronization can be scheduled or on demand.   |
| system object   | A pluggable representation of an object on an external system. For example, a user entry that is stored in an external LDAP directory is  |

represented as a system object in IDM for the period during which IDM requires access to that entry. System objects follow the same RESTful resource-based design principles as managed objects.

#### target object

In the context of reconciliation, a target object is a data object on the target system, that IDM scans after locating its corresponding object on the source system. Depending on the defined mapping, IDM then adjusts the target object to match the corresponding source object.