Scripting Guide

/ ForgeRock Identity Management 7

Latest update: 7.0.1

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

Abstract

Guide to scripting for ForgeRock® Identity Management.

Table of Contents

Overview

Scripting lets you extend IDM functionality. For example, you can provide custom logic between source and target mappings, define correlation rules, filters, triggers, and so on. This guide shows you how to use scripts in IDM and provides reference information on the script engine.

Quick Start

 Script Configuration Modify the parameters to compile, debug, and run scripts.	 Custom Endpoints Run arbitrary scripts through the REST URI.
 Script Triggers Learn where and how you can trigger scripts.	 Script Variables Learn about the variables available to scripts.

IDM supports scripts written in JavaScript and Groovy and uses the following libraries:

- Rhino version 1.7.12 to run JavaScript.

> **Note**
>
> Rhino has limited support for ES6 / ES2015 (JavaScript version 1.7). For more information, see Rhino ES2015 Support.

- Groovy version 3.0.4 for Groovy script support.
- Lodash 3.10.1 and Handlebars 4.7.6 for Rhino scripting.

> **Note**
>
> Using Handlebars JS in server-side JS scripts requires *synchronization*; for example:

```
var Handlebars = require("lib/handlebars");
var result = new Packages.org.mozilla.javascript.Synchronizer(function() {
  var template = Handlebars.compile("Handlebars {{doesWhat}}");
  return template({ doesWhat: "rocks!" });
}, Handlebars)();
console.log(result);
```

Note

Script options and locations are defined in `conf/script.json`. Default scripts are located in (`/path/to/openidm/bin/defaults/script/`). Do not modify the scripts in this directory. Rather copy the default scripts to a different location, make the changes, and update the referenced scripts in the applicable `conf/` file. You can put custom scripts in any of the locations referenced in the `sources` property in `conf/script.json`.

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

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

Chapter 1
Script Configuration

To modify the parameters used for compiling, debugging, and running scripts, edit the script configuration file (`conf/script.json`).

`script.json` *Parameters*

properties

Any custom properties.

ECMAScript

JavaScript debug and compile options. JavaScript is an ECMAScript language.

- `javascript.optimization.level` - The current optimization level. Expected integer range is from -1 to 9. For more information about optimization level, see Rhino Optimization.

  The default value is `9`.

- `javascript.recompile.minimumInterval` - The minimum time between script recompile.

  The default value is `60000`, or 60 seconds. This means that any changes made to scripts will not get picked up for up to 60 seconds. If you are developing scripts, reduce this parameter to around `100` (100 milliseconds).

  If you set the `javascript.recompile.minimumInterval` to `-1`, or remove this property from the `script.json` file, IDM does not poll JavaScript files to check for changes.

Groovy

Compilation and debugging options related to Groovy scripts. Many of these options are commented out in the default script configuration file. Remove the comments to set these properties:

- `groovy.warnings` - The Groovy script log level. Possible values are `none`, `likely`, `possible`, and `paranoia`.
- `groovy.source.encoding` - The Groovy script encoding format. Possible values are `UTF -8` and `US-ASCII`.
- `groovy.target.directory` - The compiled Groovy class output directory. The default directory is *`install-dir`*`/classes`.

- `groovy.target.bytecode` - The Groovy script bytecode version. The default version is `1.5`.
- `groovy.classpath` - The directory where the compiler should look for compiled classes. The default classpath is *`install-dir`*`/lib`.

  To call an external library from a Groovy script, you must specify the complete path to the .jar file or files, as a value of this property. For example:

  ```
  "groovy.classpath" : "/&{idm.install.dir}/lib/http-builder-0.7.1.jar:
          /&{idm.install.dir}/lib/json-lib-2.3-jdk15.jar:
          /&{idm.install.dir}/lib/xml-resolver-1.2.jar:
          /&{idm.install.dir}/lib/commons-collections-3.2.1.jar",
  ```

  Note

  If you're deploying on Microsoft Windows, use a semicolon (`;`) instead of a colon to separate directories in the `groovy.classpath`.

- `groovy.output.verbose` - Verbosity of stack traces. Boolean, `true` or `false`.
- `groovy.output.debug` - Whether to output debug messages. Boolean, `true` or `false`.
- `groovy.errors.tolerance` - The number of non-fatal errors that can occur before a compilation is aborted. The default is `10` errors.
- `groovy.script.extension` - Groovy script file extension. The default is `.groovy`.
- `groovy.script.base` - Groovy script base class. By default, any class extends `groovy.lang.Script`.
- `groovy.recompile` - Whether scripts can be recompiled. Boolean, `true` or `false`, with default `true`.
- `groovy.recompile.minimumInterval` - Groovy script minimum recompile interval.

  The default value is `60000`, or 60 seconds. Using the default value, any changes made to scripts may not be in effect for up to 60 seconds. If you are developing scripts, reduce this parameter to `100` (100 milliseconds).

- `groovy.target.indy` - Whether to use a Groovy indy test. Boolean, `true` or`false`, with default `true`.
- `groovy.disabled.global.ast.transformations` - A list of disabled Abstract Syntax Transformations (ASTs).

sources

The directories where IDM looks for referenced scripts.

Excerpt of `script.json` displaying default directories:

```
"sources" : {
   "default" : {
       "directory" : "&{idm.install.dir}/bin/defaults/script"
   },
   "install" : {
       "directory" : "&{idm.install.dir}"
   },
   "project" : {
       "directory" : "&{idm.instance.dir}"
   },
   "project-script" : {
       "directory" : "&{idm.instance.dir}/script"
}
```

Note

IDM loads scripts from `sources` in reverse order (bottom to top).

Note

By default, debug information (for example, file name and line number) is excluded from JavaScript and Groovy exceptions. To troubleshoot script exceptions, you can include debug information by changing the following settings to `true` in `resolver/boot.properties`:

```
javascript.exception.debug.info=false
groovy.exception.debug.info=false
```

Including debug information in a production environment is not recommended.

Chapter 2
Call a Script From a Configuration File

To call a script from a configuration file, provide the script source or reference a file that contains the script source. For example:

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

or

```
{
   "type" : "text/javascript",
   "file" : file location
}
```

Script variables are not necessarily simple `key:value` pairs, and can be any arbitrarily complex JSON object.

type

string, required

Specifies the type of script to be executed. Supported types include `text/javascript` and `groovy`.

source

string, required if `file` is not specified

Specifies the source code of the script to be executed.

file

string, required if `source` is not specified

Specifies the file containing the source code of the script to execute. The file path must be relative to *project-dir*. Absolute paths are not supported.

Tip

In general, you should namespace variables passed into scripts with the `globals` map. Passing variables in this way prevents collisions with the top-level reserved words for script maps, such as `file`, `source`, and `type`. The following example uses the `globals` map to namespace the variables passed in the previous example.

```
"script": {
    "type" : "text/javascript",
    "file" : "script/triggerEmailNotification.js",
    "globals" : {
        "fromSender" : "admin@example.com",
        "toEmail" : "user@example.com"
    }
}
```

+ *Examples*

The following example script (included in `sync.json`) determines whether to include or ignore a target object in the reconciliation process based on an `employeeType` of `true`:

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

The following example script (included in `sync.json`) sets the `__PASSWORD__` attribute to `defaultpwd` when IDM creates a target object:

```
"onCreate" : {
    "type" : "text/javascript",
    "source" : "target.__PASSWORD__ = 'defaultpwd'"
}
```

The following example script (included in `router.json`) uses a trigger to create Solaris home directories using a script. The script is located in the file, *`project-dir`*`/script/createUnixHomeDir.js` :

```
{
    "filters" : [
        {
            "pattern" : "^system/solaris/account$",
            "methods" : [ "create" ],
            "onResponse" : {
                "type" : "text/javascript",
                "file" : "script/createUnixHomeDir.js"
            }
        }
    ]
}
```

Often, script files are reused in different contexts. You can pass variables to your scripts to provide these contextual details at runtime. You pass variables to the scripts that are referenced in configuration files by declaring the variable name in the script reference.

The following example of a scheduled task configuration calls a script named `triggerEmailNotification.js`. The example sets the sender and recipient of the email in the schedule configuration, rather than in the script itself:

```
{
    "enabled" : true,
    "type" : "cron",
    "schedule" : "0 0/1 * * * ?",
    "persisted" : true,
    "invokeService" : "script",
    "invokeContext" : {
        "script" : {
            "type" : "text/javascript",
            "file" : "script/triggerEmailNotification.js",
            "fromSender" : "admin@example.com",
            "toEmail" : "user@example.com"
        }
    }
}
```

Chapter 3

Validate Scripts Over REST

IDM exposes a `script` endpoint over which scripts can be validated, by specifying the script parameters as part of the JSON payload. This functionality lets you test how a script will operate in your deployment, with complete control over the inputs and outputs. Testing scripts in this way can be useful in debugging.

In addition, the script registry service supports calls to other scripts. For example, you might have logic written in JavaScript, but also some code available in Groovy. Ordinarily, it would be challenging to interoperate between these two environments, but this script service lets you call one from the other on the IDM router.

The `script` endpoint supports two actions - `eval` and `compile`.

The `eval` action evaluates a script, by taking any actions referenced in the script, such as router calls to affect the state of an object. For JavaScript scripts, the last statement that is executed is the value produced by the script, and the expected result of the REST call.

The following REST call attempts to evaluate the `autoPurgeAuditRecon.js` script (provided in `openidm/bin/defaults/script/audit`), but provides an incorrect purge type (`"purgeByNumOfRecordsToKeep"` instead of `"purgeByNumOfReconsToKeep"`). The error is picked up in the evaluation. The example assumes that the script exists in the directory reserved for custom scripts (`openidm/script`):

```
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 '{
  "type": "text/javascript",
  "file": "script/autoPurgeAuditRecon.js",
  "globals": {
    "input": {
      "mappings": ["%"],
      "purgeType": "purgeByNumOfRecordsToKeep",
      "numOfRecons": 1
    }
  }
}' \
"http://localhost:8080/openidm/script?_action=eval"

"Must choose to either purge by expired or number of recons to keep"
```

Tip

The variables passed into this script are namespaced with the `globals` map. It is preferable to namespace variables passed into scripts in this way, to avoid collisions with the top-level reserved words for script maps, such as `file`, `source`, and `type`.

The `compile` action compiles a script, but does not execute it. This action is used primarily by the UI, to validate scripts that are entered in the UI. A successful compilation returns `true`. An unsuccessful compilation returns the reason for the failure.

The following REST call tests whether a transformation script will compile:

```
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 '{
  "type":"text/javascript",
  "source":"source.mail ? source.mail.toLowerCase() : null"
}' \
"http://localhost:8080/openidm/script?_action=compile"
True
```

If the script is not valid, the action returns an indication of the error, for example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--header "Content-Type: application/json" \
--request POST \
--data '{
  "type":"text/javascript",
  "source":"source.mail ? source.mail.toLowerCase()"
}' \
"http://localhost:8080/openidm/script?_action=compile"
{
  "code": 400,
  "reason": "Bad Request",
  "message": "missing : in conditional expression (386...BF2#1)in 386...BF2 at line number 1 at column
 number 39"
}
```

Chapter 4

Create Custom Endpoints to Launch Scripts

Custom endpoints let you run arbitrary scripts through the REST URI.

Custom endpoints are configured in files named `conf/endpoint-name.json`, where *name* generally describes the purpose of the endpoint. The endpoint configuration file includes an inline script or a reference to a script file, in either JavaScript or Groovy. The referenced script provides the endpoint functionality.

A sample custom endpoint configuration is provided in the `openidm/samples/example-configurations/custom-endpoint` directory. The sample includes three files:

conf/endpoint-echo.json

Provides the configuration for the endpoint.

script/echo.js

Provides the endpoint functionality in JavaScript.

script/echo.groovy

Provides the endpoint functionality in Groovy.

Note

This sample endpoint is described in detail in "*Create a Custom Endpoint*" in the *Samples Guide*.

Custom Endpoint Configuration File

An endpoint configuration file includes the following elements:

```
{
    "context" : "endpoint/linkedView/*",
    "type" : "text/javascript",
    "source" : "require('linkedView').fetch(request.resourcePath);"
}
```

`context`

string, optional

The context path under which the custom endpoint is registered, in other words, the *route* to the endpoint. An endpoint with the context `endpoint/test` is addressable over REST at the URL `http://localhost:8080/openidm/endpoint/test` or by using a script such as `openidm.read("endpoint/test")`.

Endpoint contexts support wild cards, as shown in the preceding example. The `endpoint/linkedview/*` route matches the following patterns:

```
endpoint/linkedView/managed/user/bjensen
endpoint/linkedView/system/ldap/account/bjensen
endpoint/linkedView/
endpoint/linkedView
```

The `context` parameter is not mandatory in the endpoint configuration file. If you do not include a `context`, the route to the endpoint is identified by the name of the file. For example, in the sample endpoint configuration provided in `openidm/samples/example-configurations/custom-endpoint/conf/endpoint-echo.json`, the route to the endpoint is `endpoint/echo`.

Note

This `context` path is not the same as the *context chain* of the request. For information about the request context chain, see "*Request Context Chain*".

`type`

string, required

The type of script to be executed, either `text/javascript` or `groovy`.

`file` or `source`

The path to the script file, or the script itself, inline.

For example:

```
"file" : "workflow/gettasksview.js"
```

or

```
"source" : "require('linkedView').fetch(request.resourcePath);"
```

Note

You must set authorization for any custom endpoints that you add, for example, by restricting the methods to the appropriate roles. For more information, see"Authorization and Roles" in the *Security Guide*.

Custom Endpoint Scripts

The custom endpoint script files in the `samples/example-configurations/custom-endpoint/script` directory demonstrate all the HTTP operations that can be called by a script. Each HTTP operation is

associated with a `method` - `create`, `read`, `update`, `delete`, `patch`, `action`, or `query`. Requests sent to the custom endpoint return a list of the variables available to each method.

All scripts are invoked with a global `request` variable in their scope. This request structure carries all the information about the request.

Warning

Read requests on custom endpoints must not modify the state of the resource, either on the client or the server, as this can make them susceptible to CSRF exploits.

The standard READ endpoints are safe from Cross Site Request Forgery (CSRF) exploits because they are inherently read-only. That is consistent with the *Guidelines for Implementation of REST*, from the US National Security Agency, as "... CSRF protections need only be applied to endpoints that will modify information in some way."

Custom endpoint scripts *must* return a JSON object. The structure of the return object depends on the `method` in the request.

The following example shows the `create` method in the `echo.js` file:

```
if (request.method === "create") {
    return {
        method: "create",
        resourceName: request.resourcePath,
        newResourceId: request.newResourceId,
        parameters: request.additionalParameters,
        content: request.content,
        context: context.current
    }
}
```

The following example shows the `query` method in the `echo.groovy` file:

```
else if (request instanceof QueryRequest) {
    // query results must be returned as a list of maps
    return [
        [
            method: "query",
            resourceName: request.resourcePath,
            pagedResultsCookie: request.pagedResultsCookie,
            pagedResultsOffset: request.pagedResultsOffset,
            pageSize: request.pageSize,
            queryId: request.queryId,
            queryFilter: request.queryFilter.toString(),
            parameters: request.additionalParameters,
            context: context.toJsonValue().getObject()
        ]
    ]
}
```

Depending on the method, the variables available to the script can include the following:

`resourceName`

The name of the resource, without the `endpoint/` prefix, such as `echo`.

`newResourceId`

The identifier of the new object, available as the results of a `create` request.

`revision`

The revision of the object.

`parameters`

Any additional parameters provided in the request. The sample code returns request parameters from an HTTP GET with `?param=x`, as `"parameters":{"param":"x"}`.

`content`

Content based on the latest revision of the object, using `getObject`.

`context`

The context of the request, including headers and security. For more information, see "*Request Context Chain*".

Paging parameters

The `pagedResultsCookie`, `pagedResultsOffset`, and `pageSize` parameters are specific to `query` methods. For more information see "Page Query Results" in the *Object Modeling Guide*.

Query parameters

The `queryId` and `queryFilter` parameters are specific to `query` methods. For more information see "Construct Queries" in the *Object Modeling Guide*.

Script Exceptions

Some custom endpoint scripts require exception-handling logic. To return meaningful messages in REST responses and in logs, you must comply with the language-specific method of throwing errors.

A script written in JavaScript should comply with the following exception format:

```
throw {
    "code": 400, // any valid HTTP error code
    "message": "custom error message",
    "detail" : {
        "var": parameter1,
        "complexDetailObject" : [
            "detail1",
            "detail2"
        ]
    }
}
```

Any exceptions will include the specified HTTP error code, the corresponding HTTP error message, such as `Bad Request`, a custom error message that can help you diagnose the error, and any additional detail that you think might be helpful.

A script written in Groovy should comply with the following exception format:

```
import org.forgerock.json.resource.ResourceException
import org.forgerock.json.JsonValue

throw new ResourceException(404, "Your error message").setDetail(new JsonValue([
    "var": "parameter1",
    "complexDetailObject" : [
        "detail1",
        "detail2"
    ]
]))
```

Chapter 5
Register Custom Scripted Actions

You can register custom scripts that initiate some arbitrary action on a managed object endpoint. You can declare any number of actions in your managed object schema and associate those actions with a script.

The return value of a custom scripted action is ignored. The managed object is returned as the response of the scripted action, whether that object has been updated by the script or not.

Custom scripted actions have access to the following variables:

- `context`
- `request`
- `resourcePath`
- `object`

Example Scenario

In this scenario, you want your managed users to have the option to receive update notifications. You can define an *action* that toggles the value of a specific property on the user object.

1. Add an `updates` property to the managed user schema (`conf/managed.json`):

```
"properties": {
    ...
    "updates": {
        "title": "Automatic Updates",
        "viewable": true,
        "type": "boolean",
        "searchable": true,
        "userEditable": true
    },
    ...
}
```

2. Add a `toggleUpdates` action to the managed user object definition:

```
{
    "objects" : [
        {
            "name" : "user",
            "onCreate" : {
                ...
            },
            ...
            "actions" : {
                "toggleUpdates" : {
                    "type" : "text/javascript",
                    "source" : "openidm.patch(resourcePath, null, [{ 'operation' : 'replace',
 'field' : '/updates', 'value' : !object.updates }])"
                }
            },
            ...
        }
    ]
}
```

Note

The `toggleUpdates` action calls a script that changes the value of the user's `updates` property.

3. To call the script, specify the ID of the action in a POST request on the user object:

```
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/managed/user/ID?_actionId=toggleUpdate"
```

Now you can test the functionality.

4. Create a managed user, `bjensen`, with an `updates` property set to `true`:

```
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 '{
  "userName":"bjensen",
  "sn":"Jensen",
  "givenName":"Barbara",
  "mail":"bjensen@example.com",
  "telephoneNumber":"5556787",
  "description":"Created by OpenIDM REST.",
  "updates": true,
  "password":"Passw0rd"
}' \
"http://localhost:8080/openidm/managed/user?_action=create"
{
  "_id": "9dce06d4-2fc1-4830-a92b-bd35c2f6bcbb",
  "_rev": "0000000050c62938",
  "userName": "bjensen",
  "sn": "Jensen",
  "givenName": "Barbara",
  "mail": "bjensen@example.com",
  "telephoneNumber": "5556787",
  "description": "Created by OpenIDM REST.",
  "updates": true,
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

5. Run the `toggleUpdates` action on `bjensen`:

```
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/managed/user/9dce06d4-2fc1-4830-a92b-bd35c2f6bcbb?
_action=toggleUpdates"
{
  "_id": "9dce06d4-2fc1-4830-a92b-bd35c2f6bcbb",
  "_rev": "00000000a92657c7",
  "userName": "bjensen",
  "sn": "Jensen",
  "givenName": "Barbara",
  "mail": "bjensen@example.com",
  "telephoneNumber": "5556787",
  "description": "Created by OpenIDM REST.",
  "updates": false,
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

Note

Note in the command output that this action has set bjensen's `updates` property to `false`.

Chapter 6

Request Context Chain

The context chain of any request is established as follows:

1. The request starts with a *root context*, associated with a specific context ID.
2. The root context is wrapped in the *security context* that includes the authentication and authorization detail for the request.
3. The security context is further wrapped by the *HTTP context*, with the target URI. The HTTP context is associated with the normal parameters of the request, including a user agent, authorization token, and method.
4. The HTTP context is wrapped by one or more server/router context(s), with an endpoint URI. The request can have several layers of server and router contexts.

Chapter 7
Script Triggers

Scripts can be triggered in different places, and by different events. The following list indicates the configuration files in which scripts can be referenced, the events upon which the scripts can be triggered and the actual scripts that can be triggered on each of these files.

Scripts called in mappings

Triggered by situation

onCreate, onUpdate, onDelete, onLink, onUnlink

Object filter

validSource, validTarget

Triggered when correlating objects

correlationQuery, correlationScript

Triggered on any reconciliation

result

Scripts inside properties

condition, transform

`sync.json` supports only one script per hook. If multiple scripts are defined for the same hook, only the last one is kept.

Scripts inside policies

condition

Within a synchronization policy, you can use a `condition` script to apply different policies based on the link type, for example:

```
"condition" : {
  "type" : "text/javascript",
  "source" : "linkQualifier == \"user\""
}
```

Scripts called in the managed object configuration (`conf/managed.json`) file

onCreate, onRead, onUpdate, onDelete, onValidate, onRetrieve, onStore, onSync, postCreate, postUpdate, and postDelete

`managed.json` supports only one script per hook. If multiple scripts are defined for the same hook, only the last one is kept.

Scripts called in the router configuration (`conf/router.json`) file

onRequest, onResponse, onFailure

`router.json` supports multiple scripts per hook.

Chapter 8
Script Variables

The variables available to a script depend on several factors:

- The trigger that launches the script
- The configuration file in which that trigger is defined
- The object type:
  - For a managed object (defined in `managed.json`), the object type is either a managed object configuration object, or a managed object property.
  - For a synchronization object (defined in `sync.json`), the object can be an object-mapping object, a property object, or a policy object. For more information, see "Policy Objects" in the *Synchronization Guide*).

The following tables list the variables available to scripts, based on the configuration file in which the trigger is defined.

Script Triggers Defined in `managed.json`

For information about how managed objects in `managed.json` are handled and what script triggers are available, see "*Managed Objects*" in the *Object Modeling Guide*.

Managed Object Configuration Object	
Trigger	**Variable**
`onCreate`, `postCreate`	• **object**: The content of the object being created. • **newObject**: The object after the create operation is complete. • **context**: Information related to the current request, such as client, end user, and routing. • **resourceName**: The resource path of the object of the query. For example, if you create a managed user with ID `42f8a60e-2019-4110-a10d-7231c3578e2b`, resourceName returns `managed/user/42f8a60e-2019-4110-a10d-7231c3578e2b`. • **request**: Information related to the request, such as headers, credentials, and the desired action. Also includes the endpoint, and payload to be processed.

Managed Object Configuration Object	
Trigger	**Variable**
`onUpdate`, `postUpdate` *Returns JSON object*	• **object**: The content of the object being updated. • **oldObject**: The state of the object, before the update. • **newObject**: Changes to be applied to the object. May continue with the `onUpdate` trigger. • **context**: Information related to the current request, such as client, end user, and routing. • **resourceName**: The resource path of the object the query. • **request**: Information related to the request, such as headers, credentials, and the desired action. Also includes the endpoint, and payload to be processed.
`onDelete`, `onRetrieve`, `onRead` *Returns JSON object.*	• **object**: The content of the object. • **context**: Information related to the current request, such as client, end user, and routing. • **resourceName**: The resource path of the object the query. • **request**: Information related to the request, such as headers, credentials, and the desired action. Also includes the endpoint, and payload to be processed.
`postDelete` *Returns JSON object.*	• **oldObject**: Represents the deleted object. • **context**: Information related to the current request, such as client, end user, and routing. • **resourceName**: The resource path of the object the query is performed upon. • **request**: Information related to the request, such as headers, credentials, and the desired action. Also includes the endpoint, and payload to be processed.
`onSync` *Returns JSON object*	• **oldObject**: Represents the object prior to sync. If sync has not been run before, the value will be `null`. • **newObject**: Represents the object after sync is completed. • **context**: Information related to the current request, such as client, end user, and routing. • **request**: Information related to the request, such as headers, credentials, and the desired action. Also includes the endpoint, and payload to be processed. • **resourceName**: An object representing the resource path the query is performed upon.

Managed Object Configuration Object	
Trigger	**Variable**
	• **syncResults**: A map containing the results and details of the sync, including: • **success** (boolean): Success or failure of the sync operation. • **action**: Returns the name of the action performed as a string. • **syncDetails**: The mappings attempted during synchronization.
`onStore`, `onValidate` *Returns JSON object*	• **object**: Represents the object being stored or validated • **value**: The content to be stored or validated for the object • **context**: Information related to the current request, such as client, end user, and routing. • **resourceName**: The resource path of the object the query is performed upon. • **request**: Information related to the request, such as headers, credentials, and the desired action. Also includes the endpoint, and payload to be processed.

property object	
Trigger	**Variable**
`onRetrieve`, `onStore` *Returns JSON object*	• **object**: Represents the object being operated upon • **property**: The value of the property being retrieved or stored • **propertyName**: The name of the property being retrieved or stored • **context**: Information related to the current request, such as client, end user, and routing. • **resourceName**: The resource path of the object the query is performed upon. • **request**: Information related to the request, such as headers, credentials, and the desired action. Also includes the endpoint, and payload to be processed.
`onValidate` *Returns JSON object*	• **property**: The value of the property being validated • **context**: Information related to the current request, such as client, end user, and routing. • **resourceName**: The resource path of the object the query is performed upon.

property object	
Trigger	**Variable**
	• **request**: Information related to the request, such as headers, credentials, and the desired action. Also includes the endpoint, and payload to be processed.

Script Triggers Defined in Mappings

For information about how managed objects in mappings are handled, and the script triggers available, see "Object-Mapping Objects" in the *Synchronization Guide*.

object-mapping object	
Trigger	**Variable**
`correlationQuery`, `correlationScript` *Returns JSON object*	• **source**: Represents the source object. • **linkQualifier**: The link qualifier associated with the current sync. • **context**: Information related to the current request, such as source and target.
`linkQualifiers` *Returns JSON object*	• **mapping**: The name of the current mapping. • **object**: The value of the source object. During a DELETE event, that source object may not exist, and may be null. • **oldValue**: The former value of the deleted source object, if any. If the source object is new, oldValue will be null. When there are deleted objects, oldValue is populated only if the source is a managed object. • **returnAll** (boolean): Link qualifier scripts must return every valid link qualifier when returnAll is true, independent of the source object. If returnAll is true, the script must not attempt to use the object variable, because it will be null. It's best practice to configure scripts to start with a check for the value of returnAll. • **context**: Information related to the current request, such as source and target.
`onCreate` *Returns JSON object*	• **source**: Represents the source object. • **target**: Represents the target object. • **situation**: The situation associated with the current sync operation. • **linkQualifier**: The link qualifier associated with the current sync operation. • **context**: Information related to the current sync operation. • **sourceId**: The object ID for the source object.

object-mapping object	
Trigger	**Variable**
	• **targetId**: The object ID for the target object. • **mappingConfig**: A configuration object representing the mapping being processed.
`onDelete`, `onUpdate` *Returns JSON object*	• **source**: Represents the source object. • **target**: Represents the target object. • **oldTarget**: Represents the target object prior to the DELETE or UPDATE action. • **situation**: The situation associated with the current sync operation. • **linkQualifier**: The link qualifier associated with the current sync. • **context**: Information related to the current sync operation. • **sourceId**: The object ID for the source object. • **targetId**: The object ID for the target object. • **mappingConfig**: A configuration object representing the mapping being processed.
`onLink`, `onUnlink` *Returns JSON object*	• **source**: Represents the source object. • **target**: Represents the target object. • **linkQualifier**: The link qualifier associated with the current sync operation. • **context**: Information related to the current sync operation. • **sourceId**: The object ID for the source object. • **targetId**: The object ID for the target object. • **mappingConfig**: A configuration object representing the mapping being processed.
`result` *Returns JSON object of reconciliation results*	• **source**: Provides statistics about the source phase of the reconciliation. • **target**: Provides statistics about the target phase of the reconciliation. • **context**: Information related to the current operation, such as source and target. • **global**: Provides statistics about the entire reconciliation operation.
`validSource` *Returns boolean*	• **source**: Represents the source object. • **linkQualifier**: The link qualifier associated with the current sync operation.

object-mapping object	
Trigger	**Variable**
	• **context**: Information related to the current sync operation.
`validTarget` *Returns boolean*	• **target**: Represents the target object. • **linkQualifier**: The link qualifier associated with the current sync operation. • **context**: Information related to the current sync operation.

property object	
Trigger	**Variable**
`condition` *Returns boolean*	• **object**: The current object being mapped. • **context**: Information related to the current operation, such as source and target. • **linkQualifier**: The link qualifier associated with the current sync operation. • **target**: Represents the target object. • **oldTarget**: Represents the target object prior to any changes. • **oldSource**: Available during UPDATE and DELETE operations performed through implicit sync. With implicit synchronization, the synchronization operation is triggered by a specific change to the source object. As such, implicit sync can populate the old value within the `oldSource` variable and pass it on to the sync engine. During reconciliation operations `oldSource` will be undefined. A reconciliation operation cannot populate the value of the `oldSource` variable as it has no awareness of the specific change to the source object. Reconciliation simply synchronizes the static source object to the target.
`transform` *Returns JSON object*	• **source**: Represents the source object. • **linkQualifier**: The link qualifier associated with the current sync operation. • **context**: Information related to the current sync operation.

policy object	
Trigger	**Variable**
`action` *Returns string OR JSON object*	• **source**: Represents the source object. • **target**: Represents the target object. • **sourceAction** (boolean): Indicates whether the action is being processed during the source or target synchronization phase (true if

policy object	
Trigger	**Variable**

	performed during a source synchronization, false if performed during a target synchronization).
	• **linkQualifier**: The link qualifier used for this operation (`default` if no other link qualifier is specified).
	• **context**: Information related to the current sync operation.
	• **recon**: Represents the reconciliation operation.
	• The `recon.actionParam` object contains information about the current reconciliation operation and includes the following variables:
	  • `reconId`: The ID of the reconciliation operation
	  • `mapping`: The mapping for which the reconciliation was performed, for example, `systemLdapAccounts_managedUser`.
	  • `situation`: The situation encountered, for example, AMBIGUOUS.
	  • `action`: The default action that would be used for this situation, if not for this script. The script being executed replaces the default action (and is used instead of any other named action).
	  • `sourceId`: The `_id` value of the source record.
	  • `linkQualifier`: The link qualifier used for that mapping, (`default` if no other link qualifier is specified).
	  • `ambiguousTargetIds`: An array of the target object IDs that were found in an AMBIGUOUS situation during correlation.
	  • `_action`: The synchronization action (only `performAction` is supported).

`postAction` *Returns JSON object*	• **source**: Represents the source object. • **target**: Represents the target object. • **action**: The sync action that was performed. • **sourceAction** (boolean): Indicates whether the action is being processed during the source or target synchronization phase (true if performed during a source synchronization, false if performed during a target synchronization). • **linkQualifier**: The link qualifier used for this operation (`default` if no other link qualifier is specified). • **reconId**: Represents the ID of the reconciliation. • **situation**: Represents the situation for this policy.

policy object	
Trigger	**Variable**
	• **context**: Information related to the current operation, such as source and target.

Script Triggers Defined in `router.json`

Trigger	Variable
onFailure	exception
onRequest	request
onResponse	response

Variables Available to Scripts in Custom Endpoints

All custom endpoint scripts have a `request` variable in their scope, which is a JSON object containing all information about the request. The parameters found in this object vary depending on the request method. The request may include headers, credentials, and the desired action. The request normally also includes the endpoint as well as the payload to be processed.

For more details about writing custom endpoint scripts, see "Custom Endpoint Scripts".

Variable	Variable Parameters
`request`	• **method**: The type of request, such as `query`, `create`, or `delete`. • **resourceName**: The name of the resource associated with the request. • **revision**: The revision number of the requested object. • **parameters**: JSON object mapping any additional parameters sent in the request. • **content**: The contents of the requested object. • **context**: Information related to the current request, such as client, end user, and routing. **Only available in `query` requests** • **pagedResultsCookie**: Represents the cookie used for `queryFilter` operations to track the results of a filtered query. • **pagedResultsOffset**: Specifies how many records to skip before returning a set of results. • **pageSize**: Specifies how many results to return per page.

Variable	Variable Parameters
	• **queryExpression**: A string containing a native query to query a system resource directly. • **queryId**: A string using the id of a predefined query object to return a specific set of results from a queried object. • **queryFilter**: A string with a common expression used to filter the results of a queried object. **Only available in `create` requests** • **newResourceId**: The ID of the new object. Only available in `create` requests.

Variables Available to Role Assignment Scripts

The optional `onAssignment` and `onUnassignment` event scripts specify what should happen to attributes that are affected by role assignments in the *Object Modeling Guide* when those assignments are applied to a user, or removed from a user.

These scripts have access to the following variables:

- `sourceObject`
- `targetObject`
- `existingTargetObject`
- `linkQualifier`

The standard assignment scripts, `replaceTarget.js`, `mergeWithTarget.js`, `removeFromTarget.js`, and `noOp.js` have access to all the variables in the previous list, as well as the following:

- `attributeName`
- `attributeValue`
- `attributesInfo`

Note

Role assignment scripts must always return `targetObject`, otherwise other scripts and code that occur downstream of your script will not work as expected.

The `augmentSecurityContext` Trigger

The `augmentSecurityContext` trigger, defined in `authentication.json`, can reference a script that is executed after successful authentication. Such scripts can populate the security context of the authenticated user. If the authenticated user is not found in the resource specified by `queryOnResource`, the `augmentSecurityContext` can provide the required authorization map.

Such scripts have access to the following bindings:

- `security` - includes the `authenticationId` and the `authorization` key, which includes the `moduleId`.

  The main purpose of an `augmentSecurityContext` script is to modify the `authorization` map that is part of this `security` binding. The authentication module determines the value of the `authenticationId`, and IDM attempts to populate the `authorization` map with the details that it finds, related to that `authenticationId` value. These details include the following:

  - `security.authorization.component` - the resource that contains the account (this will always will be the same as the value of `queryOnResource` by default).
  - `security.authorization.id` - the internal `_id` value that is associated with the account.
  - `security.authorization.roles` - any roles that were determined, either from reading the `userRoles` property of the account or from calculation.
  - `security.authorization.moduleId` - the authentication module responsible for performing the original authentication.

  You can use the `augmentSecurityContext` script to change any of these `authorization` values. The script can also add new values to the `authorization` map, which will be available for the lifetime of the session.

- `properties` - corresponds to the `properties` map of the related authentication module.
- `httpRequest` - a reference to the `Request` object that was responsible for handling the incoming HTTP request.

  This binding is useful to the augment script because it has access to all of the raw details from the HTTP request, such as the headers. The following code snippet shows how you can access a header using the `httpRequest` binding. This example accesses the `authToken` request header:

  ```
  httpRequest.getHeaders().getFirst('authToken').toString()
  ```

The `identityServer` Variable

IDM provides an additional variable, named `identityServer`, to scripts. You can use this variable in several ways. The `ScriptRegistryService`, described in "*Validate Scripts Over REST*", binds this variable to:

- `getProperty`

  Retrieves property information from system configuration files. Takes up to three parameters:

  - The name of the property you are requesting.
  - *(Optional)* The default result to return if the property wasn't set.
  - *(Optional)* Boolean to determine whether or not to use property substitution when getting the property. For more information about property substitution, see "Property Value Substitution" in the *Setup Guide*.

  Returns the first property found following the same order of precedence IDM uses to check for properties: environment variables, `system.properties`, `boot.properties`, then other configuration files. For more information, see "*Configure the Server*" in the *Setup Guide*.

  For example, you can retrieve the value of the `openidm.config.crypto.alias` property with the following code: `alias = identityServer.getProperty("openidm.config.crypto.alias", "true", true);`

- `getInstallLocation`

  Retrieves the IDM installation path, such as `/path/to/openidm`. May be superseded by an absolute path.

- `getProjectLocation`

  Retrieves the directory used when you started IDM. That directory includes configuration and script files for your project.

  For more information on the project location, see "Specify the Startup Configuration" in the *Installation Guide*.

- `getWorkingLocation`

  Retrieves the directory associated with database cache and audit logs. You can find `db/` and `audit/` subdirectories there.

  For more information on the working location, see "Specify the Startup Configuration" in the *Installation Guide*.

Appendix A. Router Configuration

The router service provides the uniform interface to all IDM objects: managed objects, system objects, configuration objects, and so on.

The router object as shown in `conf/router.json` defines an array of filter objects.

```
{
  "filters": [ filter object, ... ]
}
```

Filter Objects

The required filters array defines a list of filters to be processed on each router request. Filters are processed in the order in which they are specified in this array.

Filter objects are defined as follows.

```
{
  "pattern": string,
  "methods": [ string, ... ],
  "condition": script object,
  "onRequest": script object,
  "onResponse": script object,
  "onFailure": script object
}
```

pattern

string, optional

Specifies a regular expression pattern matching the JSON pointer of the object to trigger scripts. If not specified, all identifiers (including `null`) match. Pattern matching is done on the resource name, rather than on individual objects.

methods

array of strings, optional

One or more methods for which the script(s) should be triggered. Supported methods are: `"create"`, `"read"`, `"update"`, `"delete"`, `"patch"`, `"query"`, `"action"`. If not specified, all methods are matched.

condition

script object, optional

Specifies a script that is called first to determine if the script should be triggered. If the condition yields `"true"`, the other script(s) are executed. If no condition is specified, the script(s) are called unconditionally.

onRequest

script object, optional

Specifies a script to execute before the request is dispatched to the resource. If the script throws an exception, the method is not performed, and a client error response is provided.

onResponse

script object, optional

Specifies a script to execute after the request is successfully dispatched to the resource and a response is returned. Throwing an exception from this script does not undo the method already performed.

onFailure

script object, optional

Specifies a script to execute if the request resulted in an exception being thrown. Throwing an exception from this script does not undo the method already performed.

Pattern Matching in the `router.json` File

Pattern matching can minimize overhead in the router service. For example, the default `router.json` file includes instances of the `pattern` filter object, which limits script requests to specified methods and endpoints.

Based on the following code snippet, the router service would trigger the `policyFilter.js` script for `CREATE` and `UPDATE` calls to managed, system, and internal objects:

```
{
    "pattern" : "^(managed|system|internal)($|(/.+))",
    "onRequest" : {
        "type" : "text/javascript",
        "source" : "require('policyFilter').runFilter()"
    },
    "methods" : [
        "create",
        "update"
    ]
}
```

Without this `pattern`, IDM would apply the policy filter to additional objects such as the audit service, which may affect performance.

Script Execution Sequence

All `onRequest` and `onResponse` scripts are executed in sequence. First, the `onRequest` scripts are executed from the top down, then the `onResponse` scripts are executed from the bottom up.

```
client -> filter 1 onRequest -> filter 2 onRequest -> resource
client <- filter 1 onResponse <- filter 2 onResponse <- resource
```

The following sample `router.json` file shows the order in which the scripts would be executed:

```
{
    "filters" : [
        {
            "onRequest" : {
                "type" : "text/javascript",
                "source" : "require('router-authz').testAccess()"
            }
        },
        {
            "pattern" : "^managed/user",
            "methods" : [
                "read"
            ],
            "onRequest" : {
                "type" : "text/javascript",
                "source" : "console.log('requestFilter 1');"
            }
        },
        {
            "pattern" : "^managed/user",
            "methods" : [
                "read"
            ],
            "onResponse" : {
                "type" : "text/javascript",
                "source" : "console.log('responseFilter 1');"
            }
        },
```

```
        {
            "pattern" : "^managed/user",
            "methods" : [
                "read"
            ],
            "onRequest" : {
                "type" : "text/javascript",
                "source" : "console.log('requestFilter 2');"
            }
        },
        {
            "pattern" : "^managed/user",
            "methods" : [
                "read"
            ],
            "onResponse" : {
                "type" : "text/javascript",
                "source" : "console.log('responseFilter 2');"
            }
        }
    ]
}
```

Will produce a log like:

```
requestFilter 1
requestFilter 2
responseFilter 2
responseFilter 1
```

+ *Example*

The following example executes a script after a managed user object is created or updated:

```
{
    "filters": [
        {
            "pattern": "^managed/user",
            "methods": [
                "create",
                "update"
            ],
            "onResponse": {
                "type": "text/javascript",
                "file": "scripts/afterUpdateUser.js"
            }
        }
    ]
}
```

Script Scope

Scripts are provided with the following scope:

```
{
  "openidm": openidm-functions object,
  "request": resource-request object,
  "response": resource-response object,
  "exception": exception object
}
```

openidm

openidm-functions object

Provides access to IDM resources.

request

resource-request object

The resource-request context, which has one or more parent contexts. Provided in the scope of all scripts. For more information about the request context, see "*Request Context Chain*".

response

resource-response object

The response to the resource-request. Only provided in the scope of the `"onResponse"` script.

exception

exception object

The exception value that was thrown as a result of processing the request. Only provided in the scope of the `"onFailure"` script. An exception object is defined as:

```
{
  "code": integer,
  "reason": string,
  "message": string,
  "detail": string
}
```

code

integer

The numeric HTTP code of the exception.

reason

string

The short reason phrase of the exception.

message

string

A brief message describing the exception.

detail

(optional), string

A detailed description of the exception, in structured JSON format, suitable for programmatic evaluation.

Appendix B. Scripting Function Reference

Functions (access to managed objects, system objects, and configuration objects) within IDM are accessible to scripts via the `openidm` object, which is included in the top-level scope provided to each script.

The script engine supports the following functions:

+ *openidm.create(resourceName, newResourceId, content, params, fields)*

This function creates a new resource object.

Parameters

resourceName

string

The container in which the object will be created, for example, `managed/user`.

newResourceId

string

The identifier of the object to be created, if the client is supplying the ID. If the server should generate the ID, pass null here.

content

JSON object

The content of the object to be created.

params

JSON object (optional)

Additional parameters that are passed to the create request.

fields

JSON array (optional)

An array of the fields that should be returned in the result. The list of fields can include wild cards, such as `*` or `*_ref`. If no fields are specified, the entire new object is returned.

Returns

The created resource object.

Throws

An exception is thrown if the object could not be created.

Example

```
openidm.create("managed/user", ID, JSON object);
```

+ *openidm.patch(resourceName, rev, value, params, fields)*

This function performs a partial modification of a managed or system object. Unlike the `update` function, only the modified attributes are provided, not the entire object.

Parameters

resourceName

string

The full path to the object being updated, including the ID.

rev

string

The revision of the object to be updated. Use `null` if the object is not subject to revision control, or if you want to skip the revision check and update the object, regardless of the revision.

value

An array of one or more JSON objects

The value of the modifications to be applied to the object. The patch set includes the operation type, the field to be changed, and the new values. A PATCH request can `add`, `remove`, `replace`, or `increment` an attribute value.

A `remove` operation removes a property if the value of that property equals the specified value, or if no value is specified in the request. The following example `value` removes the `marital_status` property from the object, *if* the value of that property is `single`:

```
[
    {
        "operation": "remove",
        "field": "marital_status",
        "value": "single"
    }
]
```

For fields whose value is an array, it's not necessary to know the position of the value in the array, as long as you specify the full object. If the full object is found in the array, that value is removed. The following example removes user adonnelly from bjensen's `reports`:

```
{
    "operation": "remove",
    "field": "/manager",
    "value": {
      "_ref": "managed/user/adonnelly",
      "_refResourceCollection": "managed/user",
      "_refResourceId": "adonnelly",
      "_refProperties": {
        "_id": "ed6620e4-98ba-410c-abc0-e06dc1be7aa7",
        "_rev": "000000008815942b"
      }
    }
}
```

If an invalid value is specified (that is a value that does not exist for that property in the current object) the patch request is silently ignored.

A `replace` operation replaces an existing value, or adds a value if no value exists.

params

JSON object (optional)

Additional parameters that are passed to the patch request.

fields

JSON array (optional)

An array of the fields that should be returned in the result. The list of fields can include wild cards, such as `*` or `*_ref`. If no fields are specified, the entire new object is returned.

Returns

The modified resource object.

Throws

An exception is thrown if the object could not be updated.

Examples

Patching an object to add a value to an array:

```
openidm.patch("managed/role/" + role._id, null, [{"operation":"add", "field":"/
members/-", "value": {"_ref":"managed/user/" + user._id}}]);
```

Patching an object to remove an existing property:

```
openidm.patch("managed/user/" + user._id, null,
 [{"operation":"remove", "field":"marital_status", "value":"single"}]);
```

Patching an object to replace a field value:

```
openidm.patch("managed/user/" + user._id, null, [{"operation":"replace", "field":"/
password", "value":"Passw0rd"}]);
```

Patching an object to increment an integer value:

```
openidm.patch("managed/user/" + user._id, null, [{"operation":"increment","field":"/
age","value":1}]);
```

+ *openidm.read(resourceName, params, fields)*

This function reads and returns a resource object.

Parameters

resourceName

string

The full path to the object to be read, including the ID.

params

JSON object (optional)

The parameters that are passed to the read request. Generally, no additional parameters are passed to a read request, but this might differ, depending on the request. If you need to specify a list of `fields` as a third parameter, and you have no additional `params` to pass, you must pass `null` here. Otherwise, you simply omit both parameters.

fields

JSON array (optional)

An array of the fields that should be returned in the result. The list of fields can include wild cards, such as `*` or `*_ref`. If no fields are specified, the entire object is returned.

Returns

The resource object, or `null` if not found.

Example

```
openidm.read("managed/user/"+userId, null, ["*", "manager"]);
```

+ *openidm.update(resourceName, rev, value, params, fields)*

This function updates an entire resource object.

Parameters

id

string

The complete path to the object to be updated, including its ID.

rev

string

The revision of the object to be updated. Use `null` if the object is not subject to revision control, or if you want to skip the revision check and update the object, regardless of the revision.

value

object

The complete replacement object.

params

JSON object (optional)

The parameters that are passed to the update request.

fields

JSON array (optional)

An array of the fields that should be returned in the result. The list of fields can include wild cards, such as `*` or `*_ref`. If no fields are specified, the entire object is returned.

Returns

The modified resource object.

Throws

An exception is thrown if the object could not be updated.

Example

In this example, the managed user entry is read (with an `openidm.read`, the user entry that has been read is updated with a new description, and the entire updated object is replaced with the new value.

```
var user_read = openidm.read('managed/user/' + source._id);
user_read['description'] = 'The entry has been updated';
openidm.update('managed/user/' + source._id, null, user_read);
```

+ *openidm.delete(resourceName, rev, params, fields)*

This function deletes a resource object.

Parameters

resourceName

string

The complete path to the to be deleted, including its ID.

rev

string

The revision of the object to be deleted. Use `null` if the object is not subject to revision control, or if you want to skip the revision check and delete the object, regardless of the revision.

params

JSON object (optional)

The parameters that are passed to the delete request.

fields

JSON array (optional)

An array of the fields that should be returned in the result. The list of fields can include wild cards, such as `*` or `*_ref`. If no fields are specified, the entire object is returned.

Returns

Returns the deleted object if successful.

Throws

An exception is thrown if the object could not be deleted.

Example

```
openidm.delete('managed/user/'+ user._id, user._rev);
```

+ *openidm.query(resourceName, params, fields)*

This function performs a query on the specified resource object. For more information, see "Construct Queries" in the *Object Modeling Guide*.

Parameters

resourceName

string

The resource object on which the query should be performed, for example, `"managed/user"`, or `"system/ldap/account"`.

params

JSON object

The parameters that are passed to the query (`_queryFilter`, or `_queryId`). Additional parameters passed to the query will differ, depending on the query.

Certain common parameters can be passed to the query to restrict the query results. The following sample query passes paging parameters and sort keys to the query.

```
reconAudit = openidm.query("audit/recon", {
    "_queryFilter": queryFilter,
    "_pageSize": limit,
    "_pagedResultsOffset": offset,
    "_pagedResultsCookie": string,
    "_sortKeys": "-timestamp"
});
```

For more information about `_queryFilter` syntax, see "Common Filter Expressions" in the *Object Modeling Guide*. For more information about paging, see "Page Query Results" in the *Object Modeling Guide*.

fields

list

A list of the fields that should be returned in the result. The list of fields can include wild cards, such as `*` or `*_ref`. The following example returns only the `userName` and `_id` fields:

```
openidm.query("managed/user", { "_queryFilter": "/userName sw \"user.1\""}, ["userName", "_id"]);
```

This parameter is particularly useful in enabling you to return the response from a query without including intermediary code to massage it into the right format.

Fields are specified as JSON pointers.

Returns

The result of the query. A query result includes the following parameters:

query-time-ms

(For JDBC repositories only) the time, in milliseconds, that IDM took to process the query.

result

The list of entries retrieved by the query. The result includes the properties that were requested in the query.

The following example shows the result of a custom query that requests the ID, user name, and email address of all managed users in the repository.

```
{
  "result": [
    {
      "_id": "9dce06d4-2fc1-4830-a92b-bd35c2f6bcbb",
      "_rev": "00000000a059dc9f",
      "userName": "bjensen",
      "mail": "bjensen@example.com"
    },
    {
      "_id": "42f8a60e-2019-4110-a10d-7231c3578e2b",
      "_rev": "00000000d84ade1c",
      "userName": "scarter",
      "mail": "scarter@example.com"
    }
  ],
  "resultCount": 2,
  "pagedResultsCookie": null,
  "totalPagedResultsPolicy": "NONE",
  "totalPagedResults": -1,
  "remainingPagedResults": -1
}
```


Throws

An exception is thrown if the given query could not be processed.

Examples

The following sample query uses a `_queryFilter` to query the managed user repository:

```
openidm.query("managed/user", {'_queryFilter': userIdPropertyName + ' eq "' +
 security.authenticationId  + '"'});
```

The following sample query references the `for-userName` query, defined in the repository configuration, to query the managed user repository:

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

+ *openidm.action(resource, actionName, content, params, fields)*

This function performs an action on the specified resource object. The `resource` and `actionName` are required. All other parameters are optional.

Parameters

resource

string

The resource that the function acts upon, for example, `managed/user`.

actionName

string

The action to execute. Actions are used to represent functionality that is not covered by the standard methods for a resource (create, read, update, delete, patch, or query). In general, you should not use the `openidm.action` function for create, read, update, patch, delete or query operations. Instead, use the corresponding function specific to the operation (for example, `openidm.create`).

Using the operation-specific functions lets you benefit from the well-defined REST API, which follows the same pattern as all other standard resources in the system. Using the REST API enhances usability for your own API, and enforces the established patterns.

IDM-defined resources support a fixed set of actions. For user-defined resources (scriptable endpoints) you can implement whatever actions you require.

Supported Actions Per Resource

The following list outlines the supported actions for each resource or endpoint. The actions listed here are also supported over the REST interface.

Actions supported on the `authentication` endpoint (`authentication/*`)

reauthenticate

Actions supported on the configuration resource (`config/`)

No action parameter applies.

Actions supported on custom endpoints

Custom endpoints enable you to run arbitrary scripts through the REST URI, and are routed at `endpoint/name`, where name generally describes the purpose of the endpoint. For more information on custom endpoints, see "*Create Custom Endpoints to Launch Scripts*". You can implement whatever actions you require on a custom endpoint. IDM uses custom endpoints in its workflow implementation. Those endpoints, and their actions are as follows:

`endpoint/getprocessforuser` - `create, complete`
`endpoint/gettasksview` - `create, complete`

Actions supported on the `external` endpoint

- `external/email` - `send`, for example:

```
{
    emailParams = {
        "from" : 'admin@example.com',
        "to" : user.mail,
        "subject" : 'Password expiry notification',
        "type" : 'text/plain',
        "body" : 'Your password will expire soon. Please change it!'
    }
    openidm.action("external/email", "send",  emailParams);
}
```

- `external/rest` - `call`, for example:

```
openidm.action("external/rest", "call", params);
```

Actions supported on the `info` endpoint (`info/*`)

No action parameter applies.

Actions supported on managed resources (`managed/*`)

patch, triggerSyncCheck

Actions supported on the policy resource (`policy`)

validateObject, validateProperty

For example:

```
openidm.action("policy/" + fullResourcePath, "validateObject", request.content, { "external"
 : "true" });
```

Actions supported on the reconciliation resource (`recon`)

recon, reconById, cancel

For example:

```
openidm.action("recon", "cancel", content, params);
```

Actions supported on the repository (`repo`)

command

For example:

```
var r, command = {
    "commandId": "purge-by-recon-number-of",
    "numberOf": numOfRecons,
    "includeMapping": includeMapping,
    "excludeMapping": excludeMapping
};
r = openidm.action("repo/audit/recon", "command", {}, command);
```

Actions supported on the script resource (`script`)

eval

For example:

```
openidm.action("script", "eval", getConfig(scriptConfig), {});
```

Actions supported on the synchronization resource (`sync`)

getLinkedResources, notifyCreate, notifyDelete, notifyUpdate, performAction

For example:

```
openidm.action('sync', 'performAction', content, params);
```

Actions supported on system resources (`system/*`)

availableConnectors, createCoreConfig, createFullConfig, test, testConfig, liveSync, authenticate, script

For example:

```
openidm.action("system/ldap/account", "authenticate", {"username" : "bjensen", "password"
 : "Passw0rd"});
```

Actions supported on the task scanner resource (`taskscanner`)

execute, cancel

Actions supported on the workflow resource (`workflow/*`)

On `workflow/processdefinition` create, complete

On `workflow/processinstance` create, complete

For example:

```
var params = {
    "_key":"contractorOnboarding"
};
openidm.action('workflow/processinstance', 'create', params);
```

On `workflow/taskinstance` claim, create, complete

For example:

```
var params = {
    "userId":"manager1"
};
openidm.action('workflow/taskinstance/15', 'claim', params);
```

content

object

Content given to the action for processing.

params

object (optional)

Additional parameters passed to the script. The `params` object must be a set of simple key:value pairs, and cannot include complex values. The parameters must map directly to URL variables, which take the form `name1=val1&name2=val2&...`.

fields

JSON array (optional)

An array of the fields that should be returned in the result. The list of fields can include wild cards, such as `*` or `*_ref`. If no fields are specified, the entire object is returned.

Returns

The result of the action may be `null`.

Throws

If the action cannot be executed, an exception is thrown.

+ *openidm.encrypt(value, cipher, alias)*

This function encrypts a value.

Parameters

value

any

The value to be encrypted.

cipher

string

The cipher with which to encrypt the value, using the form "algorithm/mode/padding" or just "algorithm". Example: `AES/CBC/PKCS5Padding`.

alias

string

The key alias in the keystore, such as `openidm-sym-default` (deprecated) or a purpose defined in the `secrets.json` file, such as `idm.password.encryption`.

Returns

The value, encrypted with the specified cipher and key.

Throws

An exception is thrown if the object could not be encrypted.

+ *openidm.decrypt(value)*

This function decrypts a value.

Parameters

value

object

The value to be decrypted.

Returns

A deep copy of the value, with any encrypted value decrypted.

Throws

An exception is thrown if the object could not be decrypted for any reason. An error is thrown if the value is passed in as a string - it must be passed in an object.

+ *openidm.isEncrypted(object)*

This function determines if a value is encrypted.

Parameters

object to check

any

The object whose value should be checked to determine if it is encrypted.

Returns

Boolean, `true` if the value is encrypted, and `false` if it is not encrypted.

Throws

An exception is thrown if the server is unable to detect whether the value is encrypted, for any reason.

+ *openidm.hash(value, algorithm)*

This function calculates a value using a salted hash algorithm.

Parameters

value

any

The value to be hashed.

algorithm

string (optional)

The algorithm with which to hash the value. Example: `SHA-512`. If no algorithm is provided, a `null` value must be passed, and the algorithm defaults to SHA-256. For a list of supported hash algorithms, see "Encoding Attribute Values by Using Salted Hash Algorithms" in the *Security Guide*.

Returns

The value, calculated with the specified hash algorithm.

Throws

An exception is thrown if the object could not be hashed for any reason.

+ *openidm.isHashed(value)*

This function detects whether a value has been calculated with a salted hash algorithm.

Parameters

value

any

The value to be reviewed.

Returns

Boolean, `true` if the value is hashed, and `false` otherwise.

Throws

An exception is thrown if the server is unable to detect whether the value is hashed, for any reason.

+ *openidm.matches(string, value)*

This function detects whether a string, when hashed, matches an existing hashed value.

Parameters

string

any

A string to be hashed.

value

any

A hashed value to compare to the string.

Returns

Boolean, `true` if the hash of the string matches the hashed value, and `false` otherwise.

Throws

An exception is thrown if the string could not be hashed.

Log Functions

IDM also provides a `logger` object to access the Simple Logging Facade for Java (SLF4J) facilities. The following code shows an example of the `logger` object.

```
logger.info("Parameters passed in: {} {} {}", param1, param2, param3);
```

To set the log level for JavaScript scripts, add the following property to your project's `conf/logging.properties` file:

```
org.forgerock.openidm.script.javascript.JavaScript.level
```

The level can be one of `SEVERE` (highest value), `WARNING, INFO, CONFIG, FINE, FINER`, or `FINEST` (lowest value). For example:

```
org.forgerock.openidm.script.javascript.JavaScript.level=WARNING
```

In addition, JavaScript has a useful logging function named `console.log()`. This function provides an easy way to dump data to the IDM standard output (usually the same output as the OSGi console). The function works well with the JavaScript built-in function `JSON.stringify` and provides fine-grained details about any given object. For example, the following line will print a formatted JSON structure that represents the HTTP request details to STDOUT.

```
console.log(JSON.stringify(context.http, null, 4));
```

Note

These logging functions apply only to JavaScript scripts. To use the logging functions in Groovy scripts, the following lines must be added to the Groovy scripts:

```
import org.slf4j.*;
logger = LoggerFactory.getLogger('logger');
```

The script engine supports the following log functions:

+ *logger.debug(string message, object... params)*

Logs a message at DEBUG level.

Parameters

message

string

The message format to log. Params replace `{}` in your message.

params

object

Arguments to include in the message.

Returns

A `null` value if successful.

Throws

An exception is thrown if the message could not be logged.

+ *logger.error(string message, object... params)*

Logs a message at ERROR level.

Parameters

message

string

The message format to log. Params replace `{}` in your message.

params

object

Arguments to include in the message.

Returns

A `null` value if successful.

Throws

An exception is thrown if the message could not be logged.

+ *logger.info(string message, object... params)*

Logs a message at INFO level.

Parameters

message

string

The message format to log. Params replace `{}` in your message.

params

object

Arguments to include in the message.

Returns

A `null` value if successful.

Throws

An exception is thrown if the message could not be logged.

+ *logger.trace(string message, object... params)*

Logs a message at TRACE level.

Parameters

message

string

The message format to log. Params replace `{}` in your message.

params

object

Arguments to include in the message.

Returns

A `null` value if successful.

Throws

An exception is thrown if the message could not be logged.

+ *logger.warn(string message, object... params)*

Logs a message at WARN level.

Parameters

message

string

The message format to log. Params replace `{}` in your message.

params

object

Arguments to include in the message.

Returns

A `null` value if successful.

Throws

An exception is thrown if the message could not be logged.

IDM Glossary

correlation query	A correlation query specifies an expression that matches existing entries in a source repository to one or more entries in a target repository. A correlation query might be built with a script, but it is not the same as a correlation script. For more information, see "*Correlating Source Objects With Existing Target Objects*" in the *Synchronization Guide*.
correlation script	A correlation script matches existing entries in a source repository, and returns the IDs of one or more matching entries on a target repository. While it skips the intermediate step associated with a `correlation query`, a correlation script can be relatively complex, based on the operations of the script.
entitlement	An entitlement is a collection of attributes that can be added to a user entry via roles. As such, it is a specialized type of `assignment`. A user or device with an entitlement gets access rights to specified resources. An entitlement is a property of a managed object.
JCE	Java Cryptographic Extension, which is part of the Java Cryptography Architecture, provides a framework for encryption, key generation, and digital signatures.
JSON	JavaScript Object Notation, a lightweight data interchange format based on a subset of JavaScript syntax. For more information, see the JSON site.
JSON Pointer	A JSON Pointer defines a string syntax for identifying a specific value within a JSON document. For information about JSON Pointer syntax, see the JSON Pointer RFC.

JWT	JSON Web Token. As noted in the JSON Web Token draft IETF Memo, "JSON Web Token (JWT) is a compact URL-safe means of representing claims to be transferred between two parties." For IDM, the JWT is associated with the `JWT_SESSION` authentication module.
managed object	An object that represents the identity-related data managed by IDM. Managed objects are configurable, JSON-based data structures that IDM stores in its pluggable repository. The default configuration of a managed object is that of a user, but you can define any kind of managed object, for example, groups or roles.
mapping	A policy that is defined between a source object and a target object during reconciliation or synchronization. A mapping can also define a trigger for validation, customization, filtering, and transformation of source and target objects.
OSGi	A module system and service platform for the Java programming language that implements a complete and dynamic component model. For a good introduction, see the OSGi site. Currently only the Apache Felix container is supported.
reconciliation	During reconciliation, comparisons are made between managed objects and objects on source or target systems. Reconciliation can result in one or more specified actions, including, but not limited to, synchronization.
resource	An external system, database, directory server, or other source of identity data to be managed and audited by the identity management system.
REST	Representational State Transfer. A software architecture style for exposing resources, using the technologies and protocols of the World Wide Web. REST describes how distributed data objects, or resources, can be defined and addressed.
role	IDM distinguishes between two distinct role types - provisioning roles and authorization roles. For more information, see "Managed Roles" in the *Object Modeling Guide*.
source object	In the context of reconciliation, a source object is a data object on the source system, that IDM scans before attempting to find a corresponding object on the target system. Depending on the defined mapping, IDM then adjusts the object on the target system (target object).
synchronization	The synchronization process creates, updates, or deletes objects on a target system, based on the defined mappings from the source system. Synchronization can be scheduled or on demand.

system object	A pluggable representation of an object on an external system. For example, a user entry that is stored in an external LDAP directory is represented as a system object in IDM for the period during which IDM requires access to that entry. System objects follow the same RESTful resource-based design principles as managed objects.
target object	In the context of reconciliation, a target object is a data object on the target system, that IDM scans after locating its corresponding object on the source system. Depending on the defined mapping, IDM then adjusts the target object to match the corresponding source object.