Schedules Guide

/ ForgeRock Identity Management 7

Latest update: 7.0.2

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

Abstract

Guide to configuring schedules and scanning tasks.

Table of Contents

Overview iv
1. Schedule Tasks and Events 1
    Configure the Scheduler Service 1
    Configure Schedules 2
    Schedules and Daylight Savings Time 6
    Configure Persistent Schedules 7
    Schedule Examples 8
    Manage Schedules Over REST 8
    Manage Schedules Using the Admin UI 23
2. Scan Data to Trigger Tasks 24
    Configure the Task Scanner 24
    Manage Scanning Tasks Over REST 28
    Manage Scanning Tasks Using the UI 33
IDM Glossary 35

Overview

This guide covers schedules and scanning tasks.

Quick Start

	
Scheduling	Task Scanner
Schedule tasks and events.	Scan data to trigger tasks.

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

Schedule Tasks and Events

The scheduler service lets you schedule reconciliation and synchronization tasks, trigger scripts, collect and run reports, trigger workflows, and perform custom logging.

The service depends on the Quartz Scheduler (bundled with IDM), and supports Quartz simple triggers and cron triggers. Use the trigger type that suits your scheduling requirements. For more information, see the Quartz documentation on SimpleTriggers and CronTriggers.

By default, IDM picks up changes to scheduled tasks and events dynamically, during initialization and also at runtime. For more information, see "Configuration Changes" in the *Setup Guide*.

In addition to the fine-grained scheduling facility, you can perform a scheduled batch scan for a specified date in IDM data, and then automatically run a task when this date is reached. For more information, see "*Scan Data to Trigger Tasks*".

- "Configure the Scheduler Service"
- "Configure Schedules"
- "Schedules and Daylight Savings Time"
- "Configure Persistent Schedules"
- "Schedule Examples"
- "Manage Schedules Over REST"
- "Manage Schedules Using the Admin UI"

Configure the Scheduler Service

There is a distinction between the configuration of the scheduler service, and the configuration of individual scheduled tasks and events. The scheduler service is configured in your project's `conf/scheduler.json` file. This file has the following format:

```
{
    "threadPool" : {
        "threadCount" : 10
    },
    "scheduler" : {
        "executePersistentSchedules" : {
            "$bool" : "&{openidm.scheduler.execute.persistent.schedules}"
        }
    }
}
```

The properties in the `scheduler.json` file relate to the configuration of the Quartz Scheduler:

- `threadCount` specifies the maximum number of threads that are available for running scheduled tasks concurrently.
- `executePersistentSchedules` allows you to disable persistent schedules for a specific node. If this parameter is set to `false`, the Scheduler Service will support the management of persistent schedules (CRUD operations) but it will not run any persistent schedules. The value of this property can be a string or boolean. Its default value (set in `resolver/boot.properties`) is `true`.
- `advancedProperties` (optional) enables you to configure additional properties for the Quartz Scheduler.

For details of all the configurable properties for the Quartz Scheduler, see the *Quartz Scheduler Configuration Reference*.

Configure Schedules

To schedule tasks and events, select Configure > Schedules > Add Schedule in the Admin UI, or create schedule configuration files in your project's `conf` directory. By convention, IDM uses file names of the form `schedule-`*`schedule-name`*`.json`, where *schedule-name* is a logical name for the scheduled operation, for example, `schedule-reconcile_systemCsvAccounts_managedUser.json`. There are several example schedule configuration files in the `openidm/samples/example-configurations/schedules` directory.

Each schedule configuration file has the following format:

```
{
 "enabled"             : boolean,
 "persisted"           : boolean,
 "recoverable"         : boolean,
 "concurrentExecution" : boolean,
 "type"                : "simple | cron",
 "repeatInterval"      : (optional) integer,
 "repeatCount"         : (optional) integer,
 "startTime"           : "(optional) time",
 "endTime"             : "(optional) time",
 "schedule"            : "cron expression",
 "misfirePolicy"       : "optional, string",
 "invokeService"       : "service identifier",
 "invokeContext"       : "service specific context info",
 "invokeLogLevel"      : "(optional) level"
}
```

The schedule configuration properties are defined as follows:

`enabled`

Set to `true` to enable the schedule. When this property is `false`, IDM considers the schedule configuration dormant, and does not allow it to be triggered or launched.

If you want to retain a schedule configuration, but do not want it used, set `enabled` to `false` for task and event schedulers, instead of changing the configuration or **cron** expressions.

`persisted` **(optional)**

Specifies whether the schedule state should be persisted or stored *only* in RAM. Boolean (`true` or `false`), `false` by default.

In a clustered environment, this property must be set to `true` to have the schedule fire only once across the cluster. For more information, see "Configure Persistent Schedules".

Note

If the schedule is stored only in RAM, the schedule will be lost when IDM is restarted.

`recoverable` **(optional)**

Specifies whether jobs that have failed mid-execution (as a result of a JVM crash or otherwise unexpected termination) should be recovered. Boolean (`true` or `false`), `false` by default.

`concurrentExecution`

Specifies whether multiple instances of the same schedule can run concurrently. Boolean (`true` or `false`), `false` by default. Multiple instances of the same schedule cannot run concurrently by default. This setting prevents a new scheduled task from being launched before the same previously launched task has completed. For example, under normal circumstances you would want a liveSync operation to complete before the same operation was launched again. To enable

multiple schedules to run concurrently, set this parameter to `true`. The behavior of missed scheduled tasks is governed by the `misfirePolicy`.

`type`

The trigger type, either `simple` or `cron`.

To decide which trigger type to use, see the Quartz documentation on SimpleTriggers and CronTriggers.

`repeatCount`

Used only for simple triggers (`"type" : "simple"`).

The number of times the schedule must be repeated. The repeat count can be zero, a positive integer, or -1. A value of -1 indicates that the schedule should repeat indefinitely.

If you do not specify a repeat count, the value defaults to -1.

`repeatInterval`

Used only for simple triggers (`"type" : "simple"`).

Specifies the interval, in milliseconds, between trigger firings. The repeat interval must be zero or a positive long value. If you set the repeat interval to zero, the scheduler will trigger `repeatCount` firings concurrently (or as close to concurrently as possible).

If you do not specify a repeat interval, the value defaults to 0.

`startTime` (optional)

This parameter starts the schedule at some time in the future. If the parameter is omitted, empty, or set to a time in the past, the task or event is scheduled to start immediately.

Use ISO 8601 format to specify times and dates (`yyyy-MM-dd'T'HH:mm:ss`).

To specify a time zone, include the time zone at the end of the `startTime`, in the format `+|-hh:mm`, for example `2017-10-31T15:53:00+05:00`. If you specify both a `startTime` and an `endTime`, they must have the same time zone.

`endTime` (optional)

Specifies when the schedule must end, in ISO 8601 format (`yyyy-MM-dd'T'HH:mm:ss+|-hh:mm`.

`schedule`

Used only for cron triggers (`"type" : "cron"`).

Takes **cron** expression syntax. For more information, see the *CronTrigger Tutorial* and *Lesson 6: CronTrigger*.

`misfirePolicy`

For persistent schedules, this optional parameter specifies the behavior if the scheduled task is missed, for some reason. Possible values are as follows:

- `fireAndProceed`. The first run of a missed schedule is immediately launched when the server is back online. Subsequent runs are discarded. After this, the normal schedule is resumed.
- `doNothing`. All missed schedules are discarded and the normal schedule is resumed when the server is back online.

`invokeService`

Defines the type of scheduled event or action. The value of this parameter can be one of the following:

- `sync` for reconciliation
- `provisioner` for liveSync
- `script` to call some other scheduled operation defined in a script
- `taskScanner` to define a scheduled task that queries a set of objects. For more information, see "*Scan Data to Trigger Tasks*".

`invokeContext`

Specifies contextual information, depending on the type of scheduled event (the value of the `invokeService` parameter).

The following example invokes reconciliation:

```
{
    "invokeService": "sync",
    "invokeContext": {
        "action": "reconcile",
        "mapping": "systemLdapAccount_managedUser"
    }
}
```

The following example invokes a liveSync operation:

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

For scheduled liveSync tasks, the `source` property follows IDM's convention for a pointer to an external resource object and takes the form `system/resource-name/object-type`.

The following example invokes a script, which prints the node ID performing the scheduled job and the time to the console. A similar sample schedule is provided in `schedule-script.json` in the `/path/to/openidm/samples/example-configurations/schedules` directory.

```
{
    "enabled" : true,
    "type": "simple",
    "repeatInterval": 3600000,
    "persisted" : true,
    "concurrentExecution" : false,
    "invokeService": "script",
    "invokeContext": {
        "script" : {
            "type" : "text/javascript",
            "source" : "java.lang.System.out.println('Job executing on ' +
 identityServer.getProperty('openidm.node.id') + ' at: ' + java.lang.System.currentTimeMillis());"
        }
    }
}
```

Note that these are sample configurations only. Your own schedule configuration will differ according to your specific requirements.

`invokeLogLevel` (optional)

Specifies the level at which the invocation will be logged. Particularly for schedules that run very frequently, such as liveSync, the scheduled task can generate significant output to the log file, and you should adjust the log level accordingly. The default schedule log level is `info`. The value can be set to any one of the SLF4J log levels:

- `trace`
- `debug`
- `info`
- `warn`
- `error`
- `fatal`

Schedules and Daylight Savings Time

The scheduler service supports Quartz cron triggers and simple triggers. Cron triggers schedule jobs to fire at specific times with respect to a calendar (rather than every *N* milliseconds). This scheduling can cause issues when clocks change for daylight savings time (DST) if the trigger time falls around the clock change time in your specific time zone.

Depending on the trigger schedule, and on the daylight event, the trigger might be skipped or might appear not to fire for a short period. This interruption can be particularly problematic for liveSync

where schedules execute continuously. In this case, the time change (for example, from 02:00 back to 01:00) causes an hour break between each liveSync execution.

To prevent DST from having an impact on your schedules, use simple triggers instead of cron triggers.

Configure Persistent Schedules

By default, scheduling information, such as schedule state and details of the schedule run, is stored in RAM. This means that such information is lost when the server is rebooted. The schedule configuration itself (defined in your project's `conf/schedule-schedule-name.json` file) is not lost when the server is shut down, and normal scheduling continues when the server is restarted. However, there are no details of missed schedule runs that should have occurred during the period the server was unavailable.

You can configure schedules to be persistent, which means that the scheduling information is stored in the internal repository rather than in RAM. With persistent schedules, scheduling information is retained when the server is shut down. Any previously scheduled jobs can be rescheduled automatically when the server is restarted.

Persistent schedules also enable you to manage scheduling across a cluster (multiple IDM instances). When scheduling is persistent, a particular schedule will be launched only once across the cluster, rather than once on every instance. For example, if your deployment includes a cluster of nodes for high availability, you can use persistent scheduling to start a reconciliation operation on only one node in the cluster, instead of starting several competing reconciliation operations on each node.

Important

Persistent schedules rely on timestamps. In a deployment where IDM instances run on separate machines, you *must* synchronize the system clocks of these machines using a time synchronization service that runs regularly. The clocks of all machines involved in persistent scheduling must be within one second of each other. For information on how you can achieve this using the Network Time Protocol (NTP) daemon, see the NTP RFC.

To configure persistent schedules, set `persisted` to `true` in the schedule configuration file (`schedule-schedule-name.json`).

If the server is down when a scheduled task was set to occur, one or more runs of that schedule might be missed. To specify what action should be taken if schedules are missed, set the `misfirePolicy` in the schedule configuration file. The `misfirePolicy` determines what IDM should do if scheduled tasks are missed. Possible values are as follows:

- `fireAndProceed`. The first run of a missed schedule is immediately implemented when the server is back online. Subsequent runs are discarded. After this, the normal schedule is resumed.
- `doNothing`. All missed schedules are discarded and the normal schedule is resumed when the server is back online.

Schedule Examples

The following example shows a schedule for reconciliation that is not enabled. When the schedule is enabled (`"enabled" : true,`), reconciliation runs every 30 minutes (1800000 milliseconds), and repeats indefinitely:

```
{
    "enabled": false,
    "persisted": true,
    "type": "simple",
    "repeatInterval": 1800000,
    "invokeService": "sync",
    "invokeContext": {
        "action": "reconcile",
        "mapping": "systemLdapAccounts_managedUser"
    }
}
```

The following example shows a schedule for liveSync enabled to run every 15 seconds, repeating indefinitely. Note that the schedule is persisted, that is, stored in the repository rather than in memory. If one or more liveSync runs are missed, as a result of the server being unavailable, the first run of the liveSync operation is implemented when the server is back online. Subsequent runs are discarded. After this, the normal schedule is resumed:

```
{
    "enabled": true,
    "persisted": true,
    "misfirePolicy" : "fireAndProceed",
    "type": "simple",
    "repeatInterval": 15000,
    "invokeService": "provisioner",
    "invokeContext": {
        "action": "liveSync",
        "source": "system/ldap/account"
    }
}
```

Manage Schedules Over REST

The scheduler service is exposed under the `/openidm/scheduler` context path. Within this context path, the defined scheduled jobs are accessible at `/openidm/scheduler/job`. A job is the actual task that is run. Each job contains a *trigger* that starts the job. The trigger defines the schedule according to which the job is executed. You can read and query the existing triggers on the `/openidm/scheduler/trigger` context path.

The following examples show how schedules are validated, created, read, queried, updated, and deleted, over REST, by using the scheduler service. The examples also show how to pause and resume scheduled jobs, when an instance is placed in maintenance mode. For information about placing a server in maintenance mode, see "*Place a Server in Maintenance Mode*" in the *Upgrade Guide*.

Note

When you configure schedules over REST, changes made to the schedules are not pushed back into the configuration service. Managing schedules by using the `/openidm/scheduler/job` context path essentially bypasses the configuration service and sends the request directly to the scheduler.

If you need to perform an operation on a schedule that was created by using the configuration service (by placing a schedule file in the `conf/` directory), you must direct your request to the `/openidm/config` context path, and not to the `/openidm/scheduler/job` context path.

PATCH operations are not supported on the `scheduler` context path. To patch a schedule, use the `config` context path.

+ *Validate Cron Trigger Expressions*

Schedules are defined using Quartz cron or simple triggers. If you use a cron trigger, you can validate your cron expression by sending the expression as a JSON object to the `scheduler` context path:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=2.0" \
--request POST \
--data '{
    "cronExpression": "0 0/1 * * * ?"
}' \
"http://localhost:8080/openidm/scheduler/job?_action=validateQuartzCronExpression"
{
  "valid": true
}
```

+ *Define a Schedule*

To define a new schedule, send a PUT or POST request to the `scheduler/job` context path with the details of the schedule in the JSON payload. A PUT request allows you to specify the ID of the schedule. A POST request assigns an ID automatically.

The following example uses a PUT request to create a schedule that fires a script (`script/testlog.js`) every second. The example assumes that the script exists in the specified location. The schedule configuration is as described in "Configure Schedules":

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=2.0" \
--request PUT \
--data '{
    "enabled": true,
    "type": "cron",
    "schedule": "0/1 * * * * ?",
```

```
    "persisted": true,
    "misfirePolicy": "fireAndProceed",
    "invokeService": "script",
    "invokeContext": {
        "script": {
            "type": "text/javascript",
            "file": "script/testlog.js"
        }
    }
}' \
"http://localhost:8080/openidm/scheduler/job/testlog-schedule"
{
  "_id": "testlog-schedule",
  "enabled": true,
  "persisted": true,
  "recoverable": false,
  "misfirePolicy": "fireAndProceed",
  "schedule": "0/1 * * * * ?",
  "repeatInterval": 0,
  "repeatCount": 0,
  "type": "cron",
  "invokeService": "org.forgerock.openidm.script",
  "invokeContext": {
    "script": {
      "type": "text/javascript",
      "file": "script/testlog.js"
    }
  },
  "invokeLogLevel": "info",
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [
    {
      "calendar": null,
      "group": "scheduler-service-group",
      "jobKey": "scheduler-service-group.testlog-schedule",
      "name": "trigger-testlog-schedule",
      "nodeId": "node1",
      "previousState": null,
      "serialized": {
        "type": "CronTriggerImpl",
        "calendarName": null,
        "cronEx": {
          "cronExpression": "0/1 * * * * ?",
          "timeZone": "Africa/Johannesburg"
        },
        "description": null,
        "endTime": null,
        "fireInstanceId": "node1_1570611359345",
        "group": "scheduler-service-group",
        "jobDataMap": {
          "scheduler.invokeService": "org.forgerock.openidm.script",
          "scheduler.config-name": "scheduler-testlog-schedule",
          "scheduler.invokeContext": {
            "script": {
              "type": "text/javascript",
              "file": "script/testlog.js"
            }
```

```
          },
          "schedule.config": {
            "enabled": true,
            "persisted": true,
            "recoverable": false,
            "misfirePolicy": "fireAndProceed",
            "schedule": "0/1 * * * * ?",
            "repeatInterval": 0,
            "repeatCount": 0,
            "type": "cron",
            "invokeService": "org.forgerock.openidm.script",
            "invokeContext": {
              "script": {
                "type": "text/javascript",
                "file": "script/testlog.js"
              }
            },
            "invokeLogLevel": "info",
            "startTime": null,
            "endTime": null,
            "concurrentExecution": false
          },
          "scheduler.invokeLogLevel": "info"
        },
        "jobGroup": "scheduler-service-group",
        "jobName": "testlog-schedule",
        "misfireInstruction": 1,
        "name": "trigger-testlog-schedule",
        "nextFireTime": 1570611569000,
        "previousFireTime": 1570611568000,
        "priority": 5,
        "startTime": 1570611391000,
        "volatility": false
      },
      "state": "NORMAL",
      "_rev": "000000001d4724d6",
      "_id": "scheduler-service-group.trigger-testlog-schedule"
    }
  ],
  "previousRunDate": "2019-10-09T08:59:28.000Z",
  "nextRunDate": "2019-10-09T08:59:29.000Z"
}
```


Note that the output includes the `trigger` that was created as part of the scheduled job, as well as the `nextRunDate` for the job. For more information about the `trigger` properties, see Query Schedule Triggers.

The following example uses a POST request to create an identical schedule to the one created in the previous example, but with a server-assigned ID:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=2.0" \
--request POST \
--data '{
    "enabled": true,
```

```
    "type": "cron",
    "schedule": "0/1 * * * * ?",
    "persisted": true,
    "misfirePolicy": "fireAndProceed",
    "invokeService": "script",
    "invokeContext": {
        "script": {
            "type": "text/javascript",
            "file": "script/testlog.js"
        }
    }
}' \
"http://localhost:8080/openidm/scheduler/job?_action=create"
{
  "_id": "b12e4a77-a626-4a38-a1dc-8edc7498ca1c",
  "enabled": true,
  "persisted": true,
  "recoverable": false,
  "misfirePolicy": "fireAndProceed",
  "schedule": "0/1 * * * * ?",
  "repeatInterval": 0,
  "repeatCount": 0,
  "type": "cron",
  "invokeService": "org.forgerock.openidm.script",
  "invokeContext": {
    "script": {
      "type": "text/javascript",
      "file": "script/testlog.js"
    }
  },
  "invokeLogLevel": "info",
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [
    {
      "calendar": null,
      "group": "scheduler-service-group",
      "jobKey": "scheduler-service-group.b12e4a77-a626-4a38-a1dc-8edc7498ca1c",
      "name": "trigger-b12e4a77-a626-4a38-a1dc-8edc7498ca1c",
      "nodeId": null,
      "previousState": null,
      "serialized": {
        "type": "CronTriggerImpl",
        "calendarName": null,
        "cronEx": {
          "cronExpression": "0/1 * * * * ?",
          "timeZone": "Africa/Johannesburg"
        },
        "description": null,
        "endTime": null,
        "fireInstanceId": null,
        "group": "scheduler-service-group",
        "jobDataMap": {
          "scheduler.invokeService": "org.forgerock.openidm.script",
          "scheduler.config-name": "scheduler-b12e4a77-a626-4a38-a1dc-8edc7498ca1c",
          "scheduler.invokeContext": {
            "script": {
              "type": "text/javascript",
```

```
            "file": "script/testlog.js"
          }
        },
        "schedule.config": {
          "enabled": true,
          "persisted": true,
          "recoverable": false,
          "misfirePolicy": "fireAndProceed",
          "schedule": "0/1 * * * * ?",
          "repeatInterval": 0,
          "repeatCount": 0,
          "type": "cron",
          "invokeService": "org.forgerock.openidm.script",
          "invokeContext": {
            "script": {
              "type": "text/javascript",
              "file": "script/testlog.js"
            }
          },
          "invokeLogLevel": "info",
          "startTime": null,
          "endTime": null,
          "concurrentExecution": false
        },
        "scheduler.invokeLogLevel": "info"
      },
      "jobGroup": "scheduler-service-group",
      "jobName": "b12e4a77-a626-4a38-a1dc-8edc7498ca1c",
      "misfireInstruction": 1,
      "name": "trigger-b12e4a77-a626-4a38-a1dc-8edc7498ca1c",
      "nextFireTime": 1570611659000,
      "previousFireTime": null,
      "priority": 5,
      "startTime": 1570611659000,
      "volatility": false
    },
    "state": "NORMAL",
    "_rev": "000000009e2e2212",
    "_id": "scheduler-service-group.trigger-b12e4a77-a626-4a38-a1dc-8edc7498ca1c"
  }
 ],
 "previousRunDate": null,
 "nextRunDate": "2019-10-09T09:00:59.000Z"
}
```


The output includes the generated `_id` of the schedule, in this case `"_id": "b12e4a77-a626-4a38-a1dc-8edc7498ca1c"`.

+ *View Scheduled Job Details*

The following example displays the details of the schedule created in the previous example. Specify the job ID in the URL:

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

```
--header "Accept-API-Version: resource=2.0" \
--request GET \
"http://localhost:8080/openidm/scheduler/job/testlog-schedule"
{
  "_id": "testlog-schedule",
  "enabled": true,
  "persisted": true,
  "recoverable": false,
  "misfirePolicy": "fireAndProceed",
  "schedule": "0/1 * * * * ?",
  "repeatInterval": 0,
  "repeatCount": 0,
  "type": "cron",
  "invokeService": "org.forgerock.openidm.script",
  "invokeContext": {
    "script": {
      "type": "text/javascript",
      "file": "script/testlog.js"
    }
  },
  "invokeLogLevel": "info",
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [
    {
      "calendar": null,
      "group": "scheduler-service-group",
      "jobKey": "scheduler-service-group.testlog-schedule",
      "name": "trigger-testlog-schedule",
      "nodeId": null,
      "previousState": null,
      "serialized": {
        "type": "CronTriggerImpl",
        "calendarName": null,
        "cronEx": {
          "cronExpression": "0/1 * * * * ?",
          "timeZone": "Africa/Johannesburg"
        },
        "description": null,
        "endTime": null,
        "fireInstanceId": "node1_1570611359712",
        "group": "scheduler-service-group",
        "jobDataMap": {
          "scheduler.invokeService": "org.forgerock.openidm.script",
          "scheduler.config-name": "scheduler-testlog-schedule",
          "scheduler.invokeContext": {
            "script": {
              "type": "text/javascript",
              "file": "script/testlog.js"
            }
          },
          "schedule.config": {
            "enabled": true,
            "persisted": true,
            "recoverable": false,
            "misfirePolicy": "fireAndProceed",
            "schedule": "0/1 * * * * ?",
            "repeatInterval": 0,
```

```
            "repeatCount": 0,
            "type": "cron",
            "invokeService": "org.forgerock.openidm.script",
            "invokeContext": {
              "script": {
                "type": "text/javascript",
                "file": "script/testlog.js"
              }
            },
            "invokeLogLevel": "info",
            "startTime": null,
            "endTime": null,
            "concurrentExecution": false
          },
          "scheduler.invokeLogLevel": "info"
        },
        "jobGroup": "scheduler-service-group",
        "jobName": "testlog-schedule",
        "misfireInstruction": 1,
        "name": "trigger-testlog-schedule",
        "nextFireTime": 1570611719000,
        "previousFireTime": 1570611718000,
        "priority": 5,
        "startTime": 1570611391000,
        "volatility": false
      },
      "state": "NORMAL",
      "_rev": "000000002d1c2465",
      "_id": "scheduler-service-group.trigger-testlog-schedule"
    }
  ],
  "previousRunDate": "2019-10-09T09:01:58.000Z",
  "nextRunDate": "2019-10-09T09:01:59.000Z"
}
```


+ *Query Scheduled Jobs*

You can query defined and running scheduled jobs using a regular query filter.

The following query returns the IDs of all defined schedules:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request GET \
"http://localhost:8080/openidm/scheduler/job?_queryFilter=true&_fields=_id"
{
  "result": [
    {
      "_id": "reconcile_systemLdapAccounts_managedUser"
    },
    {
      "_id": "testlog-schedule"
    }
  ]
  ...
}
```

The following query returns the IDs, enabled status, and next run date of all defined schedules:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request GET \
"http://localhost:8080/openidm/scheduler/job?_queryFilter=true&_fields=_id,enabled,nextRunDate"
{
  "result": [
    {
      "_id": "reconcile_systemLdapAccounts_managedUser",
      "enabled": false,
      "nextRunDate": null
    },
    {
      "_id": "testlog-schedule",
      "enabled": true,
      "nextRunDate": "2019-10-09T09:43:17.000Z"
    }
  ]
  ...
}
```

+ *Update a Schedule*

To update a schedule definition, use a PUT request and update all the static properties of the object.

This example disables the `testlog` schedule created in the previous example by setting `"enabled":false`:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Content-Type: application/json" \
--header "Accept-API-Version: resource=2.0" \
```

```
--request PUT \
--data '{
    "enabled": false,
    "type": "cron",
    "schedule": "0/1 * * * * ?",
    "persisted": true,
    "misfirePolicy": "fireAndProceed",
    "invokeService": "script",
    "invokeContext": {
        "script": {
            "type": "text/javascript",
            "file": "script/testlog.js"
        }
    }
}' \
"http://localhost:8080/openidm/scheduler/job/testlog-schedule"
{
  "_id": "testlog-schedule",
  "enabled": false,
  "persisted": true,
  "recoverable": false,
  "misfirePolicy": "fireAndProceed",
  "schedule": "0/1 * * * * ?",
  "repeatInterval": 0,
  "repeatCount": 0,
  "type": "cron",
  "invokeService": "org.forgerock.openidm.script",
  "invokeContext": {
    "script": {
      "type": "text/javascript",
      "file": "script/testlog.js"
    }
  },
  "invokeLogLevel": "info",
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [],
  "previousRunDate": null,
  "nextRunDate": null
}
```

When you disable a schedule, all triggers are removed and the `nextRunDate` is set to `null`. If you re-enable the schedule, a new trigger is generated, and the `nextRunDate` is recalculated.

+ *List Running Scheduled Jobs*

This example returns a list of the jobs that are currently executing. The list lets you decide whether to wait for a specific job to complete before you place a server in maintenance mode.

This action does not list the jobs across a cluster, only the jobs currently executing on the node to which the request is routed.

Note that this list is accurate only at the moment the request was issued. The list can change at any time after the response is received.

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request POST \
"http://localhost:8080/openidm/scheduler/job?_action=listCurrentlyExecutingJobs"
[
  {
    "enabled": true,
    "persisted": true,
    "misfirePolicy": "fireAndProceed",
    "type": "simple",
    "repeatInterval": 3600000,
    "repeatCount": -1,
    "invokeService": "org.forgerock.openidm.sync",
    "invokeContext": {
      "action": "reconcile",
      "mapping": "systemLdapAccounts_managedUser"
    },
    "invokeLogLevel": "info",
    "timeZone": null,
    "startTime": null,
    "endTime": null,
    "concurrentExecution": false
  }
]
```

+ *Trigger a Schedule Manually*

For testing purposes, and for certain administrative tasks, you can trigger a scheduled task manually, outside of its specified schedule. A scheduled task must be `enabled` before it can be triggered.

This command triggers the `testlog-schedule` job created previously:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request POST \
"http://localhost:8080/openidm/scheduler/job/testlog-schedule?_action=trigger"
{
  "success": true
}
```

Note

This action is available only from version 2.0 of the scheduler API onwards.

+ *Pause and Resume a Scheduled Job*

Instead of deleting and recreating scheduled jobs, you can pause and resume them if necessary. This command pauses the `testlog-schedule` job:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request POST \
"http://localhost:8080/openidm/scheduler/job/testlog-schedule?_action=pause"
{
  "success": true
}
```

This command resumes the `testlog-schedule` job:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request POST \
"http://localhost:8080/openidm/scheduler/job/testlog-schedule?_action=resume"
{
  "success": true
}
```

Note

These actions are available only from version 2.0 of the scheduler API onwards.

+ *Pause All Scheduled Jobs*

In preparation for placing a server in maintenance mode, you can temporarily suspend all scheduled jobs. This action does not cancel or interrupt jobs that are already in progress; it simply prevents any scheduled jobs from being invoked during the suspension period.

This command suspends all scheduled tasks and returns `true` if the tasks could be suspended successfully:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request POST \
"http://localhost:8080/openidm/scheduler/job?_action=pauseJobs"
{
  "success": true
}
```

+ *Resume All Scheduled Jobs*

When an update has been completed, and your instance is no longer in maintenance mode, you can resume scheduled jobs to start them up again. Any jobs that were missed during the downtime will follow their configured misfire policy to determine whether they should be reinvoked.

This command resumes all scheduled jobs and returns `true` if the jobs could be resumed successfully:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request POST \
"http://localhost:8080/openidm/scheduler/job?_action=resumeJobs"
{
  "success": true
}
```

+ *Query Schedule Triggers*

When a scheduled job is created, a trigger for that job is created automatically and is included in the schedule definition. The trigger is essentially what causes the job to be started. You can read all the triggers that have been generated on a system with the following query on the `openidm/scheduler/trigger` context path:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request GET \
"http://localhost:8080/openidm/scheduler/trigger?_queryFilter=true"
{
  "result": [
    {
      "_id": "scheduler-service-group.trigger-testlog-schedule",
      "_rev": "00000000db3523f1",
      "calendar": null,
      "group": "scheduler-service-group",
      "jobKey": "scheduler-service-group.testlog-schedule",
      "name": "trigger-testlog-schedule",
      "nodeId": "node1",
      "previousState": null,
      "serialized": {
      ...
      },
      "state": "NORMAL"
    }
  ]
}
```

The contents of a trigger object are as follows:

`_id`

The ID of the trigger, which is based on the schedule ID. The trigger ID is made up of the *group name*, followed by `trigger-` prepended to the *schedule ID*: `group.trigger-schedule-id`. For example, if the schedule ID was `testlog-schedule`, then the trigger ID would be `scheduler-service-group.trigger-testlog-schedule`.

`_rev`

The revision of the trigger object. This property is reserved for internal use and specifies the revision of the object in the repository. This is the same value that is exposed as the object's ETag through the REST API. The content of this property is not defined. No consumer should make any assumptions of its content beyond equivalence comparison.

`previousState`

The previous state of the trigger, before its current state. For a description of Quartz trigger states, see the Quartz API documentation.

`name`

The trigger name, which matches the ID of the schedule that created the trigger, with `trigger-` added: `trigger-schedule-id`.

`state`

The current state of the trigger. For a description of Quartz trigger states, see the Quartz API documentation.

`nodeId`

The ID of the node that has acquired the trigger, useful in a clustered deployment. If the trigger has not been acquired by a node yet, this will return `null`.

`calendar`

This is a part of the Quartz implementation, but is not currently supported by IDM. This will always return `null`.

`serialized`

The JSON serialization of the trigger class.

`group`

The name of the group that the trigger is in, always `scheduler-service-group`.

`jobKey`

The name of the job associated with the trigger: `group.schedule-id`.

To read the contents of a specific trigger send a GET request to the trigger ID, for example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request GET \
"http://localhost:8080/openidm/scheduler/trigger/scheduler-service-group.trigger-testlog-schedule"
{
  "_id": "scheduler-service-group.trigger-testlog-schedule",
  "_rev": "00000000cd1723dd",
  "calendar": null,
  "group": "scheduler-service-group",
  "jobKey": "scheduler-service-group.testlog-schedule",
  "name": "trigger-testlog-schedule",
  "nodeId": "node1",
  "previousState": null,
  "serialized": {
  ...
  },
  "state": "NORMAL"
}
```

To view the triggers that have been acquired, send a GET request to the scheduler, with a `_queryFilter` of `nodeId`. For example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request GET \
"http://localhost:8080/openidm/scheduler/trigger?_queryFilter=(nodeId+pr)"
```

To view the triggers that have not yet been acquired by any node, send a GET request to the scheduler, with a `_queryFilter` to list the triggers with a null `nodeId`. For example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request GET \
"http://localhost:8080/openidm/scheduler/trigger?_queryFilter=%21(nodeId+pr)"
```

+ *Delete a Schedule*

To delete a schedule, send a DELETE request to the schedule ID. For example:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=2.0" \
--request DELETE \
"http://localhost:8080/openidm/scheduler/job/testlog-schedule"
{
  "_id": "testlog-schedule",
  "enabled": false,
  "persisted": true,
  "recoverable": false,
  "misfirePolicy": "fireAndProceed",
  "schedule": "0/1 * * * * ?",
  "repeatInterval": 0,
  "repeatCount": 0,
  "type": "cron",
  "invokeService": "org.forgerock.openidm.script",
  "invokeContext": {
    "script": {
      "type": "text/javascript",
      "file": "script/testlog.js"
    }
  },
  "invokeLogLevel": "info",
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [],
  "previousRunDate": null,
  "nextRunDate": null
}
```

The DELETE request returns the entire JSON object.

Manage Schedules Using the Admin UI

To manage schedules in the Admin UI, select Configure > Schedules.

Add, remove and change schedules here. By default, only persisted schedules are shown in the Schedules list. To show non-persisted (in memory) schedules, select Filter by Type > In Memory.

Chapter 2

Scan Data to Trigger Tasks

In addition to the fine-grained scheduling facility, IDM provides a task scanning mechanism. The task scanner lets you scan a set of properties with a complex query filter, at a scheduled interval, and then launches a script on the objects returned by the query.

For example, the task scanner can scan all `managed/user` objects for a "sunset date" and can invoke a script that launches a "sunset task" on the user object when this date is reached.

- "Configure the Task Scanner"
- "Manage Scanning Tasks Over REST"
- "Manage Scanning Tasks Using the UI"

Configure the Task Scanner

The task scanner is essentially a scheduled task that queries a set of managed users, then launches a script based on the query results. The task scanner is configured in the same way as a regular scheduled task in a schedule configuration file named (`schedule-`*`task-name`*`.json)`, with the `invokeService` parameter set to `taskscanner`. The `invokeContext` parameter defines the details of the scan, and the task that should be launched when the specified condition is triggered.

The following example defines a scheduled scanning task that triggers a sunset script. This sample schedule configuration file is provided in `openidm/samples/example-configurations/task-scanner/conf/schedule-taskscan_sunset.json`. To use the sample file, copy it to your project's `conf` directory and edit it as required.

```
{
  "enabled" : true,
  "type" : "simple",
  "repeatInterval" : 3600000,
  "persisted": true,
  "concurrentExecution" : false,
  "invokeService" : "taskscanner",
  "invokeContext" : {
    "waitForCompletion" : false,
    "numberOfThreads" : 5,
    "scan" : {
      "_queryFilter" : "((/sunset/date lt \"${Time.now}\") AND !(/sunset/task-completed pr))",
      "object" : "managed/user",
      "taskState" : {
        "started" : "/sunset/task-started",
        "completed" : "/sunset/task-completed"
      },
      "recovery" : {
        "timeout" : "10m"
      }
    },
    "task" : {
      "script" : {
        "type" : "text/javascript",
        "file" : "script/sunset.js"
      }
    }
  }
}
```

The schedule configuration calls a script (`script/sunset.js`). To test the sample, copy `openidm/samples/example-configurations/task-scanner/script/sunset.js` to your project's `script` directory. You will also need to assign a user a sunset date. The task will only execute on users who have a valid sunset date field. The sunset date field can be added manually to users, but will need to be added to the `managed/user` schema if you want the field to be visible from the Admin UI.

The following example command adds a sunset date field to the user `bjensen` using the REST interface:

```
curl \
--header "Content-Type: application/json" \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--request POST \
--data '[{
 "operation" : "add",
 "field" : "sunset/date",
 "value" : "2019-12-20T12:00:00Z"
}]' \
"http://localhost:8080/openidm/managed/user?_action=patch&_queryFilter=userName+eq+'bjensen'"
```

The remaining properties in the schedule configuration are as follows:

The `invokeContext` parameter takes the following properties:

`waitForCompletion` (optional)

This property specifies whether the task should be performed synchronously. Tasks are performed asynchronously by default (with `waitForCompletion` set to false). A task ID (such as `{"_id":"354ec41f-c781-4b61-85ac-93c28c180e46"}`) is returned immediately. If this property is set to true, tasks are performed synchronously and the ID is not returned until all tasks have completed.

`maxRecords` (optional)

The maximum number of records that can be processed. This property is not set by default so the number of records is unlimited. If a maximum number of records is specified, that number will be spread evenly over the number of threads.

`numberOfThreads` (optional)

By default, the task scanner runs in a multi-threaded manner, that is, numerous threads are dedicated to the same scanning task run. Multi-threading generally improves the performance of the task scanner. The default number of threads for a single scanning task is 10. To change this default, set the `numberOfThreads` property. The sample configuration sets the default number of threads to 5.

`scan`

The details of the scan. The following properties are defined:

`_queryFilter`

The query filter that identifies the entries for which this task should be run.

The query filter provided in the sample schedule configuration (`(/sunset/date lt \"${Time.now}\") AND !(/sunset/task-completed pr)`) identifies managed users whose `sunset/date` property is before the current date and for whom the sunset task has not yet completed.

The sample query supports time-based conditions, with the time specified in ISO 8601 format (Zulu time). You can write any query to target the set of entries that you want to scan.

For time-based queries, it's possible to use the `${Time.now}` macro object (which fetches the current time). You can also specify any date/time in relation to the current time, using the `+` or `-` operator, and a duration modifier. For example: changing the sample query to `${Time.now + 1d}` would return all user objects whose `/sunset/date` is the following day (current time plus one day). Note: you must include space characters around the operator (`+` or `-`). The duration modifier supports the following unit specifiers:

`s` - second
`m` - minute
`h` - hour
`d` - day
`M` - month
`y` - year

`object`

Defines the managed object type against which the query should be performed, as defined in the `managed.json` file.

`taskState`

Indicates the names of the fields in which the start message and the completed message are stored. These fields are used to track the status of the task.

`started` specifies the field that stores the timestamp for when the task begins. `completed` specifies the field that stores the timestamp for when the task completes its operation. The `completed` field is present as soon as the task has started, but its value is `null` until the task has completed.

`recovery` **(optional)**

Specifies a configurable `timeout`, after which the task scanner process ends. For clustered IDM instances, there might be more than one task scanner running at a time. A task cannot be launched by two task scanners at the same time. When one task scanner "claims" a task, it indicates that the task has been started. That task is then unavailable to be claimed by another task scanner and remains unavailable until the end of the task is indicated. In the event that the first task scanner does not complete the task by the specified timeout, for whatever reason, a second task scanner can pick up the task.

`task`

Provides details of the task that is performed. Usually, the task is invoked by a script, whose details are defined in the `script` property:

- `type` – the type of script, either JavaScript or Groovy.
- `file` – the path to the script file. The script file takes at least two objects (in addition to the default objects that are provided to all IDM scripts):
  - `input` – the individual object that is retrieved from the query (in the example, this is the individual user object).
  - `objectID` – a string that contains the full identifier of the object. The `objectID` is useful for performing updates with the script as it allows you to target the object directly. For example: `openidm.update(objectID, input['_rev'], input);`.

  A sample script file is provided in `openidm/samples/example-configurations/task-scanner/script/sunset.js`. To use this sample file, copy it to your project's `script/` directory. The sample script marks all user objects that match the specified conditions as inactive. You can use this sample script to trigger a specific workflow, or any other task associated with the sunset process.

For more information about using scripts, see "*Scripting Function Reference*" in the *Scripting Guide*.

Manage Scanning Tasks Over REST

You can trigger, cancel, and monitor scanning tasks over the REST interface, using the REST endpoint `openidm/taskscanner`:

+ *Create a Scanning Task*

You can define a scanning task in a configuration file or directly over the REST interface. For an example of a file-based scanning task, see the file `/path/to/openidm/samples/example-configurations/task-scanner/conf/schedule-taskscan_sunset.json`.

The following command defines a scanning task named `sunsetTask` over REST:

```
curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-type: application/json" \
 --header "Accept-API-Version: resource=2.0" \
 --request PUT \
 --data '{
  "enabled" : true,
  "type" : "simple",
  "repeatInterval" : 3600000,
  "persisted": true,
  "concurrentExecution" : false,
  "invokeService" : "taskscanner",
  "invokeContext" : {
    "waitForCompletion" : false,
    "numberOfThreads" : 5,
    "scan" : {
      "_queryFilter" : "((/sunset/date lt \"${Time.now}\") AND !(${taskState.completed} pr))",
      "object" : "managed/user",
      "taskState" : {
        "started" : "/sunset/task-started",
        "completed" : "/sunset/task-completed"
      },
      "recovery" : {
        "timeout" : "10m"
      }
    },
    "task" : {
      "script" : {
        "type" : "text/javascript",
        "file" : "script/sunset.js"
      }
    }
  }
}' \
"http://localhost:8080/openidm/scheduler/job/sunsetTask"
{
  "_id": "sunsetTask",
  "enabled": true,
  "persisted": true,
  "recoverable": false,
  "misfirePolicy": "fireAndProceed",
  "schedule": null,
```

```
  "repeatInterval": 3600000,
  "repeatCount": -1,
  "type": "simple",
  "invokeService": "org.forgerock.openidm.taskscanner",
  "invokeContext": {
    "waitForCompletion": false,
    "numberOfThreads": 5,
    "scan": {
      "_queryFilter": "((/sunset/date lt \"${Time.now}\") AND !(${taskState.completed} pr))",
      "object": "managed/user",
      "taskState": {
        "started": "/sunset/task-started",
        "completed": "/sunset/task-completed"
      },
      "recovery": {
        "timeout": "10m"
      }
    },
    "task": {
      "script": {
        "type": "text/javascript",
        "file": "script/sunset.js"
      }
    }
  },
  "invokeLogLevel": "info",
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [
    {
      "calendar": null,
      "group": "scheduler-service-group",
      "jobKey": "scheduler-service-group.sunsetTask",
      "name": "trigger-sunsetTask",
      "nodeId": null,
      "previousState": null,
      "serialized": {
        "type": "SimpleTriggerImpl",
        "calendarName": null,
        "complete": false,
        "description": null,
        "endTime": null,
        "fireInstanceId": null,
        "group": "scheduler-service-group",
        "jobDataMap": {
          "scheduler.invokeService": "org.forgerock.openidm.taskscanner",
          "scheduler.config-name": "scheduler-sunsetTask",
          "scheduler.invokeContext": {
            "waitForCompletion": false,
            "numberOfThreads": 5,
            "scan": {
              "_queryFilter": "((/sunset/date lt \"${Time.now}\") AND !(${taskState.completed} pr))",
              "object": "managed/user",
              "taskState": {
                "started": "/sunset/task-started",
                "completed": "/sunset/task-completed"
              },
              "recovery": {
```

```
                "timeout": "10m"
              }
            },
            "task": {
              "script": {
                "type": "text/javascript",
                "file": "script/sunset.js"
              }
            }
          },
          "schedule.config": {
            "enabled": true,
            "persisted": true,
            "recoverable": false,
            "misfirePolicy": "fireAndProceed",
            "schedule": null,
            "repeatInterval": 3600000,
            "repeatCount": -1,
            "type": "simple",
            "invokeService": "org.forgerock.openidm.taskscanner",
            "invokeContext": {
              "waitForCompletion": false,
              "numberOfThreads": 5,
              "scan": {
                "_queryFilter": "((/sunset/date lt \"${Time.now}\") AND !(${taskState.completed}
pr))",
                "object": "managed/user",
                "taskState": {
                  "started": "/sunset/task-started",
                  "completed": "/sunset/task-completed"
                },
                "recovery": {
                  "timeout": "10m"
                }
              },
              "task": {
                "script": {
                  "type": "text/javascript",
                  "file": "script/sunset.js"
                }
              }
            },
            "invokeLogLevel": "info",
            "startTime": null,
            "endTime": null,
            "concurrentExecution": false
          },
          "scheduler.invokeLogLevel": "info"
        },
        "jobGroup": "scheduler-service-group",
        "jobName": "sunsetTask",
        "misfireInstruction": 1,
        "name": "trigger-sunsetTask",
        "nextFireTime": 1570618094818,
        "previousFireTime": null,
        "priority": 5,
        "repeatCount": -1,
        "repeatInterval": 3600000,
        "startTime": 1570618094818,
```

```
        "timesTriggered": 0,
        "volatility": false
      },
      "state": "NORMAL",
      "_rev": "000000006751ccf1",
      "_id": "scheduler-service-group.trigger-sunsetTask"
    }
  ],
  "previousRunDate": null,
  "nextRunDate": "2019-10-09T10:48:14.818Z"
}
```

+ *Trigger a Scanning Task*

To trigger a scanning task over REST, use the `execute` action and specify the `name` of the task (effectively the scheduled job name). To obtain a list of task names, you can query the `/openidm/scheduler/job` endpoint. Note, however, that not all jobs are scanning tasks. Only those jobs that have which have the correct task scanner `invokeContext` can be triggered in this way.

The following example triggers the `sunsetTask` defined in the previous example:

```
curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Accept-API-Version: resource=1.0" \
 --request POST \
 "http://localhost:8080/openidm/taskscanner?_action=execute&name=sunsetTask"
{
  "_id": "9f2564c8-193c-4871-8869-6080f374b1bd-2073"
}
```

For scanning tasks that are defined in configuration files, you can determine the task name from the file name, for example, `schedule-task-name.json`. The following example triggers a task named `taskscan_sunset` that is defined in a file named `conf/schedule-taskscan_sunset.json`:

```
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/taskscanner?_action=execute&name=taskscan_sunset"
{
  "_id": "8d7742f0-5245-41cf-89a5-de32fc50e326-3323"
}
```

By default, a scanning task ID is returned immediately when the task is initiated. Clients can make subsequent calls to the task scanner service, using this task ID to query its state and to call operations on it.

To have the scanning task complete before the ID is returned, set the `waitForCompletion` property to `true` in the task definition file (`schedule-taskscan_sunset.json`).

+ *Cancel a Scanning Task*

To cancel a scanning task that is in progress, send a REST call with the `cancel` action, specifying the task ID. The following call cancels the scanning task initiated in the previous example:

```
curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Accept-API-Version: resource=1.0" \
 --request POST \
 "http://localhost:8080/openidm/taskscanner/9f2564c8-193c-4871-8869-6080f374b1bd-2073?_action=cancel"
 {
     "_id":"9f2564c8-193c-4871-8869-6080f374b1bd-2073",
     "status":"SUCCESS"
 }
```

Note

You cannot cancel a scanning task that has already completed.

+ *List the Scanning Tasks*

To retrieve a list of scanning tasks, query the `openidm/taskscanner` context path. The following example displays *all* scanning tasks, regardless of their state:

```
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/taskscanner?_queryFilter=true"
 {
  "result": [
    {
      "_id": "9f2564c8-193c-4871-8869-6080f374b1bd-2073",
      "name": "schedule/taskscan_sunset",
      "progress": {
        "state": "COMPLETED",
        "processed": 0,
        "total": 0,
        "successes": 0,
        "failures": 0
      },
      "started": "2017-12-19T11:45:53.433Z",
      "ended": "2017-12-19T11:45:53.438Z"
    },
    {
      "_id": "b32aafe5-b484-4d00-89ff-83554341f321-9970",
      "name": "schedule/taskscan_sunset",
      "progress": {
        "state": "ACTIVE",
        "processed": 80,
        "total": 980,
        "successes": 80,
```

```
        "failures": 0
      },
      "started": "2017-12-19T16:41:04.185Z",
      "ended": null
    }
  ]
  ...
}
```

Each scanning task has the following properties:

`_id`

The unique ID of that task instance.

`name`

The name of the scanning task, determined by the name of the schedule configuration file or over REST when the task is executed.

`started`

The time at which the scanning task started.

`ended`

The time at which the scanning task ended.

`progress`

The progress of the scanning task, summarised in the following fields:

`failures` - the number of records not able to be processed
`successes` - the number of records processed successfully
`total` - the total number of records
`processed` - the number of processed records
`state` - the current state of the task, `INITIALIZED`, `ACTIVE`, `COMPLETED`, `CANCELLED`, or `ERROR`

The number of processed tasks whose details are retained is governed by the `openidm.taskscanner.maxcompletedruns` property in the `conf/system.properties` file. By default, the last one hundred completed tasks are retained.

Manage Scanning Tasks Using the UI

The task scanner queries a set of managed objects, then executes a script on the objects returned in the query result. The scanner then sets a field on a specific managed object property to indicate the state of the task. Before you start, you must set up this object type property on the managed user object.

In the example that follows, the task scanner queries managed user objects and returns objects whose `sunset` property holds a date that is prior to the current date. The scanner then sets the state of the task in the `task-completed` field of the user's `sunset` property:

1. Select Configure > Schedules and click Add Schedule.
2. Enable the schedule, and set the times that the task should run, as for any other schedule.
3. Under Perform Action, select "Execute a script on objects returned by a query (Task Scanner)".
4. Select the managed object on which the query should be run, in this case, `user`.
5. Build the query that will be run against the managed user objects.

   The following query (based on the example schedule available in `/path/to/openidm/samples/example-configurations/task-scanner`) returns all managed users whose `sunset` date is prior to the current date (`${Time.now}`) and for whom the `sunset` task has not already completed (`${taskState.completed} pr`):

   ```
   ((/sunset/date lt \"${Time.now}\") AND !(${taskState.completed} pr))
   ```

6. In the Object Property Field, enter the property whose values will determine the state of the task, in this case `sunset`.
7. In the Script field, enter an inline script, or a path to the file containing the script that should be launched on the results of the query.

   The sample task scanner runs the following script on the managed users returned by the previous query:

   ```
   var patch = [{ "operation" : "replace", "field" : "/active", "value" : false },{ "operation"
    : "replace", "field" : "/accountStatus", "value" : "inactive" }];
   openidm.patch(objectID, null, patch);
   ```

   This script essentially deactivates the accounts of users returned by the query by setting the value of their `active` property to `false`.

8. (Optional) Configure the advanced properties of the schedule described in "Configure Schedules".

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 more information, see What is OSGi? 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.