

Synchronization

Configure synchronization between ForgeRock® Identity Management and other resources.

Synchronizing identity data between resources is one of the core services of ForgeRock Identity Management (IDM). In this guide, you will learn about the different types of synchronization, and how to configure the flexible synchronization mechanism. This guide is written for systems integrators building solutions based on ForgeRock Identity Management services.



Synchronization overview

Understand synchronization types and configuration.



Mappings

Map data between resources.



Situations and actions

Learn about synchronization situations and how to configure actions for each.



Filter synchronization data

Use filtering mechanisms to limit the synchronized data.




Implicit sync and liveSync



Reconciliation performance

[Configure automatic synchronization between resources.](#)

[Learn about ways to improve reconciliation performance.](#)

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.

Synchronization overview

Synchronization keeps data consistent across disparate resources. Within IDM, we refer to two resource types—*_managed resources_* (stored in the IDM repository) and *external resources*. An external resource can be any system that holds identity data, such as ForgeRock Directory Services (DS), Active Directory, a CSV file, a JDBC database, and so on.

IDM connects to external resources through *connectors*. Synchronization across resources happens when managed resources change, or when IDM discovers a change on a system resource. There are various *synchronization mechanisms* that ensure data consistency.

Synchronization types

- IDM discovers and synchronizes changes from external resources by using *reconciliation* and *liveSync*.
- IDM synchronizes changes made to managed resources by using *reconciliation* and *implicit synchronization*.

Reconciliation

Reconciliation is the process of ensuring that the objects in two different data stores are consistent. Traditionally, reconciliation applies mainly to user objects, but IDM can reconcile any objects, such as groups, roles, and devices.

In any reconciliation operation, there is a *source system* (the system that contains the changes) and a *target system* (the system to which the changes will be propagated). The source and target system are defined in a *mapping*. The IDM repository can be either the source or the target in a mapping. You can

configure multiple mappings for one IDM instance, depending on the external resources to which you are connecting.

To perform reconciliation, IDM analyzes both the source system *and* the target system, to discover the differences between them. Reconciliation can therefore be a heavyweight process. When working with large data sets, finding all changes can be more work than processing the changes.

Reconciliation is very thorough. It recognizes system error conditions and catches changes that might be missed by liveSync, and therefore serves as the basis for compliance and reporting.

LiveSync

LiveSync captures the changes that occur on an external system, and pushes those changes to IDM. IDM uses any defined mappings to replay those changes where they are required—to its managed objects, to another remote system, or to both. Unlike reconciliation, liveSync uses a polling system, and is intended to react quickly to changes as they happen.

To perform this polling, liveSync relies on a change detection mechanism on the external resource to determine which objects have changed. The change detection mechanism is specific to the external resource, and can be a time stamp, a sequence number, a change vector, or any other method of recording changes that have occurred on the system. For example, ForgeRock Directory Services (DS) implements a change log that provides IDM with a list of objects that have changed since the last request. Active Directory implements a change sequence number, and certain databases might have a `lastChange` attribute.

Implicit synchronization

Implicit synchronization automatically pushes changes that are made to IDM managed objects out to external systems.

For direct changes to managed objects, IDM immediately synchronizes those changes to all mappings configured to use those objects as their source. A direct change can originate not only as a write request through the REST interface, but also as an update resulting from reconciliation with another resource.

NOTE

Implicit synchronization only synchronizes *changed objects* to external resources. To synchronize a complete data set, you must run a reconciliation operation. The entire changed object is synchronized. If you want to synchronize only the attributes that have changed, you can modify the `onUpdate` script in your mapping to compare attribute values before pushing changes.

Synchronization configuration overview

This section describes the high-level steps required to set up synchronization between two resources. A basic synchronization configuration involves the following steps:

1. Set up a connection between the source and target resource.

A [connector configuration](#) references a specific connector type and indicates the connection details of the external resource. You must define a connector configuration for each external resource to which you are connecting.

For more information, see [Connections between resources](#).

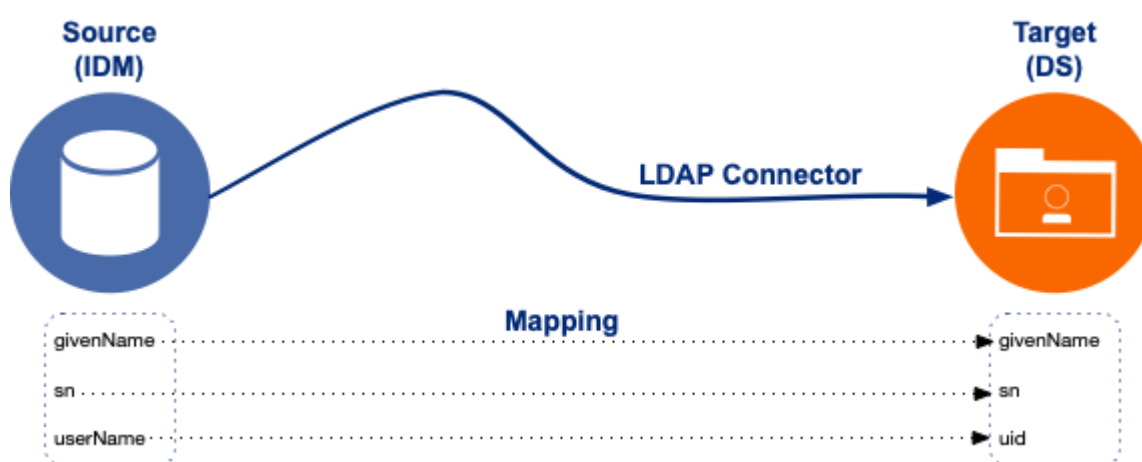
2. Map source objects to target objects.

The [mapping configuration](#) your project's `conf/sync.json` file or in individual mapping files. Mappings are synchronized in the order in which they are specified in the `sync.json` file. If there are multiple mapping files, the `syncAfter` property dictates the order in which they are processed.

For more information, see [Resource mapping](#).

3. Configure any scripts that are required to check source and target objects, and to manipulate attributes.
4. In addition to these configuration elements, IDM stores a `links` table in its repository. The links table maintains a record of relationships established between source and target objects.

The following diagram illustrates the high-level synchronization configuration:



Data mapping model

IDM uses mappings to determine which data to synchronize, and how that data must be synchronized.

In general, identity management software implements one of the following data models:

- A meta-directory data model, where all data are mirrored in a central repository.

The meta-directory model offers fast access at the risk of getting outdated data.

- A virtual data model, where only a minimum set of attributes are stored centrally, and most are loaded on demand from the external resources in which they are stored.

The virtual model guarantees fresh data, but pays for that guarantee in terms of performance.

IDM leaves the data model choice up to you. You determine the right trade-offs for a particular deployment. IDM does not hard code any particular schema or set of attributes stored in the repository. Instead, you define how external system objects map onto managed objects, and IDM dynamically updates the repository to store the managed object attributes that you configure.

Connections between resources

A connector lets you transfer data between different resource systems. The connector configuration works in conjunction with the [synchronization mapping](#) and specifies how target object attributes map to attributes on external objects.

[Connector configuration](#) files exist in your project's `conf` directory, and are named `provisioner.resource-name.json`; where **resource-name** reflects the connector technology and the external resource. For example, `openicf-csv`.

To create and modify connector configurations, use one of the following methods:

Configure connectors using the admin UI

From the navigation bar, click **Configure > Connectors**, and do one of the following.

- Select an existing connector to modify.
- Click **New Connector**, and configure the new connector.

Edit connector configuration files

IDM provides a number of sample provisioner files in the `path/to/openidm/samples/example-configurations/provisioners` directory. To modify connector configuration files directly, edit one of the sample provisioner files that corresponds to the resource to which you are connecting.

The following excerpt of an example LDAP connector configuration shows the attributes of an account object type. In the attribute mapping definitions, the attribute name is mapped from the IDM managed object to the `nativeName` (the attribute name used on the external resource). The `lastName` attribute in IDM is mapped to the `sn` attribute in LDAP. The `homePhone` attribute is defined as an array, because it can have multiple values:

```
{
  ...
  "objectTypes": {
    "account": {
      "lastName": {
        "type": "string",
        "required": true,
        "nativeName": "sn",
        "nativeType": "string"
      },
      "homePhone": {
        "type": "array",
        "items": {
          "type": "string",
          "nativeType": "string"
        },
        "nativeName": "homePhone",
        "nativeType": "string"
      }
    }
  }
}
```

For IDM to access external resource objects and attributes, the object and its attributes must match the connector configuration. Note that the connector file only maps IDM managed objects and attributes to their counterparts on the external resource. To construct attributes and to manipulate their values, you use a synchronization mapping, described in [Resource mapping](#).

Configure connectors using REST

Create connector configurations using REST with the `createCoreConfig` and `createFullConfig` actions. For more information, see [Configure Connectors Using REST](#).

Resource mapping

A synchronization mapping specifies a relationship between objects and their attributes in two data stores. The following example shows a typical attribute mapping, between objects in an external LDAP directory and an IDM managed user data store:

```
"source": "lastName",  
"target": "sn"
```

In this case, the `lastName` source attribute is mapped to the `sn` (surname) attribute in the target LDAP directory.

The core synchronization configuration is defined in the [mapping configuration](#).

You can define a single file with all your mappings (`conf/sync.json`), or a separate file per mapping. Individual mapping files are named `mapping-mappingName.json`; for example, `mapping-managedUser_systemCsvfileAccounts.json`. Individual mapping files can be useful if your deployment includes many mappings that are difficult to manage in a single file. You can also use a combination of individual mapping files and a monolithic `sync.json` file, particularly if you are adding mappings to an existing deployment.

If you use a single `sync.json` file, mappings are processed in the order in which they appear within that file. If you use multiple mapping files, mappings are processed according to the `syncAfter` property in the mapping. The following example indicates that this particular mapping must be processed after the `managedUser_systemCsvfileAccount` mapping:

```
"source" : "managed/user",  
"target" : "system/csvfile/account",  
"syncAfter" : [ "managedUser_systemCsvfileAccount" ],
```

If you use a combination of `sync.json` and individual mapping files, the synchronization engine processes the mappings in `sync.json` first (in order), and then any mappings specified in the individual mapping files, according to the `syncAfter` property in each mapping.

For a list of *all* mappings, use the following request:

```
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/sync/mappings?_queryFilter=true"
```

This call returns the mappings in the order in which they will be processed.

NOTE

NOTE

The admin UI only shows the mappings configured in the `sync.json` file. Do not use the admin UI to add or change mappings in individual mapping files.

Mappings are always defined from a *source* resource to a *target* resource. To configure bidirectional synchronization, you must define two mappings. For example, to configure bidirectional synchronization between an LDAP server and an IDM repository, you would define the following two mappings:

- LDAP Server > IDM Repository
- IDM Repository > LDAP Server

Bidirectional mappings can include a `links` property that lets you reuse the links established between objects, for both mappings. For more information, see [Reuse Links Between Mappings](#).

You can update a mapping while the server is running. To avoid inconsistencies between data stores, do not update a mapping while a reconciliation is in progress *for that mapping*.

Configure a resource mapping

Objects in external resources are specified in a mapping as `system/name/object-type`, where *name* is the name used in the [connector configuration](#), and *object-type* is the object defined in the [connector configuration](#) list of object types. Objects in the repository are specified in the mapping as `managed/object-type`, where *object-type* is defined in the [managed object configuration](#).

External resources, and IDM managed objects, can be the *source* or the *target* in a mapping. By convention, the mapping name is a string of the form `source_target`, as shown in the following example:

▼ [Basic LDAP mapping](#)

```
{
  "mappings": [
    {
      "name": "systemLdapAccounts_managedUser",
      "source": "system/ldap/account",
      "target": "managed/user",
      "properties": [
        {
          "source": "lastName",
          "target": "sn"
        }
      ]
    }
  ]
}
```

```

    {
      "source": "telephoneNumber",
      "target": "telephoneNumber"
    },
    {
      "target": "phoneExtension",
      "default": "0047"
    },
    {
      "source": "email",
      "target": "mail",
      "comment": "Set mail if non-empty.",
      "condition": {
        "type": "text/javascript",
        "source": "(object.email != null)"
      }
    },
    {
      "source": "",
      "target": "displayName",
      "transform": {
        "type": "text/javascript",
        "source": "source.lastName +', ' +
source.firstName;"
      }
    },
    {
      "source": "uid",
      "target": "userName",
      "condition": "/linkQualifier eq \"user\""
    }
  ],
}
]
}

```

In this example, the **name** of the source is the external resource (ldap), and the target is IDM's user repository; specifically, managed/user . The properties defined in the mapping correspond to attribute names that are defined in the IDM configuration. For example, the source attribute uid is defined in the ldap connector configuration file, rather than on the external resource itself.

Individual mapping files do not include a name property. The mapping name is taken from the file name. For example, the mapping shown in [Basic LDAP Mapping](#) would be in

a file named `mapping-systemLdapAccounts_managedUser.json`, and start as follows:

```
{
  "source": "system/ldap/account",
  "target": "managed/user",
  ...
}
```

Configure mappings using the admin UI

To set up a synchronization mapping using the admin UI:

1. From the navigation bar, click **Configure > Mappings**.
2. Click **New Mapping**.
3. On the **New Mapping** page, select a source and target resource from the configured resources at the bottom of the window, and click **Create Mapping**.

You can filter these resources to display only connector configurations or managed objects.

4. Select **Add property** on the **Attributes** grid to map a target property to its corresponding source property.

The **Property** list shows all configured properties on the target resource. If the target resource is specified in a connector configuration, the **Property** list shows all properties configured for this connector. If the target resource is a managed object, the **Property** list shows the list of properties (defined in the [managed object configuration](#) for that object).

TIP

- Select **Add Missing Required Properties** to add all the properties that are configured as *required* on the target resource. You can then map these required properties individually.
- Select **Quick Mapping** to show all source and target properties simultaneously. Drag a source property onto its corresponding target property, or vice versa. When you're done, click **Save**.

5. To test your mapping configuration on a single source entry, click the **Behaviors** tab and scroll down to **Single Record Reconciliation**. Search for the entry to reconcile.

The UI displays a preview of the target entry after a reconciliation. You can then click **Reconcile Selected Record** to perform the reconciliation on that one source entry.

Remove a mapping

- To remove a mapping, delete the corresponding section from your [mapping configuration](#). If you have configured mappings in individual mapping files, delete the file associated with the mapping you want to remove.
- To remove a mapping using the admin UI, select **Configure > Mappings**, and then click **Delete** under the mapping to remove.

NOTE

If you delete a mapping using the admin UI, the `delete-mapping-links` script removes all links related to that mapping from the repository. If you delete the mapping directly in the configuration file, no links are deleted from the repository.

Transform attributes using a mapping

You can use a mapping to define attribute transformations during synchronization. In the following sample mapping excerpt, the value of the `displayName` attribute on the target is set using a combination of the `lastName` and `firstName` attribute values from the source:

```
{
  "source": "",
  "target": "displayName",
  "transform": {
    "type": "text/javascript",
    "source": "source.lastName + ', ' + source.firstName;"
  }
},
```

For transformations, the `source` property is optional. However, a source object is only available if you specify the `source` property. Therefore, in order to use `source.lastName` and `source.firstName` to calculate the `displayName`, the example specifies `"source" : ""`.

If you set `"source" : ""` (not specifying an attribute), the entire object is regarded as the source, and you must include the attribute name in the transformation script. For example, to transform the source `username` to lowercase, your script would be `source.mail.toLowerCase();`. If you do specify a source attribute (for example, `"source" : "mail"`), just that attribute is regarded as the source. In this case, the transformation script would be `source.toLowerCase();`.

Configure attribute transformation using the admin UI

1. From the navigation bar, click **Configure > Mappings**, and select a mapping.
2. Select the line with the target attribute value to set.
3. On the **Transformation Script** tab, select **Javascript** or **Groovy**, and enter the transformation as an **Inline Script**, or specify the path to the file containing your transformation script.

When you use the UI to map a property with an encrypted value, you are prompted to set up a transformation script to decrypt the value when that property is synchronized. The resulting mapping looks similar to the following, which shows the transformation of a user's password property:

```
{
  "target" : "userPassword",
  "source" : "password",
  "transform" : {
    "type" : "text/javascript",
    "globals" : { },
    "source" : "openidm.decrypt(source);"
  },
  "condition" : {
    "type" : "text/javascript",
    "globals" : { },
    "source" : "object.password != null"
  }
}
```

Default attribute values in a mapping

You can use a mapping to *create* attributes on the target resource. The following mapping excerpt creates a `phoneExtension` attribute with a default value of `0047` on the target object:

```
{
  "target": "phoneExtension",
  "default": "0047"
},
```

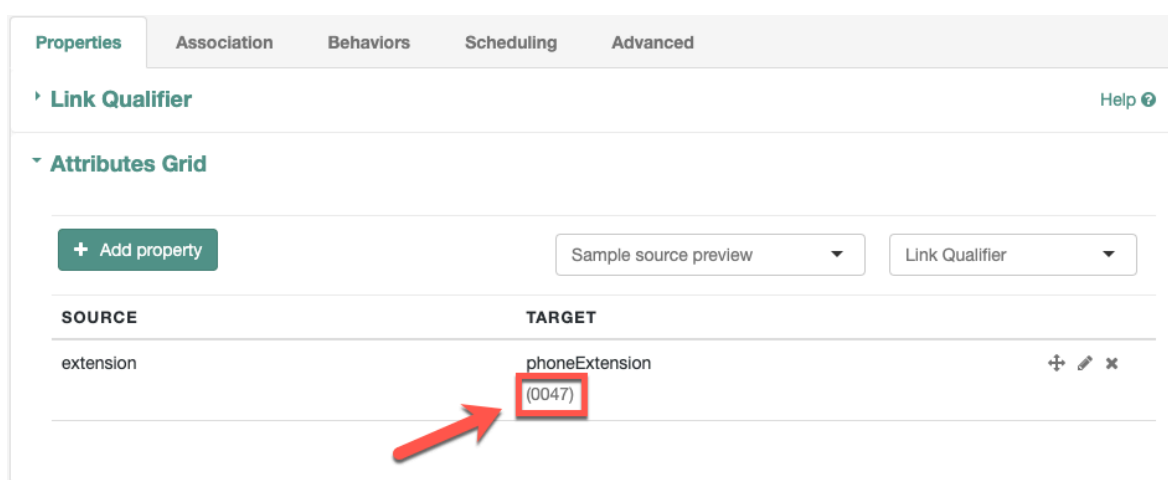
The `default` property specifies a value to assign to the attribute on the target object. Before IDM determines the value of the target attribute, it evaluates any applicable

conditions, followed by any transformation scripts. If the `source` property and the `transform` script yield a null value, IDM applies the default value in the create and update actions. The default value overrides the target value, if one exists.

Configure default attribute values using the admin UI

1. From the navigation bar, click **Configure > Mappings**, and click the mapping to edit.
2. Click the **Properties** tab.
3. Expand the **Attributes Grid** node, and click the **Target** property to edit.
4. In the **Target Property: name** window, click the **Default Values** tab, and add or edit the default values.
5. Click **Save**.

The default value displays in the **Attributes Grid**.



Scriptable conditions in a mapping

By default, IDM synchronizes all attributes in a mapping. For more complex relationships between source and target objects, you can define conditions under which IDM maps certain attributes. You can define two types of mapping conditions:

- *Scriptable conditions*, in which an attribute is mapped only if the defined script evaluates to `true`.
- *Condition filters*, a declarative filter that sets the conditions under which the attribute is mapped. Condition filters can include a *link qualifier*, that identifies the *type* of relationship between the source object and multiple target objects. For more information, see [Map a Single Source Object to Multiple Target Objects](#).

The following list shows examples of condition filters:

- `"condition": "/object/country eq 'France' "` —Only map the attribute if the object's `country` attribute equals `France`.

- "condition": "/object/password pr" —Only map the attribute if the object's password attribute is present.
- "condition": "/linkQualifier eq 'admin'" —Only map the attribute if the link between this source and target object is of type admin .

Configure mapping conditions using the admin UI

1. From the navigation bar, click **Configure > Mappings**, and click the mapping to edit.
2. Click the **Properties** tab.
3. Expand the **Attributes Grid** node, click the property to edit, click the **Conditional Updates** tab, and then do one of the following:
 - To configure a filtered condition, click **Condition Filter**.
 - To configure a scriptable condition, click **Script**.
4. Click **Save**.

Scriptable conditions create mapping logic, based on the result of the condition script. If the script does not return `true`, IDM does not manipulate the target attribute during a synchronization operation.

In the following excerpt, the value of the target `mail` attribute is set to the value of the source `email` attribute *only if* the source attribute is not empty:

```
{
  "target": "mail",
  "comment": "Set mail if non-empty.",
  "source": "email",
  "condition": {
    "type": "text/javascript",
    "source": "(object.email != null)"
  }
  ...
}
```

TIP

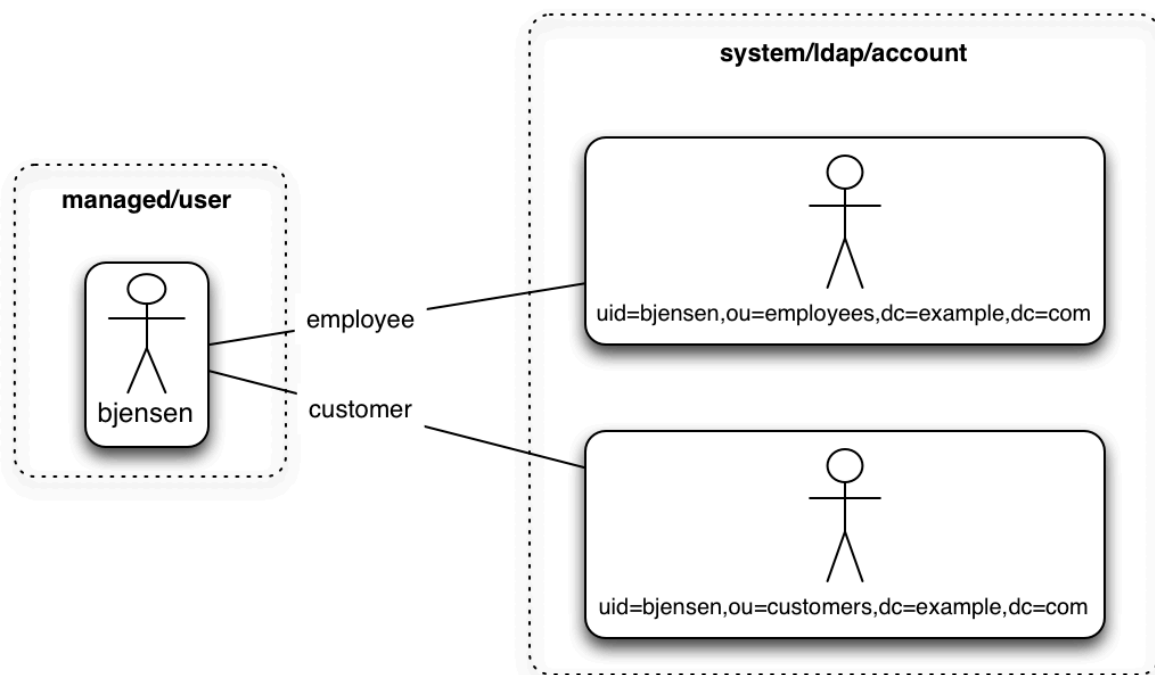
You can add comments to JSON files. This example includes a property named `comment`; however, you can use any unique property name, as long as it is not used elsewhere in the server. IDM ignores unknown property names in JSON configuration files.

Map a single source object to multiple target objects

In certain cases, you might have a single object in a resource that maps to more than one object in another resource. For example, assume that managed user, bjensen, has two distinct accounts in an LDAP directory: an `employee` account (under `uid=bjensen,ou=employees,dc=example,dc=com`) and a `customer` account (under `uid=bjensen,ou=customers,dc=example,dc=com`). You want to map both of these LDAP accounts to the same managed user account.

IDM uses *link qualifiers* to manage this one-to-many scenario. A link qualifier is essentially a label that identifies the *type* of link (or relationship) between objects.

The following diagram shows two link qualifiers that let you link both of bjensen's LDAP accounts to her managed user object:



NOTE

The previous diagram displays that the link qualifier is a property of the *link* between the source and target object, and not a property of the source or target object itself.

Link qualifiers are defined as part of the mapping. Each link qualifier must be unique within the mapping. If no link qualifier is specified (when only one possible matching target object exists), IDM uses a default link qualifier with the value `default`.

Link qualifiers can be defined as a static list, or dynamically, using a script. The following excerpt of a sample mapping shows the two static link qualifiers, `employee` and `customer`, described at the top of this topic:

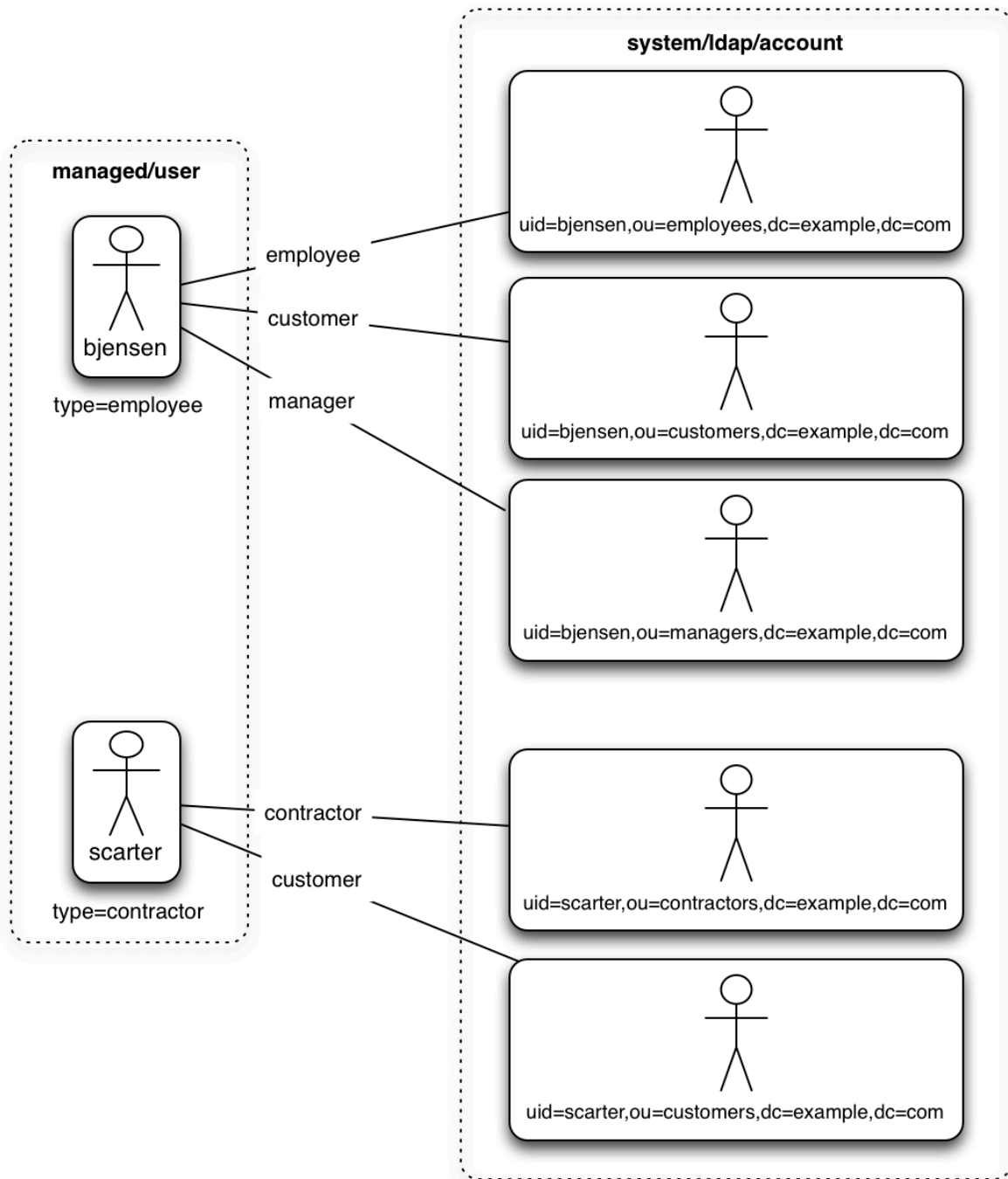
```
{  
  "mappings": [  

```

```
{
  "name": "managedUser_systemLdapAccounts",
  "source": "managed/user",
  "target": "system/MyLDAP/account",
  "linkQualifiers" : [ "employee", "customer" ],
  ...
}
```

IDM evaluates the list of static link qualifiers for *every* source record. That is, every reconciliation processes all synchronization operations, for each link qualifier, in turn.

A dynamic link qualifier script returns a list of link qualifiers that can be applied to each source record. For example, suppose you have two *types* of managed users—employees and contractors. For employees, a single managed user (source) account can correlate with three different LDAP (target) accounts—employee, customer, and manager. For contractors, a single managed user account can correlate with only two separate LDAP accounts—contractor, and customer. The following diagram displays the possible linking situations for this scenario:



In this scenario, you could write a script to generate a dynamic list of link qualifiers, based on the managed user type. For employees, the script would return [employee, customer, manager] in its list of possible link qualifiers. For contractors, the script would return [contractor, customer] in its list of possible link qualifiers. A reconciliation operation would then process only the list of link qualifiers applicable to each source object.

If your source resource includes many records, you should use a dynamic link qualifier script instead of a static list of link qualifiers. Generating the list of applicable link qualifiers dynamically avoids unnecessary additional processing for those qualifiers that will never apply to specific source records. Therefore, synchronization performance is improved for large source data sets.

You can include a dynamic link qualifier script inline (using the `source` property), or by referencing a JavaScript or Groovy script file (using the `file` property). The following link qualifier script sets up the dynamic link qualifier lists described in the previous example.

NOTE

In this example, the `source` property value has been formatted across multiple lines for clarity. In general, the script source must be formatted on a single line.

```
{
  "mappings": [
    {
      "name": "managedUser_systemLdapAccounts",
      "source": "managed/user",
      "target": "system/MyLDAP/account",
      "linkQualifiers" : {
        "type" : "text/javascript",
        "globals" : { },
        "source" : "if (returnAll) {
                    ['contractor', 'employee', 'customer',
'manager' ]
                } else {
                    if(object.type === 'employee') {
                        ['employee', 'customer', 'manager']
                    } else {
                        ['contractor', 'customer']
                    }
                }
            }"
    }
  ]
  ...
}
```

To reference an external link qualifier script, provide a link to the file in the `file` property:

```
{
  "mappings": [
    {
      "name": "managedUser_systemLdapAccounts",
      "source": "managed/user",
      "target": "system/MyLDAP/account",
      "linkQualifiers" : {
        "type" : "text/javascript",
        "file" : "script/linkQualifiers.js"
      }
    }
  ]
}
```

```
...
}
```

Dynamic link qualifier scripts must return all valid link qualifiers when the `returnAll` global variable is true. The `returnAll` variable is used during the target reconciliation phase to check whether there are any target records that are unassigned, for each known link qualifier.

If you configure dynamic link qualifiers through the UI, the complete list of dynamic link qualifiers displays in the Generated Link Qualifiers item below the script. This list represents the values returned by the script when the `returnAll` variable is passed as `true`. For a list of the variables available to a dynamic link qualifier script, see [Script Triggers Defined in Mappings](#).

Link qualifiers have no functionality on their own, but they can be referenced in reconciliation operations to manage situations where a single source object maps to multiple target objects. The following examples show how link qualifiers can be used in reconciliation operations:

- Use link qualifiers during object creation, to create multiple target objects per source object.

The following mapping excerpt defines a transformation script that generates the value of the `dn` attribute on an LDAP system. If the link qualifier is `employee`, the value of the target `dn` is set to

`"uid=userName,ou=employees,dc=example,dc=com"`. If the link qualifier is `customer`, the value of the target `dn` is set to

`"uid=userName,ou=customers,dc=example,dc=com"`. The reconciliation operation iterates through the link qualifiers for each source record. In this case, two LDAP objects, with different `dn`s are created for each managed user object:

```
{
  "target" : "dn",
  "transform" : {
    "type" : "text/javascript",
    "globals" : { },
    "source" : "if (linkQualifier === 'employee')
      { 'uid=' + source.userName +
',ou=employees,dc=example,dc=com' ; }
      else
      if (linkQualifier === 'customer')
      { 'uid=' + source.userName +
',ou=customers,dc=example,dc=com' ; }"
  },
}
```

```
    "source" : ""
  }
```

- Use link qualifiers with *correlation queries*. The correlation query assigns a link qualifier based on the values of an existing target object.

During source synchronization, IDM queries the target system for every source record *and* link qualifier, to check if there are any matching target records. If a match is found, the sourceId, targetId, and linkQualifier are all saved as the *link*.

The following excerpt of a sample mapping shows the two link qualifiers described previously (*employee* and *customer*). The correlation query first searches the target system for the *employee* link qualifier. If a target object matches the query, based on the value of its *dn* attribute, IDM creates a link between the source object and that target object, and assigns the *employee* link qualifier to that link. This process is repeated for all source records. Then, the correlation query searches the target system for the *customer* link qualifier. If a target object matches that query, IDM creates a link between the source object and that target object and assigns the *customer* link qualifier to that link:

```
"linkQualifiers" : ["employee", "customer"],
  "correlationQuery" : [
    {
      "linkQualifier" : "employee",
      "type" : "text/javascript",
      "source" : "var query = {'_queryFilter': 'dn co \'' +
uid=source.userName + 'ou=employees\''}; query;"
    },
    {
      "linkQualifier" : "customer",
      "type" : "text/javascript",
      "source" : "var query = {'_queryFilter': 'dn co \'' +
uid=source.userName + 'ou=customers\''}; query;"
    }
  ]
  ...
```

For more information about correlation queries, see [Writing Correlation Queries](#).

- Use link qualifiers during policy validation to apply different policies based on the link type.

The following excerpt of a sample mapping shows two link qualifiers, *user* and *test* . Depending on the link qualifier, different actions are taken when the target record is ABSENT:


```

{
  "mappings" : [
    {
      "name" : "systemLdapAccounts_managedUser",
      "source" : "system/ldap/account",
      "target" : "managed/user",
      "linkQualifiers" : [
        "user",
        "test"
      ],
    },
    "properties" : [
      ...
      "policies" : [
        {
          "situation" : "CONFIRMED",
          "action" : "IGNORE"
        },
        {
          "situation" : "FOUND",
          "action" : "UPDATE"
        }
      ],
      {
        "condition" : "/linkQualifier eq \"user\"",
        "situation" : "ABSENT",
        "action" : "CREATE",
        "postAction" : {
          "type" : "text/javascript",
          "source" :
"java.lang.System.out.println('Created user: ');"
        }
      },
      {
        "condition" : "/linkQualifier eq \"test\"",
        "situation" : "ABSENT",
        "action" : "IGNORE",
        "postAction" : {
          "type" : "text/javascript",
          "source" :
"java.lang.System.out.println('Ignored user: ');"
        }
      },
      ...
    ]
  ]
}

```

With this sample mapping, the synchronization operation creates an object in the target system only if the potential match is assigned a `user` link qualifier. If the match is assigned a `test` qualifier, no target object is created. In this way, the process avoids creating duplicate *test-related* accounts in the target system.

Configure link qualifiers using the admin UI

1. From the navigation bar, click **Configure > Mappings**, and click the mapping to edit.
2. Click the **Properties** tab, and expand the **Link Qualifier** node.
3. Select **Static** or **Dynamic**, configure the link qualifier, and click **Save**.

TIP

For an example that uses link qualifiers in conjunction with roles, see [Link Multiple Accounts to a Single Identity](#).

Prevent the accidental deletion of a target system

If a source resource is empty, the default behavior is to exit without failure and to log a warning similar to the following:

```
[318] Feb 19, 2020 1:51:56.455 PM
org.forgerock.openidm.sync.NonClusteredRecon dispatchRecon
WARNING: Cannot reconcile from an empty data source, unless
allowEmptySourceSet is true.
```

The reconciliation summary is also logged in the reconciliation audit log.

This behavior prevents reconciliation operations from accidentally deleting everything in a target resource. In the event that a source system is unavailable but erroneously reports its status as up, the absence of source objects should not result in objects being removed on the target resource.

If you *do* want reconciliations of an empty source resource to proceed, override the default behavior by setting the `allowEmptySourceSet` property to `true` in the mapping. For example:

```
{
  "mappings" : [
    {
      "name" : "systemCsvfileAccounts_managedUser",
      "source" : "system/csvfile/account",
```

```
"allowEmptySourceSet" : true,  
...
```

When an empty source is reconciled, the data in the target is wiped out.

Prevent accidental target deletion using the admin UI

1. From the navigation bar, click **Configure > Mappings**, and click the mapping to edit.
2. Click the **Advanced** tab, and expand the **Additional Mapping Options** node.
3. Enable **Allow Reconciliations From an Empty Source**.

Scripts in mappings

You can use a number of *script hooks* to manipulate objects and attributes during synchronization. Scripts can be triggered during various stages of the synchronization process, and are defined as part of the mapping.

You can trigger a script when a managed or system object is created (`onCreate`), updated (`onUpdate`), or deleted (`onDelete`). You can also trigger a script when a link is created (`onLink`) or removed (`onUnlink`).

In the default synchronization mapping, changes are always written to *target* objects, not to *source* objects. However, you can explicitly include a call to an action that should be taken on the source object within the script.

Construct and manipulate attributes

The most common use of synchronization scripts is when a target object is created or updated.

The `onUpdate` script is *always* called for an UPDATE situation, even if the synchronization process determines that there is no difference between the source and target objects, and that the target object will not be updated.

If the `onUpdate` script has run and the synchronization process then determines that the target value to set is the same as its existing value, the change is prevented from synchronizing to the target.

The following excerpt of a sample mapping derives a DN for an LDAP entry when the corresponding managed entry is created:

```
{  
  "onCreate": {
```

```

    "type": "text/javascript",
    "source":
        "target.dn = 'uid=' + source.uid +
        ',ou=people,dc=example,dc=com'"
    }
}

```

Perform other actions

[Construct and Manipulate Attributes With Scripts](#) shows how to manipulate attributes with scripts when objects are created and updated. You can also trigger scripts in response to other synchronization actions. For example, you might not want to delete a managed user directly when an external account is deleted, but instead unlink the objects and deactivate the user in another resource. Alternatively, you might delete the object in IDM and run a script to perform some subsequent action.

The following example shows a more advanced mapping configuration that exposes the script hooks available during synchronization:

```

{
  "mappings": [
    {
      "name": "systemLdapAccount_managedUser",
      "source": "system/ldap/account",
      "target": "managed/user",
      "validSource": {
        "type": "text/javascript",
        "file": "script/isValid.js"
      },
      "correlationQuery" : {
        "type" : "text/javascript",
        "source" : "var map = {'_queryFilter': 'uid eq \"'
+
        "        source.userName + '\"'}; map;"
      },
      "properties": [
        {
          "source": "uid",
          "transform": {
            "type": "text/javascript",
            "source": "source.toLowerCase()"
          },
          "target": "userName"
        },
      ],
    },
  ],
}

```

```

    {
      "source": "",
      "transform": {
        "type": "text/javascript",
        "source": "if (source.myGivenName)
          {source.myGivenName;} else
{source.givenName;}"
      },
      "target": "givenName"
    },
    {
      "source": "",
      "transform": {
        "type": "text/javascript",
        "source": "if (source.mySn)
          {source.mySn;} else {source.sn;}"
      },
      "target": "familyName"
    },
    {
      "source": "cn",
      "target": "fullname"
    },
    {
      "condition": {
        "type": "text/javascript",
        "source": "var clearObj =
openidm.decrypt(object);
          ((clearObj.password != null) &&
          (clearObj.ldapPassword !=
clearObj.password))"
      },
      "transform": {
        "type": "text/javascript",
        "source": "source.password"
      },
      "target": "__PASSWORD__"
    }
  ],
  "onCreate": {
    "type": "text/javascript",
    "source": "target.ldapPassword = null;
target.adPassword = null;
target.password = null;
target.ldapStatus = 'New Account'"
  }
}

```

```

    },
    "onUpdate": {
      "type": "text/javascript",
      "source": "target.ldapStatus = 'OLD'"
    },
    "onUnlink": {
      "type": "text/javascript",
      "file": "script/triggerAdDisable.js"
    },
    "policies": [
      {
        "situation": "CONFIRMED",
        "action": "UPDATE"
      },
      {
        "situation": "FOUND",
        "action": "UPDATE"
      },
      {
        "situation": "ABSENT",
        "action": "CREATE"
      },
      {
        "situation": "AMBIGUOUS",
        "action": "EXCEPTION"
      },
      {
        "situation": "MISSING",
        "action": "EXCEPTION"
      },
      {
        "situation": "UNQUALIFIED",
        "action": "UNLINK"
      },
      {
        "situation": "UNASSIGNED",
        "action": "EXCEPTION"
      }
    ]
  }
]
}

```

The following list shows the properties that you can use as hooks in mapping configurations to call scripts:

Triggered by Situation

onCreate, onUpdate, onDelete, onLink, onUnlink

Object Filter

validSource, validTarget

Correlating Objects

correlationQuery

Triggered on Reconciliation

result

Scripts Inside Properties

condition, transform

Scripts can obtain data from any connected system by using the `openidm.read(id)` function, where `id` is the identifier of the object to read.

The following example reads a managed user object from the repository:

```
repoUser = openidm.read("managed/user/9dce06d4-2fc1-4830-a92b-  
bd35c2f6bcbb");
```

The following example reads an account from an external LDAP resource:

```
externalAccount =  
openidm.read("system/ldap/account/uid=bjensen,ou=People,dc=example  
,dc=com");
```

IMPORTANT

For illustration purposes, this query targets a DN rather than a UID as it did in the previous example. The attribute that is used for the `_id` is defined in the connector configuration and, in this example, is set to `"uidAttribute" : "dn"`. Although you *can* use a DN (or any unique attribute) for the `_id`, it is a best practice to use an attribute that is both unique and immutable, such as the `entryUUID`.

Generate log messages

IDM provides a `logger` object that you can use from scripts defined in your mapping. These scripts can log messages to the OSGi console and to log files. The `logger` object includes the following functions:

- `debug()`
- `error()`

- info()
- trace()
- warn()

Consider the following mapping excerpt:

```
{
  "mappings" : [
    {
      "name" : "systemCsvfileAccounts_managedUser",
      "source" : "system/csvfile/account",
      "target" : "managed/user",
      "correlationQuery" : {
        "type" : "text/javascript",
        "source" : "var query = {'_queryId' : 'for-
userName', 'uid' : source.name};query;"
      },
      "onCreate" : {
        "type" : "text/javascript",
        "source" : "logger.warn('Case onCreate: the source
object contains: = {} ', source); source;"
      },
      "onUpdate" : {
        "type" : "text/javascript",
        "source" : "logger.warn('Case onUpdate: the source
object contains: = {} ', source); source;"
      },
      "result" : {
        "type" : "text/javascript",
        "source" : "logger.warn('Case result: the source
object contains: = {} ', source); source;"
      },
      "properties" : [
        {
          "transform" : {
            "type" : "text/javascript",
            "source" : "logger.warn('Case no Source:
the source object contains: = {} ', source); source;"
          },
          "target" : "sourceTest1Nosource"
        },
        {
          "source" : "",
          "transform" : {
            "type" : "text/javascript",
```



```

        "source" : "logger.warn('Case emptySource:
the source object contains: = {} ', source); source;"
    },
    "target" : "sourceTestEmptySource"
},
{
    "source" : "description",
    "transform" : {
        "type" : "text/javascript",
        "source" : "logger.warn('Case
sourceDescription: the source object contains: = {} ', source);
source"
    },
    "target" : "sourceTestDescription"
},
...
]
}
]
}

```

The scripts that are defined for `onCreate`, `onUpdate`, and `result` log a warning message to the console whenever an object is created or updated, or when a result is returned. The script result includes the full source object.

The scripts that are defined in the `properties` section of the mapping log a warning message if the property in the source object is missing or empty. The last script logs a warning message that includes the description of the source object.

During a reconciliation operation, these scripts would generate output in the OSGi console, similar to the following:

```

2017-02... WARN Case no Source: the source object contains: = null
[9A00348661C6790E7881A7170F747F...]
2017-02... WARN Case emptySource: the source object contains: =
{roles=openidm-..., lastname=Jensen...}
2017-02... WARN Case no Source: the source object contains: = null
[9A00348661C6790E7881A7170F747F...]
2017-02... WARN Case emptySource: the source object contains: =
{roles=openidm..., lastname=Carter,...}
2017-02... WARN Case sourceDescription: the source object
contains: = null [EEE2FF4BCE9748927A1832...]
2017-02... WARN Case sourceDescription: the source object
contains: = null [EEE2FF4BCE9748927A1832...]
2017-02... WARN Case onCreate: the source object contains: =

```

```
{roles=openidm-..., lastname=Carter, ...]
2017-02... WARN Case onCreate: the source object contains: =
{roles=openidm-..., lastname=Jensen, ...]
2017-02... WARN Case result: the source object contains: =
{SOURCE_IGNORED={count=0, ids=[]}, FOUND_ALL...]
```

You can use similar scripts to inject logging into any aspect of a mapping. You can also call the `logger` functions from any configuration file that has scripts hooks. For more information about the `logger` functions, see [Log Functions](#).

Reuse links between mappings

When two mappings synchronize the same objects bidirectionally, use the `links` property in one mapping to have IDM use the same link for both mappings. If you do not specify a `links` property, IDM maintains a separate link for each mapping.

The following excerpt shows two mappings, one from MyLDAP accounts to managed users, and another from managed users to MyLDAP accounts. In the second mapping, the `link` property indicates that IDM should reuse the links created in the first mapping, rather than create new links:

```
{
  "mappings": [
    {
      "name": "systemMyLDAPAccounts_managedUser",
      "source": "system/MyLDAP/account",
      "target": "managed/user"
    },
    {
      "name": "managedUser_systemMyLDAPAccounts",
      "source": "managed/user",
      "target": "system/MyLDAP/account",
      "links": "systemMyLDAPAccounts_managedUser"
    }
  ]
}
```

Reconcile with case-insensitive data stores

IDM is case-sensitive, which means that an uppercase ID is considered different from an otherwise identical lowercase ID during reconciliation. Some data stores, such as ForgeRock Directory Services (DS), are case-insensitive. This can be problematic during

reconciliation, because the ID of the links created by reconciliation might not match the case of the IDs expected by IDM.

If a mapping inherits links by using the `links` property, you do not need to worry about case-sensitivity, because the mapping uses the setting of the referred links.

Alternatively, you can address case-sensitivity issues with target systems in the following ways:

- Specify a case-insensitive data store. To do so, set the `sourceIdsCaseSensitive` or `targetIdsCaseSensitive` properties to `false` in the mapping for those links. For example, if the source LDAP data store is case-insensitive, set the mapping from the LDAP store to the managed user repository as follows:

```
"mappings" : [  
  {  
    "name" : "systemLdapAccounts_managedUser",  
    "source" : "system/ldap/account",  
    "sourceIdsCaseSensitive" : false,  
    "target" : "managed/user",  
    "properties" : [  
      ...  
    ]  
  }  
]
```

You might also need to modify the [connector configuration](#), setting the `enableFilteredResultsHandler` property to `false`:

```
"resultsHandlerConfig" :  
{  
  "enableFilteredResultsHandler": false  
},
```

CAUTION

Do not disable the filtered results handler for the CSV file connector. The CSV file connector does not perform filtering. Therefore, if you disable the filtered results handler for this connector, the full CSV file will be returned for every request.

- Use a case-insensitive option in your managed repository. For example, for a MySQL repository, change the collation of `managedobjectproperties.propvalue` to `utf8_general_ci`. For more information, see [Case insensitivity for a JDBC repo](#).

In general, to address case-sensitivity, focus on database-, table-, or column-level collation settings. Queries performed against repositories configured in this way are subject to the collation, and are used for comparison.

Synchronization situations and actions

The synchronization process assesses source and target objects, and the links between them, and then determines the *synchronization situation* that applies to each object. The process then performs a specific *action*, usually on the target object, depending on the assessed situation.

The action that is taken for each situation is defined in the `policies` section of your synchronization mapping.

The following excerpt of a sample mapping shows the defined actions in that sample:

```
{
  "policies": [
    {
      "situation": "CONFIRMED",
      "action": "UPDATE"
    },
    {
      "situation": "FOUND",
      "action": "LINK"
    },
    {
      "situation": "ABSENT",
      "action": "CREATE"
    },
    {
      "situation": "AMBIGUOUS",
      "action": "IGNORE"
    },
    {
      "situation": "MISSING",
      "action": "IGNORE"
    },
    {
      "situation": "SOURCE_MISSING",
      "action": "DELETE"
    },
    {
      "situation": "UNQUALIFIED",
      "action": "IGNORE"
    },
    {
      "situation": "UNASSIGNED",
```

```
        "action": "IGNORE"
    }
]
}
```

You can also define these actions using the admin UI:

1. From the navigation bar, click **Configure > Mappings**, and click the mapping to edit.
2. Click the **Behaviors** tab, expand the **Situational Event Scripts** node, and configure event actions.
3. Click **Save**.

NOTE

If you do not define an action for a particular situation, IDM takes the *default action* for that situation.

How IDM assesses synchronization situations

IDM performs reconciliation in two phases:

1. *Source reconciliation* accounts for source objects and associated links based on the configured mapping.
2. *Target reconciliation* iterates over the target objects that were not processed in the first phase.

For example, if a source object was deleted, the *source reconciliation* phase will not identify the target object that was previously linked to that source object. Instead, this *orphaned* target object is detected during the second phase.

Source reconciliation

During source reconciliation and liveSync, IDM iterates through the objects in the source resource. For reconciliation, the list of objects includes all objects that are available through the connector. For liveSync, the list contains only changed objects. IDM can filter objects from the list by using the following:

- Scripts specified in the `validSource` property
- A query specified in the `sourceCondition` property
- A query specified in the `sourceQuery` property

For each object in the list, IDM assesses the following conditions:

1. Is the source object valid?

Valid source objects are categorized `qualifies=1`. Invalid source objects are categorized `qualifies=0`. Invalid objects include objects that were filtered out by a `validSource` script or `sourceCondition`. For more information, see [Filter Source and Target Objects With Scripts](#).

2. Does the source object have a record in the links table?

Source objects that have a corresponding link in the repository's `links` table are categorized `link=1`. Source objects that do not have a corresponding link are categorized `link=0`.

3. Does the source object have a corresponding valid target object?

Source objects that have a corresponding object in the target resource are categorized `target=1`. Source objects that do not have a corresponding object in the target resource are categorized `target=0`.

The following diagram illustrates the categorization of four sample objects during source reconciliation. In this example, the source is the managed user repository and the target is an LDAP directory:

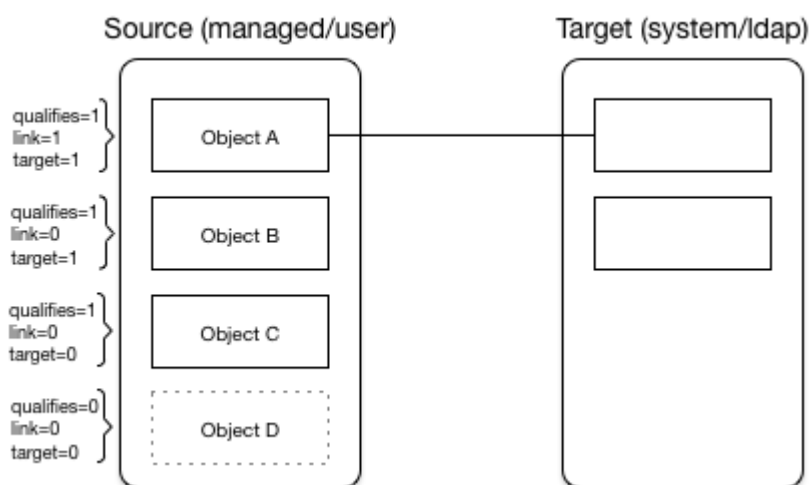


Figure 1. Object Categorization During the Source Synchronization Phase

Based on the categorizations of source objects during the source reconciliation phase, the synchronization process assesses a *situation* for each source object, and executes the *action* that is configured for each situation.

Not all situations are detected during all synchronization types (reconciliation, implicit synchronization, and liveSync). The following table describes the set of synchronization situations detected during source reconciliation, the default action taken for each situation, and valid alternative actions that can be configured for each situation:

Situations Detected During Reconciliation and Source Change Events

Source Qualifies	Link Exists	Target Objects Found	Situation	Default Action	Possible Actions
NO	NO	0	SOURCE_IGNORED	IGNORE source object	EXCEPTION, REPORT, NOREPORT, ASYNC
NO	NO	1	UNQUALIFIED	DELETE target object	EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC
NO	NO	> 1	UNQUALIFIED	DELETE target objects	EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC
NO	YES	0	UNQUALIFIED	DELETE linked target object ^[1]	EXCEPTION, REPORT, NOREPORT, ASYNC
NO	YES	1	UNQUALIFIED	DELETE linked target object	EXCEPTION, REPORT, NOREPORT, ASYNC
NO	YES	> 1	UNQUALIFIED	DELETE linked target object	EXCEPTION, REPORT, NOREPORT, ASYNC
YES	NO	0	ABSENT	CREATE target object	EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC

Source Qualifies	Link Exists	Target Objects Found	Situation	Default Action	Possible Actions
YES	NO	1	FOUND	UPDATE target object	EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC
YES	NO	1	FOUND_ALREADY_LINKED [2]	EXCEPTION	IGNORE, REPORT, NOREPORT, ASYNC
YES	NO	> 1	AMBIGUOUS [3]	EXCEPTION	REPORT, NOREPORT, ASYNC
YES	YES	0	MISSING [4]	EXCEPTION	CREATE, UNLINK, DELETE, IGNORE, REPORT, NOREPORT, ASYNC
YES	YES	1	CONFIRMED	UPDATE target object	IGNORE, REPORT, NOREPORT, ASYNC

Based on this table, the following situations would be assigned to the previous diagram:

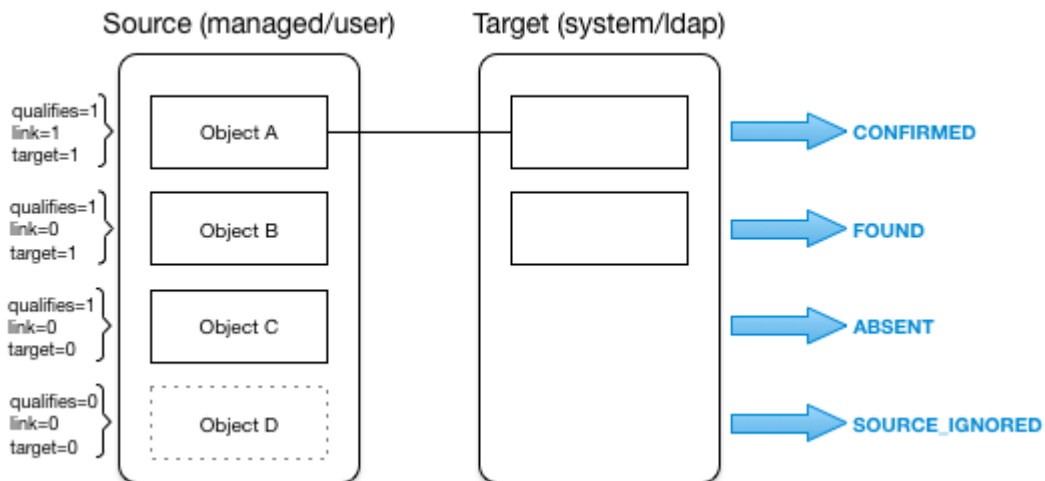


Figure 2. Situation Assignment During the Source Synchronization Phase

Target reconciliation

During source reconciliation, the synchronization process cannot detect situations where no source object exists. In this case, the situation is detected during the second reconciliation phase, target reconciliation.

Target reconciliation iterates through the target objects that were not accounted for during source reconciliation. The process checks each object against the `validTarget` filter, determines the appropriate situation, and executes the action configured for the situation. Target reconciliation evaluates the following conditions:

1. Is the target object valid?

Valid target objects are categorized `qualifies=1`. Invalid target objects are categorized `qualifies=0`. Invalid objects include objects that were filtered out by a `validTarget` script. For more information, see [Filter Source and Target Objects With Scripts](#).

2. Does the target object have a record in the links table?

Target objects that have a corresponding link in the `links` table are categorized `link=1`. Target objects that do not have a corresponding link are categorized `link=0`.

3. Does the target object have a corresponding source object?

Target objects that have a corresponding object in the source resource are categorized `source=1`. Target objects that do not have a corresponding object in the source resource are categorized `source=0`.

The following diagram illustrates the categorization of three sample objects during target reconciliation:

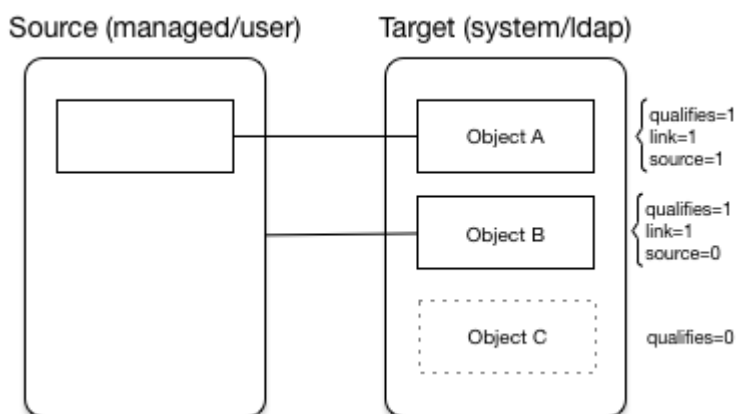


Figure 3. Object Categorization During the Target Synchronization Phase

Based on the categorizations of target objects during the target reconciliation phase, a *situation* is assessed for each remaining target object. Not all situations are detected in

all synchronization types. The following table describes the set of situations that can be detected during the target reconciliation phase:

Situations Detected During Target Reconciliation

Target Qualifies	Link Exists	Source Exists	Source Qualifies	Situation	Default Action	Possible Actions
NO	n/a	n/a	n/a	TARGET_IGNORED [5]	IGNORE	DELETE, UNLINK, REPORT, NOREPORT, ASYNC
YES	NO	NO	n/a	UNASSIGNED	EXCEPTION	IGNORE, REPORT, NOREPORT, ASYNC
YES	YES	YES	YES	CONFIRMED	UPDATE target object	IGNORE, REPORT, NOREPORT
YES	YES	YES	NO	UNQUALIFIED [6]	DELETE	UNLINK, EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC
YES	YES	NO	n/a	SOURCE_MISSING [7]	EXCEPTION	DELETE, UNLINK, IGNORE, REPORT, NOREPORT, ASYNC

Based on this table, the following situations would be assigned to the previous diagram:

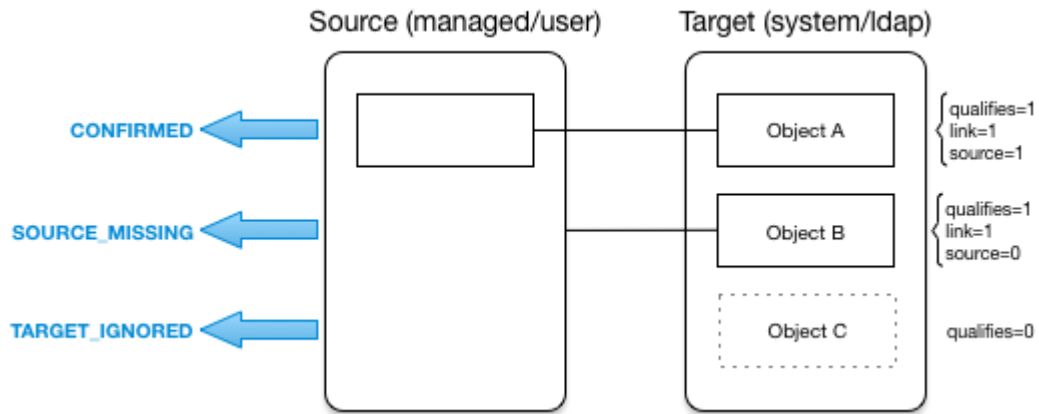


Figure 4. Situation Assignment During the Target Synchronization Phase

Situations specific to implicit synchronization and liveSync

Certain situations occur only during implicit synchronization (when changes made in the repository are pushed out to external systems) and liveSync (when IDM polls external system change logs for changes and updates the repository).

The following table shows the situations that pertain only to implicit sync and liveSync, when records are *deleted* from the source or target resource.

Situations detected during target reconciliation

Source Qualifies	Link Exists	Targets Found ^[8]	Targets Qualify	Situation	Default Action	Possible Actions
n/a	YES	0	n/a	LINK_ONLY	EXCEPTION	IGNORE, REPORT, NOREPORT, ASYNC
n/a	YES	1	1	SOURCE_MISSING	EXCEPTION	IGNORE, REPORT, NOREPORT, ASYNC
n/a	YES	1	0	TARGET_IGNORED	IGNORE	DELETE, UNLINK, EXCEPTION, REPORT, NOREPORT, ASYNC

Source Qualifies	Link Exists	Targets Found ^[8]	Targets Qualify	Situation	Default Action	Possible Actions
n/a	NO	0	n/a	ALL_GONE	IGNORE	EXCEPTION, REPORT, NOREPORT, ASYNC
YES	NO	0	n/a	ALL_GONE	IGNORE	EXCEPTION, REPORT, NOREPORT, ASYNC
YES	NO	1	1	UNASSIGNED	EXCEPTION	REPORT, NOREPORT
YES	NO	> 1	> 1	AMBIGUOUS	EXCEPTION	IGNORE, REPORT, NOREPORT, ASYNC
NO	NO	0	n/a	ALL_GONE	IGNORE	EXCEPTION, REPORT, NOREPORT, ASYNC
NO	NO	1	1	TARGET_IGNORED	IGNORE target object	DELETE, UNLINK, EXCEPTION, REPORT, NOREPORT, ASYNC
NO	NO	> 1	> 1	UNQUALIFIED	DELETE target objects	EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC

Synchronization actions

When an object has been assigned a situation, the synchronization process takes the configured *action* on that object. If no action is configured, the default action for that

situation applies.

The following actions can be taken:

CREATE

Create and link a target object.

UPDATE

Link and update a target object.

DELETE

Delete and unlink the target object.

LINK

Link the correlated target object.

UNLINK

Unlink the linked target object.

EXCEPTION

Flag the link situation as an exception.

IMPORTANT

Do *not* use this action for liveSync mappings.

In the context of liveSync, the EXCEPTION action triggers the liveSync failure handler, and the operation is retried in accordance with the configured retry policy. This is not useful because the operation will never succeed. If the configured number of retries is high, these pointless retries can continue for a long period of time.

If the maximum number of retries is exceeded, the liveSync operation terminates and does not continue processing the entry that follows the failed (EXCEPTION) entry. LiveSync is only resumed at the next liveSync polling interval.

This behavior differs from reconciliation, where a failure to synchronize a single source-target association does not fail the entire reconciliation.

IGNORE

Do not change the link or target object state.

REPORT

Do not perform any action but report what would happen if the default action were performed.

NOREPORT

Do not perform any action or generate any report.

ASYNC

An asynchronous process has been started, so do not perform any action or generate any report.

Launch a script as an action

In addition to the static synchronization actions described previously, you can provide a script to run in specific synchronization situations. The script can be either JavaScript or Groovy. You can specify the script inline (with the "source" property), or reference it from a file, (with the "file" property).

The following excerpt of a sample mapping specifies that an inline script should be invoked when a synchronization operation assesses an entry as `ABSENT` in the target system. The script checks whether the `employeeType` property of the corresponding source entry is `contractor`. If so, the source entry is ignored. Otherwise, the entry is created on the target system:

```
{
  "situation" : "ABSENT",
  "action" : {
    "type" : "text/javascript",
    "globals" : { },
    "source" : "if (source.employeeType === 'contractor')
{action='IGNORE'}
                else {action='CREATE'};action;"
  },
}
```

The following variables are available to a script that is called as an action:


- `source`
- `target`
- `linkQualifier`
- `recon` (where `recon.actionParam` contains information about the current reconciliation operation)

For more information about the variables available to scripts, see [Script variables](#).

The result obtained from evaluating this script must be a string whose value is one of the synchronization actions listed in [Synchronization actions](#). This resulting action is shown in the reconciliation log.

To launch a script as a synchronization action using the admin UI:

1. From the navigation bar, click **Configure > Mappings**, and click the mapping to edit.
2. Click the **Behaviors** tab, and expand the **Policies** node.

3. Click the edit button  for the situation action to edit.
4. On the **Perform this Action** tab, click **Script**, and enter the script that corresponds to the action.
5. Click **Submit**, and then click **Save**.

Launch a workflow as an action


The `triggerWorkflowFromSync.js` script launches a predefined workflow when a synchronization operation assesses a particular situation. The mechanism for triggering this script is the same as for any other script. The script is provided in the `openidm/bin/defaults/script/workflow` directory. If you customize the script, copy it to the `script` directory of your project to ensure that your customizations are preserved during an upgrade.

The parameters for the workflow are passed as properties of the `action` parameter.

The following extract of a sample mapping specifies that, when a synchronization operation assesses an entry as `ABSENT`, the workflow named `managedUserApproval` is invoked:

```
{
  "situation" : "ABSENT",
  "action" : {
    "workflowName" : "managedUserApproval",
    "type" : "text/javascript",
    "file" : "workflow/triggerWorkflowFromSync.js"
  }
}
```

To launch a workflow as a synchronization action Using the admin UI:

1. From the navigation bar, click **Configure > Mappings**, and click the mapping to edit.
2. Click the **Behaviors** tab, and expand the **Policies** node.
3. Click the edit button  for the situation action to edit.
4. On the **Perform this Action** tab, click **Workflow**, and enter the details of the workflow to launch.
5. Click **Submit**, and then click **Save**.

Correlate source objects with existing target objects

When a synchronization operation creates an object on a target system, it also creates a *link* between the source and target object. IDM then uses that link to determine the

object's *synchronization situation* during later synchronization operations. For a list of synchronization situations, see [How IDM assesses synchronization situations](#).

Every synchronization operation can *correlate* existing source and target objects. Correlation matches source and target objects, based on the results of a query or script, and creates links between matched objects.

Correlation queries and correlation scripts are configured as part of the mapping. Each query or script is specific to the mapping for which it is configured.

Configure correlation using the admin UI

1. From the navigation bar, click **Configure > Mappings**.
2. From the **Mappings** page, click the mapping to correlate.
3. From the **Mapping Detail** page, click the **Association** tab.
4. Expand the **Association Rules** node, click the drop-down menu, and select one of the following:
 - **Correlation Queries**
 - **Correlation Script**
5. Build and/or write your script or query, and click **Save**.

Correlation queries

IDM processes a correlation query by constructing a query map. The content of the query is generated dynamically, using values from the source object. For each source object, a new query is sent to the target system, using (possibly transformed) values from the source object for its execution.

Queries are run against *target resources*, either managed or system objects, depending on the mapping. Correlation queries on system objects access the connector, which executes the query on the external resource.

You express a correlation query using a query filter (`_queryFilter`). For more information about query filters, see [Define and call data queries](#). The synchronization process executes the correlation query to search through the target system for objects that match the current source object.

To configure a correlation query, define a script whose source returns a query that uses the `_queryFilter` , for example:

```
{ "_queryFilter" : "uid eq \"\" + source.userName + "\"\" }
```

Use filtered queries to correlate objects

For filtered queries, the script that is defined or referenced in the `correlationQuery` property must return an object with the following elements:

- The element that is being compared on the target object; for example, `uid`.

The element on the target object is not necessarily a single attribute. Your query filter can be simple or complex; valid query filters range from a single operator to an entire boolean expression tree.

If the target object is a system object, this attribute must be referred to by its IDM name rather than its ICF `nativeName`. For example, with the following provisioner configuration, the attribute to use in the correlation query would be `uid` and not `+NAME``:

```
...
  "uid" : {
    "type" : "string",
    "nativeName" : "__NAME__",
    "required" : true,
    "nativeType" : "string"
  }
...
```

- The value to search for in the query.

This value is generally based on one or more values from the source object. However, it does not have to match the value of a single source object property. You can define how your script uses the values from the source object to find a matching record in the target system.

You might use a transformation of a source object property, such as `toUpperCase()`. You can concatenate that output with other strings or properties. You can also use this value to call an external REST endpoint, and redirect the response to the final "value" portion of the query.

The following correlation query matches source and target objects if the value of the `uid` attribute on the target is the same as the `userName` attribute on the source:

```
"correlationQuery" : {
  "type" : "text/javascript",
  "source" : "var qry = {'_queryFilter': 'uid eq \'' +
source.userName + '\''}"; qry"
},
```

The query can return zero or more objects. The situation assigned to the source object depends on the number of target objects that are returned, and on the presence of any

link qualifiers in the query. For information about synchronization situations, see [How Synchronization Situations Are Assessed](#). For information about link qualifiers, see [Map a Single Source Object to Multiple Target Objects](#).

Create Correlation Queries Using the Expression Builder

The *Expression Builder* is a wizard that lets you quickly build expressions using drop-down menu options.

1. From the navigation bar, click **Configure > Mappings**.
2. On the **Mappings** page, click the mapping to correlate.
3. From the **Mapping Detail** page, click the **Association** tab.
4. Expand the **Association Rules** node, click the drop-down menu, and select **Correlation Queries**.
5. Click **Add Correlation Query**.
6. In the **Correlation Query** window, click the **Link Qualifier** drop-down menu, and select a link qualifier.

If you do not need to correlate multiple potential target objects per source object, select the **default** link qualifier. For more information about linking to multiple target objects, see [Map a Single Source Object to Multiple Target Objects](#).

7. Select **Expression Builder**.
8. To create an expression, use the drop-down menus to add and remove items, as necessary. List the fields to use for matching existing items in your source to items in your target.

The following example displays an **Expression Builder** correlation query for a mapping from `managed/user` to `system/ldap/accounts` objects. The query creates a match between the source (managed) object and the target (LDAP) object if the value of the `givenName` or the `telephoneNumber` of those objects is the same.

Correlation Query



Create using Expression Builder or Script..

Link Qualifier:

default

Expression Builder

List the fields which will be used to match existing items in your source to items in your target:

Any of the following fields

givenName

telephoneNumber



Script

Cancel

Submit

9. After you finish building the expression, click **Submit**.

10. On the **Mapping Detail** page, under the **Association Rules** node, click **Save**.

The correlation query displays as follows in the mapping:

```
"correlationQuery" : [
  {
    "linkQualifier" : "default",
    "expressionTree" : {
      "any" : [
        "givenName",
        "telephoneNumber"
      ]
    },
    "mapping" : "managedUser_systemLdapAccounts",
    "type" : "text/javascript",
    "file" : "ui/correlateTreeToQueryFilter.js"
  }
]
```

Correlation scripts

In general, a correlation query should meet the requirements of most deployments. However, if you need a more powerful correlation mechanism than a simple query can provide, you can write a correlation script with additional logic. Correlation scripts can be useful if your query needs extra processing, such as fuzzy-logic matching or out-of-band verification with a third-party service over REST. Correlation scripts are generally more complex than correlation queries, and impose no restrictions on the methods used to find matching objects.

A correlation script must execute a query and return the result of that query. The result of a correlation script is a list of maps, each of which contains a candidate `_id` value. If no match is found, the script returns a zero-length list. If exactly one match is found, the script returns a single-element list. If there are multiple ambiguous matches, the script returns a list with multiple elements. There is no assumption that the matching target record or records can be found by a simple query on the target system. All of the work required to find matching records is left to the script.

To invoke a correlation script, use one of the following properties:

correlationQuery

Returns a `Map` whose values specify the `QueryFilter` for the sync engine to execute.

correlationScript

Returns a `List<Map>` whose value is a list of correlated objects from the target.

You can invoke a correlation script inline:

```
"correlationScript" : {
  "type": "text/javascript",
  "source": " var resultData =
openidm.query("system/ldap/account", myQuery); return
resultData.result;"
}
```

You can also invoke a correlation script using a script file:

```
"correlationScript" : {
  "type": "text/javascript",
  "file": "myCustomCorrelationScript.js"
}
```

Correlation Script Using Link Qualifiers

The following example shows a correlation script that uses link qualifiers. The script returns `resultData.result`—a list of maps, each of which has an `_id` entry. These entries will be the values that are used for correlation.

```
(function () {
  var query, resultData;
  switch (linkQualifier) {
    case "test":
      logger.info("linkQualifier = test");
      query = {'_queryFilter': 'uid eq \'' +
```

```

source.userName + '-test\''});
    break;
    case "user":
        logger.info("linkQualifier = user");
        query = {'_queryFilter': 'uid eq \'' +
source.userName + '\''});
        break;
    case "default":
        logger.info("linkQualifier = default");
        query = {'_queryFilter': 'uid eq \'' +
source.userName + '\''});
        break;
    default:
        logger.info("No linkQualifier provided.");
        break;
}
var resultData = openidm.query("system/ldap/account", query);
logger.info("found " + resultData.result.length + " results
for link qualifier " + linkQualifier)
for (i=0;i<resultData.result.length;i++) {
    logger.info("found target: " + resultData.result[i]._id);
}
return resultData.result;
} ());

```

Configure a correlation script using the admin UI

1. From the navigation bar, click **Configure > Mappings**.
2. On the **Mappings** page, select the mapping to correlate.
3. From the **Mapping Detail** page, click the **Association** tab.
4. Expand the **Association Rules** node, click the drop-down menu, and select **Correlation Script**.
5. From the **Type** drop-down menu, select **JavaScript** or **Groovy**.
6. Enter the correlation script:
 - To use an inline script, select **Inline Script**, and type the script source.
 - To use a script file, select **File Path**, and enter the path to the script.

To create a correlation script, use the details from the source object to find the matching record in the target system. If you are using link qualifiers to match a single source record to multiple target records, you must also use the value of the `linkQualifier` variable within your correlation script to find the target ID that applies for that qualifier.

7. Click **Save**.

Synchronization operations

Manage reconciliation

To trigger, cancel, and monitor reconciliation operations over REST, use the `openidm/recon` REST endpoint. You can perform most of these actions using the admin UI.

Trigger a reconciliation

The following example triggers a reconciliation operation over REST based on the `systemLdapAccounts_managedUser` mapping:

```
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/recon?
_action=recon&mapping=systemLdapAccounts_managedUser"
```

By default, a reconciliation run ID is returned immediately when the reconciliation operation is initiated. Clients can make subsequent calls to the reconciliation service, using this reconciliation run ID to query its state, and to call operations on it. For an example, see [Reconciliation Details](#).

The reconciliation run initiated previously would return something similar to the following:

```
{
  "_id": "05f63bce-4aaa-492e-9e86-a702d5c9d6c0-1144",
```

```
"state": "ACTIVE"
}
```

To complete the reconciliation operation before the reconciliation run ID is returned, set the `waitForCompletion` property to `true` when the reconciliation is initiated:

```
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/recon?
_action=recon&mapping=systemLdapAccounts_managedUser&waitForComple
tion=true"
```

TIP

To trigger this reconciliation using the admin UI, click **Configure > Mappings**, select a mapping, then click **Reconcile**.

If you click **Cancel Reconciliation** before it completes, you will need to start the reconciliation again.

Cancel a reconciliation

To cancel an in progress reconciliation, specify the reconciliation run ID. The following REST call cancels the reconciliation run initiated 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 POST \
"http://localhost:8080/openidm/recon/0890ad62-4738-4a3f-8b8e-
f3c83bbf212e?_action=cancel"
```

The output for a reconciliation cancellation request is similar to the following:

```
{
  "status": "INITIATED",
  "action": "cancel",
  "_id": "0890ad62-4738-4a3f-8b8e-f3c83bbf212e"
}
```

If the reconciliation run is waiting for completion before its ID is returned, obtain the reconciliation run ID from the list of active reconciliations, as described in the following section.

TIP

To cancel a reconciliation run in progress using the admin UI, click **Configure > Mappings**, click on the mapping reconciliation to cancel, and click **Cancel Reconciliation**.

List reconciliation history

Display a list of reconciliation processes that have completed, and those that are in progress, by running a RESTful GET on "http://localhost:8080/openidm/recon".

The following example displays all reconciliation runs:

```
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/recon"
```

▼ [Example Output](#)

The output is similar to the following, with one item for each reconciliation run:

```
"reconciliations": [
  {
    "_id": "05f63bce-4aaa-492e-9e86-a702d5c9d6c0-1144",
    "mapping": "systemLdapAccounts_managedUser",
    "state": "SUCCESS",
    "stage": "COMPLETED_SUCCESS",
    "stageDescription": "reconciliation completed.",
    "progress": {
      "source": {
        "existing": {
          "processed": 2,
          "total": "2"
        }
      },
      "target": {
        "existing": {
          "processed": 0,
          "total": "0"
        }
      }
    }
  }
]
```



```
    },
    "created": 2,
    "unchanged": 0,
    "updated": 0,
    "deleted": 0
  },
  "links": {
    "existing": {
      "processed": 0,
      "total": "0"
    },
    "created": 2
  }
},
"situationSummary": {
  "SOURCE_IGNORED": 0,
  "FOUND_ALREADY_LINKED": 0,
  "UNQUALIFIED": 0,
  "ABSENT": 2,
  "TARGET_IGNORED": 0,
  "MISSING": 0,
  "ALL_GONE": 0,
  "UNASSIGNED": 0,
  "AMBIGUOUS": 0,
  "CONFIRMED": 0,
  "LINK_ONLY": 0,
  "SOURCE_MISSING": 0,
  "FOUND": 0
},
"statusSummary": {
  "SUCCESS": 2,
  "FAILURE": 0
},
"durationSummary": {
  "sourceQuery": {
    "min": 42,
    "max": 42,
    "mean": 42,
    "count": 1,
    "sum": 42,
    "stdDev": 0
  },
  "auditLog": {
    "min": 0,
    "max": 1,
```

```
    "mean": 0,  
    "count": 24,  
    "sum": 15,  
    "stdDev": 0  
  },  
  "linkQuery": {  
    "min": 5,  
    "max": 5,  
    "mean": 5,  
    "count": 1,  
    "sum": 5,  
    "stdDev": 0  
  },  
  "targetQuery": {  
    "min": 3,  
    "max": 3,  
    "mean": 3,  
    "count": 1,  
    "sum": 3,  
    "stdDev": 0  
  },  
  "targetPhase": {  
    "min": 0,  
    "max": 0,  
    "mean": 0,  
    "count": 1,  
    "sum": 0,  
    "stdDev": 0  
  },  
  "sourceObjectQuery": {  
    "min": 6,  
    "max": 34,  
    "mean": 21,  
    "count": 22,  
    "sum": 474,  
    "stdDev": 9  
  },  
  "postMappingScript": {  
    "min": 0,  
    "max": 1,  
    "mean": 0,  
    "count": 22,  
    "sum": 17,  
    "stdDev": 0  
  },  
}
```

```

    "onMappingScript": {
      "min": 0,
      "max": 4,
      "mean": 2,
      "count": 22,
      "sum": 48,
      "stdDev": 2
    },
    "sourcePhase": {
      "min": 490,
      "max": 490,
      "mean": 490,
      "count": 1,
      "sum": 490,
      "stdDev": 0
    }
  },
  "parameters": {
    "sourceQuery": {
      "resourceName": "system/ldap/account",
      "queryFilter": "true",
      "_fields": "_id"
    },
    "targetQuery": {
      "resourceName": "managed/user",
      "queryFilter": "true",
      "_fields": "_id"
    }
  },
  "started": "2020-05-07T09:14:57.740Z",
  "ended": "2020-05-07T09:14:58.325Z",
  "duration": 585,
  "sourceProcessedByNode": {}
}
]

```

You can adjust the number of reconciliation runs that are stored in IDM by adding the `maxAnalysisRunsPerMapping` and `maxNonAnalysisRunsPerMapping` properties to your [mapping](#):

```

"reconAssociation" : {
  "maxAnalysisRunsPerMapping" : 1,
  "maxNonAnalysisRunsPerMapping" : 3
}

```

In this context, *analysis* refers to reconciliation runs that are triggered with the `analyze=true` parameter. These runs don't perform any actions, but determine which actions *would* be performed in a real reconciliation. Non-analysis refers to a normal reconciliation. The default value for both properties is `1`.

In contrast, the admin UI displays the results of only the most recent reconciliation. For more information, see [synchronization-guide:manage-recon.adoc#recon-details-ui](#).

Reconciliation Properties

Each reconciliation run includes the following properties:

_id

The ID of the reconciliation run.

mapping

The name of the [mapping](#).

state

The high-level state of the reconciliation run. Values can be as follows:

- ACTIVE

The reconciliation run is in progress.

- CANCELED

The reconciliation run was successfully canceled.

- FAILED

The reconciliation run was terminated because of failure.

- SUCCESS

The reconciliation run completed successfully.

stage

The current stage of the reconciliation run. Values can be as follows:

- ACTIVE_INITIALIZED

The initial stage, when a reconciliation run is first created.

- ACTIVE_QUERY_ENTRIES

Querying the source, target, and possibly link sets to reconcile.

- ACTIVE_RECONCILING_SOURCE

Reconciling the set of IDs retrieved from the mapping source.

- ACTIVE_RECONCILING_TARGET

Reconciling any remaining entries from the set of IDs retrieved from the mapping target, that were not matched or processed during the source phase.

- ACTIVE_LINK_CLEANUP

Checking whether any links are now unused and should be cleaned up.

- ACTIVE_PROCESSING_RESULTS

Post-processing of reconciliation results.

- ACTIVE_CANCELING

Attempting to abort a reconciliation run in progress.

- COMPLETED_SUCCESS

Successfully completed processing the reconciliation run.

- COMPLETED_CANCELED

Completed processing because the reconciliation run was aborted.

- COMPLETED_FAILED

Completed processing because of a failure.

stageDescription

A description of the stages described previously.

progress

The progress object has the following structure (annotated here with comments):

```
"progress":{
  "source":{                                // Progress on set of existing entries
in the mapping source
    "existing":{
      "processed":1001,
      "total":"1001" // Total number of entries in source
set, if known, "?" otherwise
    }
  },
  "target":{                                // Progress on set of existing entries
in the mapping target
    "existing":{
      "processed":1001,
      "total":"1001" // Total number of entries in target
set, if known, "?" otherwise
    },
    "created":0 // New entries that were created
  },
}
```

```

"links":{
    // Progress on set of existing links
    // between source and target
    "existing":{
        "processed":1001,
        "total":"1001" // Total number of existing links,
        // if known, "?" otherwise
    },
    "created":0 // Denotes new links that were created
    }
},

```

Reconciliation details

To display the details of a specific reconciliation over REST, include the reconciliation run ID in the URL. The following call shows the details of the reconciliation run initiated in [synchronization-guide:manage-recon.adoc#triggering-recons](#).

```

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/recon/05f63bce-4aaa-492e-9e86-
a702d5c9d6c0-1144"

```

▼ [Example Output](#)

```

{
  "_id": "05f63bce-4aaa-492e-9e86-a702d5c9d6c0-1144",
  "mapping": "systemLdapAccounts_managedUser",
  "state": "SUCCESS",
  "stage": "COMPLETED_SUCCESS",
  "stageDescription": "reconciliation completed.",
  "progress": {
    "source": {
      "existing": {
        "processed": 2,
        "total": "2"
      }
    },
    "target": {
      "existing": {
        "processed": 0,
        "total": "0"
      }
    }
  }
}

```

```

    },
    "created": 2,
    "unchanged": 0,
    "updated": 0,
    "deleted": 0
  },
  "links": {
    "existing": {
      "processed": 0,
      "total": "0"
    },
    "created": 2
  }
},
"situationSummary": {
  "SOURCE_IGNORED": 0,
  "FOUND_ALREADY_LINKED": 0,
  "UNQUALIFIED": 0,
  "ABSENT": 2,
  "TARGET_IGNORED": 0,
  "MISSING": 0,
  "ALL_GONE": 0,
  "UNASSIGNED": 0,
  "AMBIGUOUS": 0,
  "CONFIRMED": 0,
  "LINK_ONLY": 0,
  "SOURCE_MISSING": 0,
  "FOUND": 0
},
"statusSummary": {
  "SUCCESS": 2,
  "FAILURE": 0
},
"durationSummary": {
  "sourceQuery": {
    "min": 42,
    "max": 42,
    "mean": 42,
    "count": 1,
    "sum": 42,
    "stdDev": 0
  },
  "auditLog": {
    "min": 0,
    "max": 1,

```

```
    "mean": 0,
    "count": 24,
    "sum": 15,
    "stdDev": 0
  },
  "linkQuery": {
    "min": 5,
    "max": 5,
    "mean": 5,
    "count": 1,
    "sum": 5,
    "stdDev": 0
  },
  "targetQuery": {
    "min": 3,
    "max": 3,
    "mean": 3,
    "count": 1,
    "sum": 3,
    "stdDev": 0
  },
  "targetPhase": {
    "min": 0,
    "max": 0,
    "mean": 0,
    "count": 1,
    "sum": 0,
    "stdDev": 0
  },
  "sourceObjectQuery": {
    "min": 6,
    "max": 34,
    "mean": 21,
    "count": 22,
    "sum": 474,
    "stdDev": 9
  },
  "postMappingScript": {
    "min": 0,
    "max": 1,
    "mean": 0,
    "count": 22,
    "sum": 17,
    "stdDev": 0
  },
}
```



```

"onMappingScript": {
  "min": 0,
  "max": 4,
  "mean": 2,
  "count": 22,
  "sum": 48,
  "stdDev": 2
},
"sourcePhase": {
  "min": 490,
  "max": 490,
  "mean": 490,
  "count": 1,
  "sum": 490,
  "stdDev": 0
}
},
"parameters": {
  "sourceQuery": {
    "resourceName": "system/ldap/account",
    "queryFilter": "true",
    "_fields": "_id"
  },
  "targetQuery": {
    "resourceName": "managed/user",
    "queryFilter": "true",
    "_fields": "_id"
  }
},
"started": "2020-05-07T09:14:57.740Z",
"ended": "2020-05-07T09:14:58.325Z",
"duration": 585,
"sourceProcessedByNode": {}
}

```

View reconciliation details using the admin UI

You can display the details of the most recent reconciliation in the admin UI. Select the mapping. In the page that displays, you'll see a message similar to:

```
Completed: Last reconciled November 20, 2019 15:28
```

Clicking on the reconciliation run date displays the details of the reconciliation run. Click **Reconciliation Results** for additional information.

If a reconciliation fails, select the **Failure Summary** tab for more information about the failure.

To view reconciliation audit logs in the UI, add an [Audit widget](#) to your dashboard. The reconciliation Audit widget shows the same information that you get over REST.

Reconciliation association details

When performing a reconciliation run, information is reconciled between the source object and the target object. This creates an association between the two objects, which can be recorded in IDM by including the `persistAssociations=true` parameter when triggering a reconciliation. This information can then be retrieved by querying the `recon/assoc` endpoint.

To get a list of currently stored recon associations, run the following query:

```
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/recon/assoc?_queryFilter=true"
{
  "result": [
    {
      "_id": "da88b9a5-1fe5-4f8d-a6a8-7e0a2b4e136b-9230",
      "_rev": "1",
      "mapping": "managedUser_systemLdapAccounts",
      "sourceResourceCollection": "managed/user",
      "targetResourceCollection": "system/ldap/account",
      "isAnalysis": "false",
      "finishTime": "2019-05-01T23:36:24.434153Z"
    },
    {
      "_id": "da88b9a5-1fe5-4f8d-a6a8-7e0a2b4e136b-99638",
      "_rev": "1",
      "mapping": "systemLdapAccounts_managedUser",
      "sourceResourceCollection": "system/ldap/account",
      "targetResourceCollection": "managed/user",
      "isAnalysis": "true",
      "finishTime": "2019-05-06T21:31:42.140066Z"
    }
  ],
  "resultCount": 2,
  "pagedResultsCookie": null,
```

```
"totalPagedResultsPolicy": "NONE",
"totalPagedResults": -1,
"remainingPagedResults": -1
}
```

You can also get information for a specific reconciliation by querying the recon 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/recon/assoc/da88b9a5-1fe5-4f8d-
a6a8-7e0a2b4e136b-9230"
{
  "_id": "da88b9a5-1fe5-4f8d-a6a8-7e0a2b4e136b-9230",
  "_rev": "1",
  "mapping": "managedUser_systemLdapAccounts",
  "sourceResourceCollection": "managed/user",
  "targetResourceCollection": "system/ldap/account",
  "isAnalysis": "false",
  "finishTime": "2019-05-01T23:36:24.434153Z"
}
```

It is possible to also get the specific association details of each entry in the reconciliation run by appending `/entry` to your query:

```
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/recon/assoc/da88b9a5-1fe5-4f8d-
a6a8-7e0a2b4e136b-9230/entry?_queryFilter=true"
{
  "result": [
    {
      "_id": "400d40fd-da58-41f5-857b-71855eb97bd9",
      "_rev": "0",
      "mapping": "managedUser_systemLdapAccounts",
      "reconId": "da88b9a5-1fe5-4f8d-a6a8-7e0a2b4e136b-9230",
      "situation": "CONFIRMED",
      "action": "UPDATE",
      "linkQualifier": "default",

```

```

    "sourceObjectId": "07978ba5-b31d-4f8b-9f60-506c07f68495",
    "targetObjectId": "ca8abc7f-7b97-3e96-94fb-6b27b0ec5aed",
    "sourceResourceCollection": "managed/user",
    "targetResourceCollection": "system/ldap/account",
    "status": "SUCCESS",
    "exception": null,
    "message": null,
    "messageDetail": "null",
    "ambiguousTargetObjectIds": null
  },
  ...
  {
    "_id": "08ec633c-744f-4092-b88d-fe253b1d8e52",
    "_rev": "0",
    "mapping": "managedUser_systemLdapAccounts",
    "reconId": "da88b9a5-1fe5-4f8d-a6a8-7e0a2b4e136b-9230",
    "situation": "CONFIRMED",
    "action": "UPDATE",
    "linkQualifier": "default",
    "sourceObjectId": "ee2449a8-01e6-4c0b-84d3-e65e25c3e38c",
    "targetObjectId": "67a6596e-ebfc-3542-a664-1ab1610e082a",
    "sourceResourceCollection": "managed/user",
    "targetResourceCollection": "system/ldap/account",
    "status": "SUCCESS",
    "exception": null,
    "message": null,
    "messageDetail": "null",
    "ambiguousTargetObjectIds": null
  }
],
...
}

```

NOTE

For particularly large reconciliations, the results returned can be quite substantial, since it includes the details of every object reconciled. We encourage using query filters to tune your queries to only return the subset of results you're looking for.

Purge reconciliation statistics

When the number of completed reconciliation runs for a given mapping reaches the number specified by `maxAnalysisRunsPerMapping` or `maxNonAnalysisRunsPerMapping`, statistics are purged automatically. Statistics and

reconciliation run information (such as recon associations) are purged chronologically by mapping, with the oldest reconciliation run for that mapping purged first.

You can also manually remove reconciliation statistics. To purge reconciliation statistics from the repository manually, run a DELETE command on the reconciliation run ID. For 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/recon/da88b9a5-1fe5-4f8d-a6a8-7e0a2b4e136b-9230"
```

Manage liveSync

Because you can trigger liveSync operations using REST (or the resource API) you can use an external scheduler to trigger liveSync operations, rather than using the IDM scheduling mechanism.

There are two ways to trigger liveSync over REST:

- Use the `_action=liveSync` parameter directly on the resource. This is the recommended method. The following example calls liveSync on the user accounts in an external LDAP system:

```
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/system/ldap/account?_action=liveSync"
```

- Target the `system` endpoint and supply a `source` parameter to identify the object that should be synchronized. This method matches the scheduler configuration and can therefore be used to test schedules before they are implemented.

The following example calls the same liveSync operation as the previous 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/system?  
_action=liveSync&source=system/ldap/account"
```


A successful liveSync operation returns the following response:

```
{  
  "_rev": "000000001ade755f",  
  "_id": "SYSTEMLDAPACCOUNT",  
  "connectorData": {  
    "nativeType": "JAVA_TYPE_LONG",  
    "syncToken": 1  
  }  
}
```

Do not run two identical liveSync operations simultaneously. Rather, ensure that the first operation has completed before launching a second, similar operation.

Trigger liveSync using the admin UI

LiveSync operations are specific to a system object type (such as `system/ldap/account`). Apart from scheduling liveSync, as described in [Scheduling LiveSync Through the UI](#), you can launch a liveSync operation on demand for a particular system object type as follows:

1. From the navigation bar, click **Configure > Connectors**.
2. On the **Connectors** page, select a connector.
3. On the **connector-name** page, click the **Object Types** tab.
4. Click the edit button  adjacent to the object type to synchronize.
5. Click the **Sync** tab, and then click **Sync Now**.

The **Sync Token** field displays the current synchronization token for the object type.

Troubleshoot liveSync failures

To troubleshoot a liveSync operation that has not succeeded, include the `detailedFailure` parameter to return additional information. 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/system?  
_action=liveSync&source=system/ldap/account&detailedFailure=true"
```

```
--request POST \  
"http://localhost:8080/openidm/system/ldap/account?  
_action=liveSync&detailedFailure=true"
```

The first time liveSync is called, it does not have a synchronization token in the database to establish which changes have already been processed. The default liveSync behavior is to locate the last existing entry in the change log, and to store that entry in the database as the current starting position from which changes should be applied. This behavior prevents liveSync from processing changes that might already have been processed during an initial data load. Subsequent liveSync operations will pick up and process any new changes.

Typically, in setting up liveSync on a new system, you would load the data initially (by using reconciliation, for example) and then enable liveSync, starting from that base point.

In the case of DS, the change log (`cn=changelog`) can be read only by `uid=admin` by default. If you are configuring liveSync with DS, the `principal` that is defined in the LDAP connector configuration must have access to the change log. For information about allowing a regular user to read the change log, see [Allow a User or Application to Read the Change Log](#).

If you see the following error message, you might have forgotten to set `changelog-read` access for a regular user:

```
Unable to locate the DS replication change log suffix. Please make  
sure it's enabled, and changelog-read access is granted.
```

Filter synchronization data

By default, IDM synchronizes all objects that match those defined in the connector configuration for the resource. Many connectors let you limit the scope of objects that the connector accesses. For example, the LDAP connector lets you specify base DN's and LDAP filters so that you do not need to access every entry in the directory.

The following sections describe other ways to filter out objects or attributes to restrict the synchronization load.

Filter source and target objects with scripts

You can filter the source or target objects that are included in a synchronization operation using the `validSource`, `validTarget`, or `sourceCondition` properties in your mapping:

validSource

A script that determines if a source object is valid to be mapped.

The script yields a boolean value: `true` indicates that the source object is valid; `false` can be used to defer mapping until some condition is met. In the root scope, the source object is provided in the `"source"` property. If the script is not specified, then all source objects are considered valid:

```
{
  "validSource": {
    "type": "text/javascript",
    "source": "source.ldapPassword != null"
  }
}
```

validTarget

A script used during the second phase of reconciliation that determines if a target object is valid to be mapped.

The script yields a boolean value: `true` indicates that the target object is valid; `false` indicates that the target object should not be included in reconciliation. In the root scope, the source object is provided in the `"target"` property. If a `validTarget` script is not specified, then all target objects are considered valid for mapping:

```
{
  "validTarget": {
    "type": "text/javascript",
    "source": "target.employeeType == 'internal'"
  }
}
```

sourceCondition

An additional filter that must be met for a source object to be included in a mapping.

This condition works like a `validSource` script. Its value can be either a `queryFilter` string, or a script configuration. `sourceCondition` is used mainly to specify that a mapping applies only to a particular role or entitlement.

The following `sourceCondition` restricts synchronization to those user objects whose account status is `active`:

```
{
  "mappings": [
    {
```



```

        "name": "managedUser_systemLdapAccounts",
        "source": "managed/user",
        "sourceCondition": "/source/accountStatus eq
\"active\"",
        ...
    }
]
}

```

During synchronization, scripts and filters have access to a `source` object and a `target` object. Examples already shown in this section use `source.attributeName` to retrieve attributes from the source objects. Scripts can also write to target attributes using `target.attributeName` syntax, for example:

```

{
  "onUpdate": {
    "type": "text/javascript",
    "source": "if (source.email != null) {target.mail =
source.email;}"
  }
}

```

The `sourceCondition` filter also has the `linkQualifier` variable in its scope.

For more information about scripting, see [Scripting function reference](#).

Restrict reconciliation by using queries

Every reconciliation operation performs a query on the source and on the target resource, to determine which records should be reconciled. The default source and target queries are `_queryFilter=true&_fields=_id`, which means that all records in both the source and the target are considered candidates for that reconciliation operation.

You can restrict reconciliation to specific entries by defining an explicit `sourceQuery` or `targetQuery` in the mapping configuration.

NOTE

The `sourceQuery` filter is ignored during the target phase, and the `targetQuery` filter is ignored during the source phase.

For example, to restrict reconciliation to those records whose `employeeType` on the source resource is `Permanent`, you might specify a source query as follows:

```

"mappings" : [
  {
    "name" : "managedUser_systemLdapAccounts",
    "source" : "managed/user",
    "target" : "system/ldap/account",
    "sourceQuery" : {
      "_queryFilter" : "employeeType eq \"Permanent\""
    },
    ...
  }
]

```

The format of the query can be any query type that is supported by the resource, and can include additional parameters, if applicable. Use the `_queryFilter` parameter, in common filter notation.

The source and target queries send the query to the resource that is defined for that source or target, by default. You can override the resource the query is sent to by specifying a `resourceName` in the query. For example, to query a specific endpoint instead of the source resource, you might modify the preceding source query as follows:

```

{
  "mappings" : [
    {
      "name" : "managedUser_systemLdapAccounts",
      "source" : "managed/user",
      "target" : "system/ldap/account",
      "sourceQuery" : {
        "resourceName" : "endpoint/scriptedQuery"
        "_queryFilter" : "employeeType eq \"Permanent\""
      },
      ...
    }
  ]
}

```

To override a source or target query that is defined in the mapping, you can specify the query when you call the reconciliation operation. For example, to reconcile all employee entries, and not just the permanent employees, you would run the reconciliation operation as follows:

```

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 '{"sourceQuery": {"_queryFilter" : "true"}}' \

```

```
"http://localhost:8080/openidm/recon?
_action=recon&mapping=managedUser_systemLdapAccounts"
```

By default, a reconciliation operation runs both the source and target phase. To avoid queries on the target resource, set `runTargetPhase` to `false` in the mapping configuration. To prevent the target resource from being queried during the reconciliation operation configured in the previous example, amend the mapping configuration as follows:

```
{
  "mappings" : [
    {
      "name" : "systemLdapAccounts_managedUser",
      "source" : "system/ldap/account",
      "target" : "managed/user",
      "sourceQuery" : {
        "_queryFilter" : "employeeType eq \"Permanent\""
      },
      "runTargetPhase" : false,
      ...
    }
  ]
}
```

Restrict reconciliation queries using the admin UI

1. From the navigation bar, click **Configure > Mappings**.
2. On the **Mappings** page, select the mapping to restrict.
3. Click the **Association** tab, and expand the **Reconciliation Query Filters** node.
4. Create a source or target query, and click **Save**.

Restrict reconciliation to a specific ID

You can restrict reconciliation to a specific record in much the same way as you restrict reconciliation by using queries.

To restrict reconciliation to a specific ID, use the `reconById` action, instead of the `recon` action when you call the reconciliation operation. Specify the ID with the `id` parameter. Reconciling more than one ID with the `reconById` action is not supported.

The following command reconciles only the user with ID `b3c2f414-e7b3-46aa-8ce6-f4ab1e89288c`, for the mapping `managedUser_systemLdapAccounts`. The command synchronizes this particular user account in LDAP with the data from the managed user repository. The example assumes that implicit synchronization has been disabled, and that a reconciliation operation is required to copy changes made in the repository to the LDAP system:

```
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/recon?
_action=reconById&mapping=managedUser_systemLdapAccounts&id=b3c2f4
14-e7b3-46aa-8ce6-f4ab1e89288c"
```

Reconciliation by ID takes the default reconciliation options that are specified in the mapping, so the source and target queries, and source and target phases apply equally to reconciliation by ID.

Restrict implicit synchronization to specific property changes

For a mapping that has managed objects as the source, an implicit synchronization is triggered if *any* source property changes, regardless of whether the modified property is explicitly defined as a `source` property in the mapping.

This default behavior is helpful in situations where no source properties are explicitly defined—any property within the object is included as part of the mapping.

However, this behavior adds a processing overhead, because every mapping from the managed object is invoked when *any* managed object property changes. If several mappings are configured from the managed object, this default behavior can cause performance issues.

In these situations, you can restrict the properties that should trigger an implicit synchronization *per mapping*, using the `triggerSyncProperties` attribute. This attribute contains an array of JSON pointers to the properties that must change before an implicit synchronization to the target is triggered. If none of these properties changes, no synchronization is triggered, even if other properties in the object change.

In the following example, implicit synchronization is triggered *only* if the `mail`, `telephoneNumber`, or `userName` of an object changes:

```
{
  "mappings" : [
    {
      "name" : "managedUser_systemLdapAccounts",
      "source" : "managed/user",
      "target" : "system/ldap/account",
      "enableLinking" : false,
      "triggerSyncProperties" : [
        "/mail",
```

```

        "/telephoneNumber",
        "/userName"
    ],
    "properties" : [],
    "policies" : []
}
]
}

```

If any other property changes on the managed object, no implicit synchronization is triggered.

Implicit synchronization and liveSync

Implicit synchronization and *liveSync* refer to the automatic synchronization of changes from and to the managed object repository.

These topics describe the mechanisms for configuring these automatic synchronization mechanisms.

Disable automatic synchronization operations

By default, all mappings are automatically synchronized. A change to a managed object is automatically synchronized to all resources for which the managed object is configured as a source. If *liveSync* is enabled for a system, changes to an object on that system are automatically propagated to the managed object repository.

To prevent automatic synchronization for a specific mapping, set the `enableSync` property of that mapping to `false`. In the following example, implicit synchronization is disabled. This means that changes to objects in the internal repository are not automatically propagated to the LDAP directory. To propagate changes to the LDAP directory, reconciliation must be launched manually:

```

{
  "mappings" : [
    {
      "name" : "managedUser_systemLdapAccounts",
      "source" : "managed/user",
      "target" : "system/ldap/account",
      "enableSync" : false,
      ...
    }
  ]
}

```

If `enableSync` is set to `false` for a mapping from a system resource to managed/user (for example `"systemLdapAccounts_managedUser"`), `liveSync` is disabled for that mapping.

Configure the liveSync retry policy

If a `liveSync` operation fails, IDM reattempts the change an infinite number of times until the change is successful. This behavior can increase data consistency in the case of transient failures (for example, when the connection to the database is temporarily lost). However, in situations where the cause of the failure is permanent (for example, if the change does not meet certain policy requirements) the change will never succeed, regardless of the number of attempts. In this case, the infinite retry behavior can effectively block subsequent `liveSync` operations from starting.

To avoid this, you can configure a `liveSync` retry policy to specify the number of times a failed modification should be reattempted, and what should happen if the modification is unsuccessful after the specified number of attempts.

Generally, a scheduled reconciliation operation will eventually force consistency. However, to prevent repeated retries that block `liveSync`, restrict the number of times that the same modification is attempted. You can then specify what happens to failed `liveSync` changes. The failed modification can be stored in a *dead letter queue*, discarded, or reapplied. Alternatively, an administrator can be notified of the failure by email or by some other means. This behavior can be scripted. The default configuration in the samples provided with IDM is to retry a failed modification five times, and then to log and ignore the failure.

You configure the `liveSync` retry policy in the [connector configuration](#). The sample connector configurations have a retry policy defined as follows:

```
"syncFailureHandler" : {  
  "maxRetries" : 5,  
  "postRetryAction" : "logged-ignore"  
},
```

maxRetries

Specifies the number of attempts that IDM should make to process the failed modification.

The value of this property must be a positive integer, or `-1`. A value of zero indicates that failed modifications should not be reattempted. In this case, the post-retry action is executed immediately when a `liveSync` operation fails. A value of `-1` (or omitting the `maxRetries` property, or the entire `syncFailureHandler` from the configuration) indicates that failed modifications should be retried an infinite number of times. In this case, no post retry action is executed.

The default retry policy relies on the scheduler, or whatever invokes liveSync. Therefore, if retries are enabled and a liveSync modification fails, IDM will retry the modification the next time that liveSync is invoked.

postRetryAction

Indicates what should happen if the maximum number of retries has been reached (or if `maxRetries` has been set to zero). The post-retry action can be one of the following:

- `logged-ignore`

IDM should ignore the failed modification, and log its occurrence.

- `dead-letter-queue`

IDM should save the details of the failed modification in a table in the repository (accessible over REST at `repo/synchronisation/deadLetterQueue/provisioner-name`).

- `script`

Specifies a custom script that should be executed when the maximum number of retries has been reached. For information about using custom scripts in the configuration, see [Scripting function reference](#). In addition to the regular objects described in that section, the following objects are available in the script scope:

syncFailure

Provides details about the failed record. The structure of the `syncFailure` object is as follows:

```
"syncFailure" :
{
  "token" : the ID of the token,
  "systemIdentifier" : a string identifier that matches
the "name" property in the connector configuration,
  "objectType" : the object type being synced, one of
the keys in the "objectTypes" property in the connector
configuration,
  "uid" : the UID of the object (for example
uid=joe,ou=People,dc=example,dc=com),
  "failedRecord", the record that failed to synchronize
},
```

To access these fields, include `syncFailure.fieldname` in your script.

failureCause

Provides the exception that caused the original liveSync failure.

failureHandlers

Two synchronization failure handlers are provided by default:

- `loggedIgnore` indicates that the failure should be logged, after which no further action should be taken.
- `deadLetterQueue` indicates that the failed record should be written to a specific table in the repository, where further action can be taken.

TIP

To invoke one of the internal failure handlers from your script, use a call similar to the following (shown here for JavaScript):

```
failureHandlers.deadLetterQueue.invoke(syncFailure, failureCause);
```

The following `liveSync` retry policy configuration specifies a maximum of four retries before the failed modification is sent to the dead letter queue:

```
...
  "syncFailureHandler" : {
    "maxRetries" : 4,
    "postRetryAction" : dead-letter-queue
  },
...
```

In the case of a failed modification, a message similar to the following is output to the logs:

```
INFO: sync retries = 1/4, retrying
```

IDM reattempts the modification the specified number of times. If the modification is still unsuccessful, a message similar to the following is logged:

```
INFO: sync retries = 4/4, retries exhausted
Jul 19, 2013 11:59:30 AM

org.forgerock.openidm.provisioner.openicf.syncfailure.DeadLetterQueueHandler invoke
INFO: uid=jdoe,ou=people,dc=example,dc=com saved to dead letter queue
```

The log message indicates the entry for which the modification failed (`uid=jdoe` , in this example).

You can view the failed modification in the dead letter queue, over the REST interface, 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/repo/synchronisation/deadLetterQueue/ldap?_queryFilter=true&_fields=_id"
{
  "result":
  [
    {
      "_id": "4",
      "_rev": "000000001298f6a6"
    }
  ],
  ...
}

```

To view the details of a specific failed modification, include its ID in the URL:

```

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/repo/synchronisation/deadLetterQueue/ldap/4"
{
  "objectType": "account",
  "systemIdentifier": "ldap",
  "failureCause":
"org.forgerock.openidm.sync.SynchronizationException:
    org.forgerock.openidm.objset.ConflictException:
    org.forgerock.openidm.sync.SynchronizationException:
    org.forgerock.openidm.script.ScriptException:
    ReferenceError: \"bad\" is not defined.

(PropertyMapping/mappings/0/properties/3/condition#1)",
  "token": 4,
  "failedRecord": "complete record, in xml format"
  "uid": "uid=jdoe,ou=people,dc=example,dc=com",
  "_rev": "000000001298f6a6",
  "_id": "4"
}

```

NOTE

NOTE

The `repo` endpoint is an *internal interface*. Although it is used in the preceding example for the purposes of demonstration, you should not rely on this endpoint in production.

Improve reliability with queued synchronization

By default, IDM implicitly synchronizes managed object changes out to all resources for which the managed object is configured as a source. If there are several targets that must be synchronized, these targets are synchronized one at a time, one after the other. If any of the targets is remote or has a high latency, the implicit synchronization operations can take some time, delaying the successful return of the managed object change.

To decouple the managed object changes from the corresponding synchronizations, you can configure *queued synchronization*, which persists implicit synchronization events to the IDM repository. Queued events are then read from the repository and executed according to the queued synchronization configuration.

Because synchronization operations are performed in parallel, queued synchronization can improve performance if you have several fast, reliable targets. However, queued synchronization is also useful when your targets are slow or unreliable, because the managed object changes can complete before all targets have been synchronized.

The following illustration shows how synchronization operations are added to a local, in-memory queue. Note that this queue is distinct from the repository queue for synchronization events:

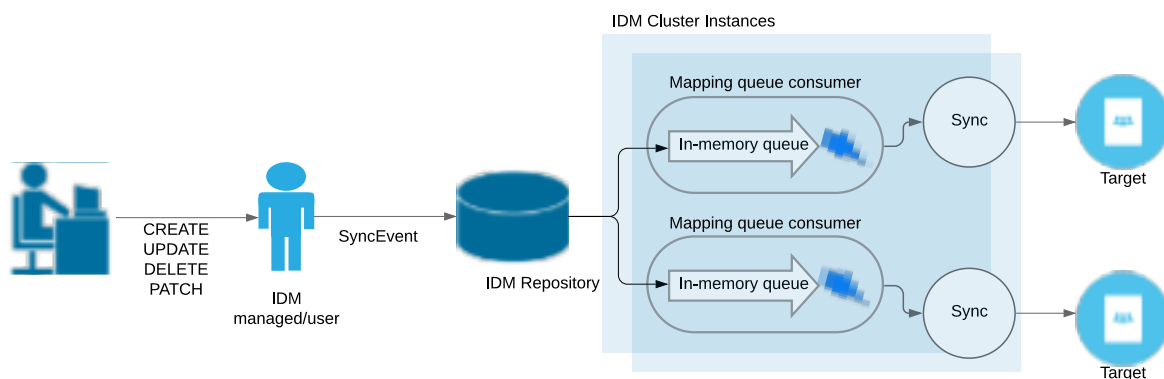


Figure 5. Queued Synchronization

Configure queued synchronization

Queued synchronization is disabled by default. To enable it, add a `queuedSync` object to your mapping, as follows:

```

{
  "mappings" : [
    {
      "name" : "managedUser_systemLdapAccounts",
      "source" : "managed/user",
      "target" : "system/ldap/account",
      "links" : "systemLdapAccounts_managedUser",
      "queuedSync" : {
        "enabled" : true,
        "pageSize" : 100,
        "pollingInterval" : 1000,
        "maxQueueSize" : 20000,
        "maxRetries" : 5,
        "retryDelay" : 1000,
        "postRetryAction" : "logged-ignore"
      },
      ...
    }
  ]
}

```

NOTE

- These settings apply *only* to the implicit synchronization operations for that mapping. Reconciliation is unaffected by queued synchronization settings. Events associated with mappings where queued synchronization is enabled are submitted to the synchronization queue for asynchronous processing. Events associated with mappings where queued synchronization is not enabled are processed immediately, and block further event processing until they are complete.
- During implicit synchronization, mappings are processed in the order in which they are defined, regardless of whether queued synchronization is enabled for those mappings. If you want all queued synchronization mappings to be processed first, you must explicitly order your mappings accordingly.
- Processing the synchronization queue for a mapping is *paused* if either the source or target system route is *unregistered*. A route is unregistered when you remove the [\\$connector configuration](#), set "enabled" : false in the [\\$connector configuration](#), delete the mapping, or remove the managed object type from the [\\$managed object configuration](#).

The `queuedSync` object has the following configuration:

enabled

Specifies whether queued synchronization is enabled for that mapping. Boolean, true, or false.

pageSize (integer)

Specifies the maximum number of events to retrieve from the repository queue within a single polling interval. The default is 100 events.

pollingInterval (integer)

Specifies the repository queue polling interval, in milliseconds. The default is 1000 ms.

maxQueueSize (integer)

Specifies the maximum number of synchronization events that can be accepted into the in-memory queue. The default is 20000 events.

maxRetries (integer)

The number of retries to perform before invoking the `postRetry` action. Most sample configurations set the maximum number of retries to 5. To set an infinite number of retries, either omit the `maxRetries` property, or set it to a negative value, such as -1.

retryDelay (integer)

In the event of a failed queued synchronization operation, this parameter specifies the number of milliseconds to delay before attempting the operation again. The default is 1000 ms.

postRetryAction

The action to perform after the retries have been exhausted. Possible options are `logged-ignore`, `dead-letter-queue`, and `script`. These options are described in [Configure the LiveSync Retry Policy](#). The default action is `logged-ignore`.

NOTE

Retries occur synchronously to the failure. For example, if the `maxRetries` is set to 10, at least 10 seconds will pass between the failing sync event and the next sync. (There are 10 retries, and the `retryDelay` is 1 second by default.) These 10 seconds do not take into account the latency of the ten sync requests. Retries are configured per-mapping and block processing of all subsequent sync events until the configured retries have been exhausted.

Tune queued synchronization

Queued synchronization employs a single worker thread. While implicit synchronization operations are being generated, that worker thread should always be occupied. The occupation of the worker thread is a function of the `pageSize`, the `pollingInterval`, the latency of the poll request, and the latency of each synchronization operation for the mapping.

For example, assume that a poll takes 500 milliseconds to complete. Your system must provide operations to the worker thread at approximately the same rate at which the thread can consume events (based on the page size, poll frequency, and poll latency). Operation consumption is a function of the `notifyaction.execution` for that particular mapping. If the system does not provide operations fast enough, implicit synchronization will not occur as optimally as it could. If the system provides operations too quickly, the operations in the queue could exceed the default maximum of 20000. If the `maxQueueSize` is reached, additional synchronization events will result in a `RejectedExecutionException`.

Depending on your hardware and workload, you might need to adjust the default `pageSize`, `pollingInterval`, and `maxQueueSize`.

Monitor the queued synchronization metrics; specifically, the `rejected-executions`, and adjust the `maxQueueSize` accordingly. Set a large enough `maxQueueSize` to prevent slow mappings and heavy loads from causing newly-submitted synchronization events to be rejected.

Monitor the synchronization latency using the `sync.queue.mapping-name.poll-pending-events` metric.

For more information on monitoring metrics, see [Metrics reference](#).

Manage the synchronization queue

You can manage queued synchronization events over the REST interface, at the `openidm/sync/queue` endpoint. The following examples show the operations that are supported on this endpoint:

List all events in the synchronization queue:

```
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/sync/queue?_queryFilter=true"
{
  "result": [
    {
      "_id": "03e6ab3b-9e5f-43ac-a7a7-a889c5556955",
      "_rev": "0000000034dba395",
      "mapping": "managedUser_systemLdapAccounts",
      "resourceId": "e6533cfe-81ad-4fe8-8104-55e17bd9a1a9",
      "syncAction": "notifyCreate",
      "state": "PENDING",
```

```

    "resourceCollection": "managed/user",
    "nodeId": null,
    "createDate": "2018-11-12T07:45:00.072Z"
  },
  {
    "_id": "ed940f4b-ce80-4a7f-9690-1ad33ad309e6",
    "_rev": "000000007878a376",
    "mapping": "managedUser_systemLdapAccounts",
    "resourceId": "28b1bd90-f647-4ba9-8722-b51319f68613",
    "syncAction": "notifyCreate",
    "state": "PENDING",
    "resourceCollection": "managed/user",
    "nodeId": null,
    "createDate": "2018-11-12T07:45:00.150Z"
  },
  {
    "_id": "f5af2eed-d83f-4b70-8001-8bc86075134f",
    "_rev": "00000000099aa321",
    "mapping": "managedUser_systemLdapAccounts",
    "resourceId": "d2691a45-0a10-4f51-aa2a-b6854b2f8086",
    "syncAction": "notifyCreate",
    "state": "PENDING",
    "resourceCollection": "managed/user",
    "nodeId": null,
    "createDate": "2018-11-12T07:45:00.276Z"
  },
  ...
],
"resultCount": 8,
"pagedResultsCookie": null,
"totalPagedResultsPolicy": "NONE",
"totalPagedResults": -1,
"remainingPagedResults": -1
}

```

Query the queued synchronization events based on the following properties:

- mapping —the mapping associated with this event. 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/sync/queue?
_queryFilter=mapping+eq+'managedUser_systemLdapAccount' "
```

- `nodeId` —the ID of the node that has acquired this event.
- `resourceId` —the source object resource ID.
- `resourceCollection` —the source object resource collection.
- `_id` —the ID of this sync event.
- `state` —the state of the synchronization event. 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/sync/queue?
_queryFilter=state+eq+'PENDING' "
```

The `state` of a queued synchronization event is one of the following:

`PENDING` —the event is waiting to be processed.

`ACQUIRED` —the event is being processed by a node.

- `remainingRetries` —the number of retries available for this synchronization event before it is abandoned. For more information about how synchronization events are retried, see [Configure the LiveSync Retry Policy](#). 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/sync/queue?
_queryFilter=remainingRetries+lt+2"
```

- `syncAction` —the synchronization action that initiated this event. Possible synchronization actions are `notifyCreate`, `notifyUpdate`, and `notifyDelete`. 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/sync/queue?
_queryFilter=syncAction+eq+'notifyCreate' "
```

- `createDate` —the date that the event was created.

Recover mappings when nodes are down

Synchronization events for mappings with queued synchronization enabled are processed by a single cluster node. While a node is present in the cluster, that node holds a *lock* on the specific mapping. The node can release or reacquire the mapping lock if a balancing event occurs (see [Balance Mapping Locks Across Nodes](#)). However, the mapping lock is held across all events on that mapping. In a stable running cluster, a single node will hold the lock for a mapping indefinitely.

It is possible that a node goes down, or is removed from the cluster, while holding a mapping lock on operations in the synchronization queue. To prevent these operations from being lost, the queued synchronization facility includes a *recovery monitor* that checks for any *orphaned* mappings in the cluster.

A mapping is considered orphaned in the following cases:

- No active node holds a lock on the mapping.
- The node that holds a lock on the mapping has an instance state of `STATE_DOWN`.
- The node that holds a lock on the mapping does not exist in the cluster.

The recovery monitor periodically checks for orphaned mappings. When all orphaned mappings have been recovered, it attempts to initialize new queue consumers.

The recovery monitor is enabled by default and executes every 300 seconds. To change the default behavior for a mapping, add the following to the mapping configuration and change the parameters as required:

```
{
  "mappings" : [...],
  "queueRecovery" : {
    "enabled" : true,
    "recoveryInterval" : 300
  }
}
```


IMPORTANT

If a queued synchronization job has already been claimed by a node, and that node is *shut down*, IDM notifies the entire cluster of the shutdown. This lets a different node pick up the job in progress. The recovery monitor takes over jobs in a synchronization queue that have not been fully processed by an available cluster node, so no job should be lost. If you have configured queued synchronization for one or more mappings, do not use the `enabled` flag in the cluster configuration to remove a node from the cluster. Instead, shut down the node so that the remaining nodes in the cluster can take over the queued synchronization jobs.

Balance mapping locks across nodes

Queued synchronization mapping locks are balanced equitably across cluster nodes. At a specified interval, each node attempts to release and acquire mapping locks, based on the number of running cluster nodes. When new cluster nodes come online, existing nodes release sufficient mapping locks for new nodes to pick them up, resulting in an equitable distribution of locks.

Lock balancing is enabled by default, and the interval at which nodes attempt to balance locks in the queue is 5 seconds. To change the default configuration, add a `queueBalancing` object to your `mapping` and set the following parameters:

```
{
  "mappings" : [...],
  "queueBalancing" : {
    "enabled" : true,
    "balanceInterval" : 5
  }
}
```

Synchronization failure compensation

If implicit synchronization fails for a target resource (for example, due to a policy validation failure on the target, or the target being unavailable), the synchronization operation stops at that point. In this scenario, a record might be changed in the repository, and in the targets on which synchronization was successful, but not on the failed target, or on any targets that would have been synchronized *after* the failure. This can result in disparate data sets across resources. Although a reconciliation operation would eventually bring all targets back in sync, reconciliation can be an expensive operation with large data sets.

You can configure *synchronization failure compensation* to prevent data sets from becoming out of sync. This mechanism involves reverting an implicit synchronization operation if it is not completely successful across all configured mappings.

Failure compensation ensures that either all resources are synchronized successfully, or that the original change is rolled back. This mechanism uses an `onSync` script hook in the [managed object configuration](#). The `onSync` hook calls a script that prevents partial synchronization by "reverting" a partial change in the event that all resources are not synchronized.

The following sample [managed object configuration](#) shows the addition of the `onSync` hook:

```
...
"onDelete" : {
  "type" : "text/javascript",
  "file" : "onDelete-user-cleanup.js"
},
"onSync" : {
  "type" : "text/javascript",
  "file" : "compensate.js"
},
"properties" : [
  ...
```

With this configuration, a change to a managed object triggers an implicit synchronization for each configured [mapping](#), in the order in which the mappings are defined. If synchronization is successful for all configured mappings, IDM exits from the script. If synchronization fails for a particular resource, the `onSync` hook invokes the `compensate.js` script, which attempts to revert the original change by performing another update to the managed object. This change, in turn, triggers another implicit synchronization operation to all external resources for which mappings are configured.

If the synchronization operation fails again, the `compensate.js` script is triggered a second time. This time, however, the script recognizes that the change was originally called as a result of a compensation and aborts. IDM logs warning messages related to the sync action (`notifyCreate`, `notifyUpdate`, `notifyDelete`), along with the error that caused the sync failure.

If failure compensation is not configured, any issues with connections to an external resource can result in out of sync data stores.

With the `compensate.js` script, any such errors will result in each data store retaining the information it had before implicit synchronization started. That information is stored, temporarily, in the `oldObject` variable.

Schedule synchronization

You can schedule synchronization operations, such as liveSync and reconciliation, using Quartz triggers. IDM supports simple triggers and cron triggers.

Use the trigger type that suits your scheduling requirements. Because simple triggers are not bound to the local timezone, they are better suited to scenarios such as liveSync, where the requirement is to trigger the schedule at regular intervals, regardless of the local time. For more information, see the Quartz documentation on [SimpleTriggers](#) and [CronTriggers](#).

This section describes scheduling specifically for reconciliation and liveSync, and shows simple triggers in all the examples. You can use the scheduler service to schedule any other event by supplying a script in which that event is defined. For information about scheduling other events, see [Schedule tasks and events](#).

Configure scheduled synchronization

Each scheduled reconciliation and liveSync task requires a [schedule configuration](#), with the following format:

```
{
  "enabled"      : boolean, true/false
  "type"         : "string",
  "repeatInterval" : long integer,
  "repeatCount"  : integer,
  "persisted"    : boolean, true/false
  "startTime"    : "(optional) time",
  "endTime"      : "(optional) time",
  "schedule"     : "cron expression",
  "misfirePolicy" : "optional, string",
  "invokeService" : "service identifier",
  "invokeContext" : "service specific context info"
}
```

These properties are specific to the scheduler service, and are explained in [Schedule tasks and events](#).

To schedule a reconciliation or liveSync task, set the `invokeService` property to either `sync` (for reconciliation) or `provisioner` for liveSync.

The value of the `invokeContext` property depends on the type of scheduled event. For reconciliation, the properties are set as follows:

```
{
  "invokeService": "sync",
  "invokeContext": {
```

```
    "action": "reconcile",
    "mapping": "systemLdapAccount_managedUser"
  }
}
```

The `mapping` is referenced by its name in the [mapping configuration](#).

For `liveSync`, the properties are set as follows:

```
{
  "invokeService": "provisioner",
  "invokeContext": {
    "action": "liveSync",
    "source": "system/ldap/account"
  }
}
```

The `source` property follows the convention for a pointer to an external resource object, and takes the form `system/resource-name/object-type`.

IMPORTANT

When you schedule a reconciliation operation to run at regular intervals, do not set `"concurrentExecution" : true`. This parameter enables multiple scheduled operations to run concurrently. You cannot launch multiple reconciliation operations for a single mapping concurrently.

Schedule `liveSync` using the admin UI

To configure `liveSync` using the admin UI, set up a `liveSync` schedule:

1. From the navigation bar, click **Configure > Schedules**, and then click **Add Schedule**.
2. Complete the schedule configuration, and click **Save**.

For more information about these fields, see [Configure Scheduled Synchronization](#).

NOTE

The scheduler configuration assumes a `simple` trigger type by default, so the `Cron-like Trigger` field is disabled. You should use simple triggers for liveSync schedules to avoid problems related to daylight savings time. For more information, see [Schedules and daylight savings time](#).

By default, the admin UI creates schedules using the scheduler service, rather than the configuration service. To create this schedule in the configuration service, select the **Save as Config Object** option.

If your deployment enables writes to configuration files, this option also creates a corresponding `schedule-schedule-name.json` file in your project's `conf` directory.

For more information on the distinction between the scheduler service and the configuration service, see [Configure the scheduler service](#).

Clustered reconciliation

In a clustered deployment, you can configure reconciliation jobs to be distributed across multiple nodes in the cluster. Clustered reconciliation is configured *per mapping* and can improve reconciliation performance, particularly for very large data sets.

Clustered reconciliation uses the paged reconciliation mechanism and the scheduler service to divide the *source* data set into pages, and then to schedule reconciliation "sub-jobs" per page, distributing these sub-jobs across the nodes in the cluster.

Regular (non-clustered) reconciliation has two phases—a source phase and a target phase. Clustered reconciliation effectively has three phases:

Source page phase

During this phase, reconciliation sub-jobs are scheduled in succession, page by page. Each source page job does the following:

- Executes a source query using the paging cookie from the invocation context.
- Schedules the next source page job.
- Performs the reconciliation of the source IDs returned by the query.
- Writes statistics summary information which is aggregated so that you can obtain the status of the complete reconciliation run by performing a GET on the `recon` endpoint.
- On completion, writes the `repo_id`, `source_id`, and `target_id` to the repository.

Source phase completion check

This phase is scheduled when the source query returns null. This check runs, and continues to reschedule itself, as long as source page jobs are running. When the completion check determines that all the source page jobs are complete, it schedules the target phase.

Target phase

This phase queries the target IDs, then removes all of the IDs that correspond to the `repo_id`, `source_id`, and `target_id` written by the source pages. The remaining target IDs are used to run the target phase, taking into account all records on the target system that were not correlated to a source ID during the source phase sub-jobs.

Configure clustered reconciliation for a mapping

To specify that the reconciliation for a specific mapping should be distributed across a cluster, add the `clusteredSourceReconEnabled` property to the mapping and set it to `true`. For example:

```
{
  "mappings" : [
    {
      "name" : "systemLdapAccounts_managedUser",
      "source" : "system/ldap/account",
      "target" : "managed/user",
      "clusteredSourceReconEnabled" : true,
      ...
    }
  ]
}
```

NOTE

When clustered reconciliation is enabled, source query paging is enabled automatically, regardless of the value that you set for the `reconSourceQueryPaging` property in the mapping.

By default, the number of records per page is **10000**. You can also enable *target query paging* with the `reconTargetQueryPaging` property (defaults to `false`). To change the query page sizes, set the `reconSourceQueryPageSize` and `reconTargetQueryPageSize` properties.

The following example enables *target query paging* and changes the target and source query page sizes:

```
{
  "mappings" : [
    {
```

```

    "name" : "systemLdapAccounts_managedUser",
    "source" : "system/ldap/account",
    "target" : "managed/user",
    "clusteredSourceReconEnabled" : true,
    "reconTargetQueryPaging" : true,
    "reconSourceQueryPageSize" : 12000,
    "reconTargetQueryPageSize" : 12000,
    ...
}

```

To set these properties using the admin UI, click **Configure > Mappings**, select the mapping to change, and click the **Advanced** tab.

Clustered reconciliation has the following limitations:

- A complete non-clustered reconciliation run is synchronous with the single reconciliation invocation.

By contrast, a clustered reconciliation is *asynchronous*. In a clustered reconciliation, the first execution is synchronous only with the reconciliation of the first page. This job also schedules the subsequent pages of the clustered reconciliation to run on other cluster nodes. When you schedule a clustered reconciliation or call the operation over REST, do not set `waitForCompletion` to `true`, because you cannot wait for the operation to complete before the next operation starts.

Because this first execution does not encompass the entire reconciliation operation for that mapping, you cannot rely on the Quartz `concurrentExecution` property to prevent two reconciliation operations from running concurrently. If you use Quartz to schedule clustered reconciliations (as described in [Configure Scheduled Synchronization](#)), make sure that the interval between scheduled operations exceeds the known run of the entire clustered reconciliation. The run-length of a specific clustered reconciliation can vary. You must therefore build in appropriate buffer times between schedules, or use a scheduled script that performs a GET on the `recon/` endpoint, and dispatches the next reconciliation on a mapping only when the previous reconciliation run has completed.

- Clustered reconciliations can recover missing source pages (for example, if a cluster goes offline during a clustered reconciliation run), except when used with a connector using server-side logic to handle paging that returns a static paging cookie.

Clustered reconciliation progress

The `sourceProcessedByNode` property indicates how many records are processed by each node. You can verify the load distribution per node by running a GET on the `recon` endpoint, 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/recon"
...
  "started": "2017-05-11T10:04:59.563Z",
  "ended": "",
  "duration": 342237,
  "sourceProcessedByNode": {
    "node2": 21500,
    "node1": 22000
  }
}

```

You can also display the nodes responsible for each source page in the admin UI. Click on the relevant mapping and expand the In Progress or Reconciliation Results item. The following image shows a clustered reconciliation in progress. The details include the number of records that have been processed, the current duration of the reconciliation, and the load distribution, per node:

The screenshot shows the 'MAPPING DETAIL' page for 'systemLdapAccounts_managedUser'. It features a 'Cancel Reconciliation' button and a 'Help' icon. The source is 'system/ldap/account ldap' and the target is 'managed/user managed'. The reconciliation is 'In Progress: reconciling page of source entries - 15500/15500'. The 'Reconciliation Results' section shows 15500 succeeded and 0 failed. The 'Duration' is 00:01:36:126. The 'Records Processed by Node' table is as follows:

Node	Records Processed
node2	6500
node1	9000

Figure 6. Clustered Reconciliation Results

Cancel a clustered reconciliation

You cancel a clustered reconciliation in the same way as a non-clustered reconciliation, 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/recon/90892122-5ceb-4bbe-86f7-94272df834ad-406025?_action=cancel"
{
  "_id": "90892122-5ceb-4bbe-86f7-94272df834ad-406025",
  "action": "cancel",
  "status": "INITIATED"
}
```

When the cancellation has completed, a query on that reconciliation ID will show the state and stage of the reconciliation as follows:

```
{
  "_id": "90892122-5ceb-4bbe-86f7-94272df834ad-406025",
  "mapping": "systemLdapAccounts_managedUser",
  "state": "CANCELED",
  "stage": "COMPLETED_CANCELED",
  "stageDescription": "reconciliation aborted.",
  "progress": {
    "source": {
      "existing": {
        "processed": 23500,
        "total": "23500"
      }
    },
    "target": {
      "existing": {
        "processed": 23498,
        "total": "?"
      }
    },
    ...
  }
}
```

In a clustered environment, *all* reconciliation operations are considered to be "cluster-friendly". This means that even if a mapping is configured as "clusteredSourceReconEnabled" : false , you can view the in progress operation on *any* node in the cluster, even if that node is not currently processing the

reconciliation. You can also cancel a reconciliation in progress from any node in the cluster.

Tuning reconciliation performance

By default, reconciliation is configured to perform optimally. In some cases, the default optimizations might not be suitable for your deployment. The following sections describe these default optimizations, how they can be configured, and additional methods you can use to improve reconciliation performance.

Correlate empty target sets

To optimize performance, reconciliation does not correlate source objects to target objects if the set of target objects is empty when the correlation is started. This considerably speeds up the process the first time reconciliation is run. You can change this behavior for a specific mapping by adding the `correlateEmptyTargetSet` property to the mapping definition and setting it to `true`. For example:

```
{
  "mappings": [
    {
      "name": "systemMyLDAPAccounts_managedUser",
      "source": "system/MyLDAP/account",
      "target": "managed/user",
      "correlateEmptyTargetSet": true
    },
  ]
}
```

Be aware that this setting will have a performance impact on the reconciliation process.

Prefetch links

All links are queried at the start of reconciliation and the results of that query are used. You can disable the link prefetching so that the reconciliation process looks up each link in the database as it processes each source or target object. To disable link prefetching, add the `prefetchLinks` property to your mapping, and set it to `false`:

```
{
  "mappings": [
    {
```

```

        "name": "systemMyLDAPAccounts_managedUser",
        "source": "system/MyLDAP/account",
        "target": "managed/user"
        "prefetchLinks" : false
    }
]
}

```

Be aware that this setting will have a performance impact on the reconciliation process.

Parallel reconciliation threads

By default, reconciliation is multithreaded; numerous threads are dedicated to the same reconciliation run. Multithreading generally improves reconciliation performance. The default number of threads for a single reconciliation run is 10 (plus the main reconciliation thread). Under normal circumstances, you should not need to change this number. However the default might not be appropriate in the following situations:

- The hardware has many cores and supports more concurrent threads. As a rule of thumb for performance tuning, start with setting the thread number to twice the number of cores.
- The source or target is an external system with high latency or slow response times. Threads may then spend considerable time waiting for a response from the external system. Increasing the available threads enables the system to prepare or continue with additional objects.

To change the number of threads, set the `taskThreads` property in your mapping:

```

"mappings" : [
  {
    "name" : "systemCsvfileAccounts_managedUser",
    "source" : "system/csvfile/account",
    "target" : "managed/user",
    "taskThreads" : 20
    ...
  }
]

```

A zero value runs reconciliation as a serialized process, on the main reconciliation thread.

Improve reconciliation query performance

Reconciliation operations are processed in two phases; a *source phase* and a *target phase*. In most reconciliation configurations, source and target queries make a read call to every record on the source and target systems to determine candidates for reconciliation. On slow source or target systems, these frequent calls can incur a substantial performance cost.

To improve query performance in these situations, you can preload the entire result set into memory on the source or target system, or on both systems. Subsequent read queries on known IDs are made against the data in memory, rather than the data on the remote system. For this optimization to be effective, the entire result set must fit into the available memory on the system for which it is enabled.

The optimization works by defining a `sourceQuery` or `targetQuery` in the synchronization mapping that returns not just the ID, but the complete object.

The following example query loads the full result set into memory during the source phase of the reconciliation. The example uses a common filter expression, called with the `_queryFilter` keyword. The query returns the complete object:

```
"mappings" : [  
  {  
    "name" : "systemLdapAccounts_managedUser",  
    "source" : "system/ldap/account",  
    "target" : "managed/user",  
    "sourceQuery" : {  
      "_queryFilter" : "true"  
    },  
    ...  
  }  
]
```

IDM attempts to detect what data has been returned. The autodetection mechanism assumes that a result set that includes three or more fields per object (apart from the `_id` and `rev` fields) contains the complete object.

You can explicitly state whether a query is configured to return complete objects by setting the value of `sourceQueryFullEntry` or `targetQueryFullEntry` in the mapping. The setting of these properties overrides the autodetection mechanism.

Setting these properties to `false` indicates that the returned object is not the complete object. This might be required if a query returns more than three fields of an object, but not the complete object. Without this setting, the autodetect logic would assume that the complete object was being returned. IDM uses only the IDs from this query result. If the complete object is required, the object is queried on demand.

Setting these properties to `true` indicates that the complete object is returned. This setting is typically required only for very small objects, for which the number of returned fields does not reach the threshold required for the auto-detection mechanism to

assume that it is a full object. In this case, the query result includes all the details required to pre-load the full object.

The following excerpt indicates that the full objects are returned and that IDM should not autodetect the result set:

```
"mappings" : [  
  {  
    "name" : "systemLdapAccounts_managedUser",  
    "source" : "system/ldap/account",  
    "target" : "managed/user",  
    "sourceQueryFullEntry" : true,  
    "sourceQuery" : {  
      "_queryFilter" : "true"  
    },  
    ...  
  }  
]
```

By default, all the attributes defined in the [connector configuration](#) are loaded into memory. If your mapping uses only a small subset of the attributes in the connector configuration, you can restrict your query to return only those attributes required for synchronization by using the `_fields` parameter with the query filter.

The following excerpt loads only a subset of attributes into memory, for all users in an LDAP directory.

```
"mappings" : [  
  {  
    "name" : "systemLdapAccounts_managedUser",  
    "source" : "system/ldap/account",  
    "target" : "managed/user",  
    "sourceQuery" : {  
      "_queryFilter" : "true",  
      "_fields" : "cn,sn,dn,uid,employeeType,mail"  
    },  
    ...  
  }  
]
```

NOTE

The default source query for clustered reconciliations and for paged reconciliations is a `queryFilter` that returns the full source objects, not just their IDs. So, source queries for clustered and paged reconciliations are optimized for performance by default.

Paging reconciliation query results

Improve Reconciliation Query Performance describes how to improve reconciliation performance by loading all entries into memory to avoid making individual requests to the external system for every ID. However, this optimization depends on the entire result set fitting into the available memory on the system for which it is enabled. For particularly large data sets (for example, data sets of hundreds of millions of users), having the entire data set in memory might not be feasible.

To alleviate this constraint, you can use reconciliation paging, which breaks down extremely large data sets into chunks. It also lets you specify the number of entries that should be reconciled in each chunk or page.

Reconciliation paging is disabled by default, and can be enabled per mapping. To configure reconciliation paging, set the `reconSourceQueryPaging` property to `true` and set the `reconSourceQueryPageSize` in the synchronization mapping:

```
{
  "mappings" : [
    {
      "name" : "systemLdapAccounts_managedUser",
      "source" : "system/ldap/account",
      "target" : "managed/user",
      "reconSourceQueryPaging" : true,
      "reconSourceQueryPageSize" : 100,
      ...
    }
  ]
}
```

The value of `reconSourceQueryPageSize` must be a positive integer, and specifies the number of entries that will be processed in each page. If reconciliation paging is enabled but no page size is set, a default page size of 1000 is used.

If you are reconciling from a JDBC database using the Database Table connector, you *must* set the `_sortkeys` property in the source query and ensure that the corresponding column is indexed in the database.

The following excerpt of a mapping configures paged reconciliation queries using the Database Table connector:

```
{
  "mappings" : [
    {
      "name" : "systemHrdb_managedUser",
      "source" : "system/db/users",
      "target" : "managed/user",
      "reconSourceQueryPaging" : true,
      "reconSourceQueryPageSize" : 1000,
      "sourceQueryFullEntry" : true,
      "sourceQuery" : {
        "_queryFilter" : "true",
        "_sortKeys" : "email"
      },
      ...
    }
  ]
}
```

Asynchronous reconciliation

Reconciliation can work in tandem with workflows to provide additional business logic to the reconciliation process. You can define scripts to determine the action that should be taken for a particular reconciliation situation. A reconciliation process can launch a workflow after it has assessed a situation, and then perform the reconciliation or some other action.

For example, you might want a reconciliation process to assess new user accounts that need to be created on a target resource. However, new user account creation might require some kind of approval from a manager before the accounts are actually created. The initial reconciliation process can assess the accounts that need to be created, then launch a workflow to request management approval for those accounts. The workflow performs the sync action, based upon the situation assessed during reconciliation (and provided to the workflow through the `ASYNC` action). The workflow then calls the `sync` endpoint with the `performAction` action and triggers a synchronization operation for the specified object.

In this scenario, the defined script returns `ASYNC` for new accounts, and the reconciliation engine does not continue processing the given object. The script then

initiates an asynchronous process which, on completion, performs an explicit sync of the source object.

A sample configuration for this scenario is available in `openidm/samples/sync-asynchronous`, and described in [Asynchronous reconciliation using workflow](#).

Configure asynchronous reconciliation Using a workflow

1. Create the workflow definition file (`.xml` or `.bar` file) and place it in the `openidm/workflow` directory. For more information about creating workflows, see [Create workflows](#).
2. Modify the mapping for the situation or situations that should call the workflow. Reference the workflow name in the configuration for that situation.

For example, the following mapping excerpt calls the `managedUserApproval` workflow if the situation is assessed as `ABSENT` :

```
{
  "situation" : "ABSENT",
  "action" : {
    "workflowName" : "managedUserApproval",
    "type" : "text/javascript",
    "file" : "workflow/triggerWorkflowFromSync.js"
  }
}
```

In the sample configuration, the workflow makes an explicit call to the `sync` endpoint with the `performAction` action (`openidm.action('sync', 'performAction', content, params)`).

You can also use this kind of explicit synchronization to perform a specific action on a source or target record, regardless of the assessed situation.

To call such an operation over the REST interface, specify the source, and/or target IDs, the mapping, and the action to be taken. The action can be any one of the supported reconciliation actions: `CREATE`, `UPDATE`, `DELETE`, `LINK`, `UNLINK`, `EXCEPTION`, `REPORT`, `NOREPORT`, `ASYNC`, `IGNORE`.

The following example calls the `DELETE` action on user `bjensen`, whose `_id` in the LDAP directory is `uid=bjensen,ou=People,dc=example,dc=com`. The user is deleted in the target resource; in this case, the repository.

NOTE

The `_id` must be URL-encoded in the REST call:

```
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/sync?
_action=performAction&sourceId=uid%3Dbjensen%2Cou%3DPeople%2Cdc%3Dexample%
2Cdc%3Dcom&mapping=
systemLdapAccounts_ManagedUser&action=DELETE"
{
  "status": "OK"
}
```

The following example creates a link between a managed object and its corresponding system object. Such a call is useful in the context of manual data association, when correlation logic has linked an incorrect object, or when IDM has been unable to determine the correct target object.

In this example, there are two separate target accounts (`scarter.user` and `scarter.admin`) that should be mapped to the managed object. This call creates a link to the `user` account and specifies a link qualifier that indicates the type of link that will be created:

```
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/sync?
_action=performAction&action=LINK
&sourceId=4b39f74d-92c1-4346-9322-
d86cb2d828a8&targetId=scarter.user
&mapping=managedUser_systemCsvfileAccounts&linkQualifier=user"
{
  "status": "OK"
}
```

For more information about linking to multiple accounts, see [Map a Single Source Object to Multiple Target Objects](#).

Import bulk data

The bulk import service 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

The bulk import service assumes 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.

The bulk import service assumes a singular `type`. If you submit an array of `type` attributes, the service sets the `type` as the last element of the array.

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/p
lain'"
{
  "_id": "template",
  "header":
  "\"userName\", \"givenName\", \"sn\", \"mail\", \"description\", \"ac
countStatus\", \"telephoneNumber\",

  \"postalAddress\", \"city\", \"postalCode\", \"country\", \"statePro
```

```
vince\", \"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](#)

You can use the bulk import service with a CSV file up to 50MBytes large and less than 100,000 records. If you need to import a larger file or more records, divide your data into chunks and import each file separately.

You can increase the maximum file size by changing the value of the `maxRequestSizeInMegabytes` property in your `conf/servletfilter-upload.json` file.

You need to use a CSV template to perform a bulk import. For more information, refer to [Generate a CSV template](#).

After formatting your CSV file to match your template's structure, 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": "000000003e8ef4f7",
    "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"  
}
```

▼ [Change the 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, refer to [Bulk import](#).

Synchronization reference

The synchronization engine is one of the core IDM services. You configure the synchronization service through a `mappings` property that specifies mappings between objects that are managed by the synchronization engine.

```
{  
  "mappings": [ object-mapping object, ... ]  
}
```

Object-mapping objects

An object-mapping object specifies the configuration for a mapping of source objects to target objects. The `name`, `source`, and `target` properties are mandatory. Other properties are optional or implicit (that is, they have a default value if not set).

```
{  
  "correlationQuery"           : script object,  
  "correlationScript"         : script object,
```



```

    "displayName"                : string,
    "enableLinking"              : boolean,
    "enableSync"                 : boolean,
    "linkQualifiers"             : [ list of strings ] or
script object,
    "links"                      : string,
    "name"                       : string,
    "onCreate"                   : script object,
    "onDelete"                   : script object,
    "onLink"                     : script object,
    "onMapping"                  : script object,
    "onUnlink"                   : script object,
    "onUpdate"                   : script object,
    "optimizeAssignmentSync"     : boolean,
    "policies"                   : [ policy object, ... ],
    "postMapping"                : script object,
    "properties"                 : [ property object, ... ],
    "queuedSync"                 : { property object },
    "reconAssociation"           : { property object },
    "reconProgressStateUpdateInterval" : integer,
    "reconSourceQueryPageSize"   : integer,
    "reconSourceQueryPaging"     : boolean,
    "reconTargetQueryPageSize"   : integer,
    "reconTargetQueryPaging"     : boolean,
    "result"                     : script object,
    "runTargetPhase"             : boolean,
    "source"                     : string,
    "sourceCondition"            : script object or
queryFilter string,
    "sourceIdsCaseSensitive"     : boolean,
    "sourceQueryFullEntry"       : boolean,
    "syncAfter"                  : [ list of strings ],
    "target"                     : string,
    "targetIdsCaseSensitive"     : boolean,
    "targetQueryFullEntry"       : boolean,
    "taskThreads"                : integer,
    "triggerSyncProperties"      : [ list of JSON pointers ],
    "validSource"                : script object,
    "validTarget"                : script object
}

```

Mapping object properties

correlationQuery

script object, optional

A script that yields a query object to query the target object set when a source object has no linked target. The syntax for writing the query depends on the target system of the correlation. For examples of correlation queries, see [Writing Correlation Queries](#). The source object is provided in the `source` property in the script scope.

correlationScript

script object, optional

A script that goes beyond a `correlationQuery` of a target system. Used when you need another method to determine which records in the target system relate to the given source record. The syntax depends on the target of the correlation. For information about defining correlation scripts, see [Writing Correlation Scripts](#).

displayName

string, optional

The mapping name displayed in the UI.

enableLinking

boolean, true or false

Specifies whether links should be maintained between source and target objects for a mapping.

Default: true

enableSync

boolean, true or false

Specifies whether automatic synchronization (liveSync and implicit synchronization) should be enabled for a specific mapping. For more information, see [Disable Automatic Synchronization Operations](#).

Default: true

linkQualifiers

list of strings or script object, optional

Enables mapping of a single source object to multiple target objects.

Example: `"linkQualifiers" : ["employee", "customer"]` or

```
"linkQualifiers" : {
  "type" : "text/javascript",
  "globals" : { },
  "source" : "if (returnAll) {
              ['contractor', 'employee', 'customer',
'manager' ]
            } else {
```

```

        if(object.type === 'employee') {
            ['employee', 'customer', 'manager']
        } else {
            ['contractor', 'customer']
        }
    }"
}

```

If a script object, the script must return a list of strings.

links

string, optional

Enables reuse of the links created in another mapping. Example:

"systemLdapAccounts_managedUser" reuses the links created by a previous mapping whose name is "systemLdapAccounts_managedUser" .

name

string, required

Uniquely names the object mapping. Used in the link object identifier.

onCreate

script object, optional

A script to execute when a target object is to be created, after property mappings have been applied. In the root scope, the source object is provided in the `source` property, the projected target object in the `target` property, and the link situation that led to the create operation in the `situation` property. Properties on the target object can be modified by the script. If a property value is not set by the script, IDM falls back on the default property mapping configuration. If the script throws an exception, the target object creation is aborted.

onDelete

script object, optional

A script to execute when a target object is to be deleted, after property mappings have been applied. In the root scope, the source object is provided in the `source` property, the target object in the `target` property, and the link situation that led to the delete operation in the `situation` property. If the script throws an exception, the target object deletion is aborted.

onLink

script object, optional

A script to execute when a source object is to be linked to a target object, after property mappings have been applied. In the root scope, the source object is

provided in the `source` property, and the projected target object in the `target` property.

Note that, although an `onLink` script has access to a copy of the target object, changes made to that copy will not be saved to the target system automatically. If you want to persist changes made to target objects by an `onLink` script, you must explicitly include a call to the action that should be taken on the target object (for example `openidm.create`, `openidm.update` or `openidm.delete`) within the script.

In the following example, when an LDAP target object is linked, the `"description"` attribute of that object is updated with the value `"Active Account"`. A call to `openidm.update` is made within the `onLink` script, to set the value.

```
"onLink" : {
  "type" : "text/javascript",
  "source" : "target.description = 'Active Account';
             openidm.update('system/ldap/account/' +
target._id, null, target);"
}
```

If the script throws an exception, target object linking is aborted.

onMapping

script object, optional

A script that is run as part of a mapping. For example, a script that applies effective assignments as part of the mapping. This is run after the mappings have been applied, but before `onUpdate` scripts are executed.

onUnlink

script object, optional

A script to execute when a source and a target object are to be unlinked, after property mappings have been applied. In the root scope, the source object is provided in the `source` property, and the target object in the `target` property.

Although an `onUnlink` script has access to a copy of the target object, changes made to that copy will not be saved to the target system automatically. If you want to persist changes made to target objects by an `onUnlink` script, you must explicitly include a call to the action that should be taken on the target object (for example, `openidm.create`, `openidm.update` or `openidm.delete`) within the script.

In the following example, when an LDAP target object is unlinked, the `description` attribute of that object is updated with the value `Inactive Account`. A call to `openidm.update` is made within the `onUnlink` script, to set the value.

```
"onUnlink" : {
  "type" : "text/javascript",
  "source" : "target.description = 'Inactive Account';
             openidm.update('system/ldap/account/' +
target._id, null, target);"
}
```

If the script throws an exception, target object unlinking is aborted.

onUpdate

script object, optional

A script to execute when a target object is to be updated, after property mappings have been applied. In the root scope, the source object is provided in the `source` property, the projected target object in the `target` property, and the link situation that led to the update operation in the `situation` property. Any changes that the script makes to the target object will be persisted when the object is finally saved to the target resource. If the script throws an exception, the target object update is aborted.

optimizeAssignmentSync

boolean

When set to `true`, a synchronization event is only triggered by modifications to assignments if those modifications are relevant to that particular mapping. When set to `false` (the default), a synchronization event is triggered for all members of a role or assignment if the relationships or attributes for an assignment are modified.

IMPORTANT

Don't include `effectiveAssignments` in `triggerSyncProperties` as it negates all performance improvements provided by `optimizeAssignmentSync`.

policies

array of policy objects, optional

Specifies a set of link conditions and associated actions to take in response.

postMapping

script object, optional

A script to execute when sync has been performed on a managed object. This is run after the source object has been successfully synchronized with the target system.

properties

array of property-mapping objects, optional

Specifies mappings between source object properties and target object properties, with optional transformation scripts. See [Property Object Properties](#).

queuedSync

list of properties, optional

Specifies the [queued_synchronization](#) configuration.

reconAssociation

list of properties, optional

Specifies the [recon_association](#) configuration.

reconProgressStateUpdateInterval

integer, optional

Overrides the number of reconciliation operations required before the reconciliation progress state statistics are persisted to the repository. A value of 50 will write statistics to the repository every 50 operations.

Default : 1024 , minimum : 1 .

reconSourceQueryPageSize

integer

Sets the page size for reconciliation source queries, if paging is enabled.

Default : 10000

reconSourceQueryPaging

boolean, true or false

Specifies whether paging should be used for reconciliation source queries.

- Default for non-clustered reconciliation : false
- Default for clustered reconciliation : true

reconTargetQueryPageSize

integer

Sets the page size for reconciliation target queries, if paging is enabled.

Default : 10000

reconTargetQueryPaging

boolean, true or false

Specifies whether paging should be used for reconciliation target queries.

- Default for non-clustered reconciliation : false
- Default for clustered reconciliation : true

result

script object, optional

A script executed after a reconciliation finishes.

The variables available to a `result` script are as follows:

source Provides statistics about the source phase of the reconciliation

target Provides statistics about the target phase of the reconciliation

global Provides statistics about the entire reconciliation operation

context Information related to the current operation, such as source and target.

mappingConfig A configuration object representing the mapping being processed.

reconState Provides the state of reconciliation operation; such as, *success*, *failure*, or *active*.

runTargetPhase

boolean, true or false

Specifies whether reconciliation operations should run both the source and target phase. To avoid queries on the target resource, set to `false`.

Default: `true`

source

string, required

Specifies the path of the source object set. Example: `"managed/user"`.

sourceCondition

script object or `queryFilter` string, optional

A script or query filter that determines if a source object should be included in the mapping. If no `sourceCondition` element (or `validSource` script) is specified, all source objects are included in the mapping.

sourceIdsCaseSensitive

boolean, true or false

Consider case sensitivity when linking source IDs. Only effective if this mapping defines links, ignored if the mapping re-uses another mapping's links.

Default: `true`

sourceQueryFullEntry

boolean, true or false, optional

Specifies whether the defined source query returns full object data (`true`) or IDs only (`false`).

No default. If not set in the configuration, IDM will attempt to auto-detect the setting, based on the query results.

syncAfter

list of strings, optional

The specified mapping must be synchronized after all mappings in this list.

target

string, required

Specifies the path of the target object set. Example: "system/ldap/account" .

targetIdsCaseSensitive

boolean, true or false

Consider case sensitivity when linking target IDs. Only effective if this mapping defines links, ignored if the mapping re-uses another mapping's links.

Default : `true`

targetQueryFullEntry

Boolean `true` or `false`, optional

Specifies whether the defined target query returns full object data (`true`) or IDs only (`false`).

No default. If not set in the configuration, IDM will attempt to auto-detect the setting, based on the query results.

taskThreads

integer, optional

Sets the number of threads dedicated to the same reconciliation run.

Default : `10`

triggerSyncProperties

list, optional

A list of JsonPointers to fields in the source object whose changes should trigger a synchronization operation.

validSource

script object, optional

A script that determines if a source object is valid to be mapped. The script yields a boolean value: `true` indicates the source object is valid; `false` can be used to defer mapping until some condition is met. In the root scope, the source object is provided in the `source` property. If the script is not specified, then all source objects are considered valid.

validTarget

script object, optional

A script used during the target phase of reconciliation that determines if a target object is valid to be mapped. The script yields a boolean value: `true` indicates that the target object is valid; `false` indicates that the target object should not be included in reconciliation. In the root scope, the target object is provided in the `target` property. If the script is not specified, then all target objects are considered valid for mapping.

Property objects

A property object specifies how the value of a target property is determined.

```
{
  "target" : string,
  "source" : string,
  "transform" : script object,
  "condition" : script object,
  "default": value
}
```

Property object properties

target

string, required

Specifies the path of the property in the target object to map to.

source

string, optional

Specifies the path of the property in the source object to map from. If not specified, then the target property value is derived from the script or default value.

transform

script object, optional

A script to determine the target property value. The root scope contains the value of the source in the `source` property, if specified. If the `source` property has a value of `" "`, the entire source object of the mapping is contained in the root scope. The resulting value yielded by the script is stored in the target property.

condition

script object, optional

A script to determine whether the mapping should be executed or not. The condition has an "object" property available in root scope, which (if specified) contains the full source object. For example "source": "(object.email != null)". The script is considered to return a boolean value.

default

any value, optional

Specifies the value to assign to the target property if a non-null value is not established by source or transform. If not specified, the default value is null.

Policy objects

A policy object specifies a link condition and the associated actions to take in response.

```
{
  "condition" : optional, script object,
  "situation" : string,
  "action"    : string or script object
  "postAction" : optional, script object
}
```

Policy object properties

condition

script object or queryFilter condition, optional

Applies a policy, based on the link type, for example "condition" :
"/linkQualifier eq \"user\"".

A queryFilter condition can be expressed in two ways—as a string ("condition" :
"/linkQualifier eq \"user\"") or a map, for example:

```
"condition" : {
  "type" : "queryFilter",
  "filter" : "/linkQualifier eq \"user\""
}
```

It is generally preferable to express a queryFilter condition as a map.

A condition script has the following variables available in its scope: object and linkQualifier.

situation

string, required

Specifies the situation for which an associated action is to be defined.

action

string or script object, required

Specifies the action to perform. If a script is specified, the script is executed and is expected to yield a string containing the action to perform.

The `action` script has the following variables available in its scope: `source`, `target`, `sourceAction`, `linkQualifier`, and `recon`.

postAction

script object, optional

Specifies the action to perform after the previously specified action has completed.

The `postAction` script has the following variables available in its scope: `source`, `target`, `action`, `sourceAction`, `linkQualifier`, and `reconID`. `sourceAction` is `true` if the action was performed during the source reconciliation phase, and `false` if the action was performed during the target reconciliation phase. For more information, see [How Synchronization Situations Are Assessed](#).

NOTE

No `postAction` script is triggered if the `action` is either `IGNORE` or `ASYNC`.

Script Object

Script objects take the following form.

```
{
  "type" : "text/javascript",
  "source": string
}
```

type

string, required

The script type.

IDM supports `"text/javascript"` and `"groovy"`.

source

string, required

Specifies the source code of the script to be executed.

Links

By default, links are maintained between source and target objects in mappings. This behavior is governed by the `enableLinking` property in the mapping. If `enableLinking` is set to `false`, links are not maintained.

You might want to disable linking in the case of a [bulk import](#) or during [data migration](#).

To maintain links between source and target objects in mappings, IDM stores an object set in the repository. The object set identifier follows this scheme:

```
links/mapping
```

Here, `mapping` represents the name of the mapping for which links are managed.

Link entries have the following structure:

```
{
  "_id":string,
  "_rev":string,
  "linkType":string,
  "firstId":string
  "secondId":string,
}
```

_id

string

The identifier of the link object.

_rev

string, required

The value of link object's revision.

linkType

string, required

The type of the link. Usually the name of the mapping which created the link.

firstId

string, required

The identifier of the first of the two linked objects.

secondId

string

The identifier of the second of the two linked objects.

Queries

IDM performs the following queries on a link object set:

1. Find link(s) for a given firstId object identifier.

```
SELECT * FROM links WHERE linkType = value AND firstId = value
```

Although a single result makes sense, this query is intended to allow multiple results so that this scenario can be handled as an exception.

2. Select link(s) for a given second object identifier.

```
SELECT * FROM links WHERE linkType = value AND secondId = value
```

Although a single result makes sense, this query is intended to allow multiple results so that this scenario can be handled as an exception.

Reconciliation stages

IDM performs reconciliation on a per-mapping basis. The process of reconciliation for a given mapping includes these stages:

1. Iterate through all objects for the object set specified as `source`. For each source object, carry out the following steps.
 - a. Look for a link to a target object in the link object set, and perform a correlation query (if defined).
 - b. Determine the link condition, as well as whether a target object can be found.
 - c. Determine the action to perform based on the policy defined for the condition.
 - d. Perform the action.
 - e. Keep track of the target objects for which a condition and action has already been determined.
 - f. Write the results.
2. Iterate through all object identifiers for the object set specified as `target`. For each identifier, carry out the following steps:
 - a. Find the target in the link object set.

Determine if the target object was handled in the first phase.
 - b. Determine the action to perform based on the policy defined for the condition.

- c. Perform the action.
 - d. Write the results.
3. Iterate through all link objects, carrying out the following steps.
- a. If the `reconId` is "my" , then skip the object.

If the `reconId` is not recognized, then the source or the target is missing.
 - b. Determine the action to perform based on the policy.
 - c. Perform the action.
 - d. Store the `reconId` identifier in the mapping to indicate that it was processed in this run.

NOTE

To optimize a reconciliation operation, the reconciliation process does not attempt to correlate source objects to target objects if the set of target objects is empty when the correlation is started. For information on changing this default behaviour, see [Correlate Empty Target Sets](#).

REST API

External synchronized objects expose an API to request immediate synchronization. This API includes the following requests and responses.

Request

Example:

```
POST /openidm/system/csvfile/account/jsmith?_action=liveSync
HTTP/1.1
```

Response (success)

Example:

```
HTTP/1.1 204 No Content
...
```

Response (synchronization failure)

Example:

```
HTTP/1.1 409 Conflict
...
[JSON representation of error]
```

Reconciliation duration metrics

[Obtaining the Details of a Reconciliation](#) describes how to obtain the details of a reconciliation run over REST. This section provides more information on the metrics returned when you query the `recon` endpoint. Reconciliation is processed as a series of distinct tasks. The `durationSummary` property indicates the period of time spent on each task. You can use this information to address reconciliation performance bottlenecks.

The following sample output shows the kind of information returned for each reconciliation run:

```
{
  "_id": "3bc72717-a4bb-4871-b936-3a5a560c1a7c-37",
  "duration": 781561,
  "durationSummary": {
    "auditLog": {
      ...
    },
    ...
    "sourceObjectQuery": {
      "count": 100,
      "max": 96,
      "mean": 14,
      "min": 6,
      "stdDev": 16,
      "sum": 1450
    },
    "sourcePagePhase": {
      "count": 1,
      "max": 20944,
      "mean": 20944,
      "min": 20944,
      "stdDev": 0,
      "sum": 20944
    },
    "sourceQuery": {
      "count": 1,
      "max": 120,
      "mean": 120,
      "min": 120,
      "stdDev": 0,
      "sum": 120
    },
    "targetPhase": {
```

```

        "count": 1,
        "max": 0,
        "mean": 0,
        "min": 0,
        "stdDev": 0,
        "sum": 0
    },
    "targetQuery": {
        "count": 1,
        "max": 19657,
        "mean": 19657,
        "min": 19657,
        "stdDev": 0,
        "sum": 19657
    }
},
...
}

```

The specific reconciliation tasks that are run depend on the configuration for that mapping. For example, the `sourcePagePhase` is run only if paging is enabled. The `linkQuery` is run only for non-clustered reconciliation operations, because an initial query of all links does not make sense if a single source page query is being run.

Recon tasks

The following list describes all the possible tasks that can be run for a single reconciliation:

sourcePhase

This phase runs only for non-clustered, non-paged reconciliations. The total duration (`sum`) is the time spent processing all records on the source system.

sourcePagePhase

Queries and processes individual objects in a page, based on their IDs. This phase is run only for clustered reconciliations or for non-clustered reconciliations that have source paging configured. The total duration (`sum`) is the total time spent processing source pages across all cluster nodes. This processing occurs in parallel across all cluster nodes, so it is normal for the `sourcePagePhase` duration to exceed the total reconciliation duration.

sourceQuery

Obtains all IDs on the source system, or in a specific source page.

NOTE

When the `sourceQuery` returns a null paging cookie, indicating that there are no more source IDs to reconcile, the clustered reconciliation process dispatches a scheduled job named `sourcePageCompletionCheck`.

This job checks for remaining source page jobs on the scheduler. If there are no remaining source page jobs, the `sourcePageCompletionCheck` schedules the target phase. If there are still source page jobs to process, the `sourcePageCompletionCheck` schedules another instance of itself to perform these checks again after a few seconds.

Because the target phase reconciles all IDs that were not reconciled during the source phase, it cannot start until all of the source pages are complete. Final reconciliation statistics cannot be generated and logged until all source page jobs have completed, so the `sourcePageCompletionCheck` runs even if the target phase is not enabled.

sourceObjectQuery

Queries the individual objects on the source system or page, based on their IDs.

validSourceScript

Processes any scripts that should be run to determine if a source object is valid to be mapped.

linkQuery

Queries any existing links between source and target objects.

This phase includes the following tasks:

sourceLinkQuery

Queries any existing links from source objects to target objects.

targetLinkQuery

Queries any existing links from target objects that were not processed during the `sourceLinkQuery` phase.

linkQualifiersScript

Runs any link qualifier scripts. For more information, see [Map a Single Source Object to Multiple Target Objects](#).

onLinkScript

Processes any scripts that should be run when source and target objects are linked.

onUnlinkScript

Processes any scripts that should be run when source and target objects are unlinked.

deleteLinkObject

Deletes any links that are no longer relevant between source and target objects.

correlationQuery

Processes any configured correlation queries. For more information, see [Writing Correlation Queries](#).

correlationScript

Processes any configured correlation scripts. For more information, see [Writing Correlation Scripts](#).

onMappingScript

For roles, processes the script that applies the effective assignments as part of the mapping.

activePolicyScript

Sets the action and active policy based on the current situation.

activePolicyPostActionScript

Processes any scripts configured to run after policy validation.

targetPhase

The aggregated result for time spent processing records on the target system.

targetQuery

Queries all IDs on the target system. The list of IDs is restricted to IDs that have not already been linked to a source ID during the source phase. The target query generates a list of *orphan* IDs that must be reconciled if the target phase is not disabled.

targetObjectQuery

Queries the individual objects on the target system, based on their IDs.

validTargetScript

Processes any scripts that should be run to determine if a target object is valid to be mapped.

onCreateScript

Processes any scripts that should be run when a new target object is created.

updateTargetObject

Updates existing linked target objects, based on the configured situations and actions.

onUpdateScript

Processes any scripts that should be run when a target object is updated.

deleteTargetObject

Deletes any objects on the target resource that must be removed in accordance with the defined synchronization actions.

onDeleteScript

Processes any scripts that should be run when a target object is deleted.

resultScript

Processes the script that is executed when a reconciliation process has finished.

propertyMappingScript

Runs any scripts configured for when source and target properties are mapped.

postMappingScript

Processes any scripts that should be run when synchronization has been performed on the managed/user object.

onReconScript

Processes any scripts that should be run after source and target systems are reconciled.

auditLog

Writes reconciliation results to the audit log.

Metrics Collected

For each phase, the following metrics are collected:

count

The number of objects or records processed during that phase. For the `sourcePageQuery` phase, the `count` parameter refers to the page size.

When the `count` statistic of a particular task refers to the number of records being reconciled, the `sum` statistic of that task represents the total time across the total number of threads running in all nodes in the cluster. For example:

```
"updateTargetObject": {  
  "count": 1000000,  
  "max": 1193,  
  "mean": 35,  
  "min": 11,  
  "stdDev": 0,  
  "sum": 35065991  
}
```

max

The maximum time, in milliseconds, spent processing a record during that phase.

mean

The average time, in milliseconds, spent processing a record during that phase.

min

The minimum time, in milliseconds, spent processing a record during that phase.

stdDev

The standard deviation, which measures the variance of the individual values from the mean.

sum

The total amount of time, in milliseconds, spent during that phase.

-
1. In this case (and the two following cases), the DELETE action is applied to the linked target object and not necessarily to the target object(s) found by the correlation query. If the source is no longer valid and a link existed, the correlation logic is skipped.
 2. The source object qualifies for a target object and is not linked to an existing target object. There is a single target object that correlates with this source object, according to the logic in the correlation, but that target object is already linked to a different source object.
 3. The source object qualifies for a target object, is not linked to an existing target object, but there is more than one correlated target object (that is, more than one possible match on the target system).
 4. If the action is CREATE for the situation MISSING, the orphaned link associated with the source object is updated to point to the new target object. When a target object is deleted, the link from the target to the corresponding source object is not deleted automatically. This lets IDM detect and report items that might have been removed without permission or might need review. If you need to remove the corresponding link when a target object is deleted, change the action to UNLINK to remove the link, or to DELETE to remove the target object and the link.
 5. During target reconciliation, the target becomes unqualified by the `validTarget` script.
 6. Detected during reconciliation and target change events
 7. Detected during reconciliation and target change events
 8. If no link exists for the source object, IDM executes any included correlation logic. If a link exists, correlation does not apply.

Was this helpful?  