Integrator's Guide

/ OpenIDM 5

Latest update: 5.0.1.1

Anders Askåsen
Paul Bryan
Mark Craig
Andi Egloff
Laszlo Hordos
Matthias Tristl
Lana Frost
Mike Jang
Daly Chikhaoui
Nabil Maynard

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

Abstract

Guide to configuring and integrating OpenIDM software into identity management solutions. This software offers flexible services for automating management of the identity life cycle.

Table of Contents

Preface

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.

1. About This Guide

In this guide you will learn how to integrate OpenIDM software as part of a complete identity management solution.

This guide is written for systems integrators building solutions based on OpenIDM services. This guide describes the product functionality, and shows you how to set up and configure OpenIDM software as part of your overall identity management solution.

2. Formatting Conventions

Most examples in the documentation are created in GNU/Linux or Mac OS X operating environments. If distinctions are necessary between operating environments, examples are labeled with the operating environment name in parentheses. To avoid repetition file system directory names are often given only in UNIX format as in `/path/to/server`, even if the text applies to `C:\path\to\server` as well.

Absolute path names usually begin with the placeholder `/path/to/`. This path might translate to `/opt/`, `C:\Program Files\`, or somewhere else on your system.

Command-line, terminal sessions are formatted as follows:

```
$ echo $JAVA_HOME
/path/to/jdk
```

Command output is sometimes formatted for narrower, more readable output even though formatting parameters are not shown in the command.

Program listings are formatted as follows:

```
class Test {
    public static void main(String [] args)  {
        System.out.println("This is a program listing.");
    }
}
```

3. Accessing Documentation Online

ForgeRock publishes comprehensive documentation online:

- The ForgeRock Knowledge Base offers a large and increasing number of up-to-date, practical articles that help you deploy and manage ForgeRock software.

  While many articles are visible to community members, ForgeRock customers have access to much more, including advanced information for customers using ForgeRock software in a mission-critical capacity.

- ForgeRock product documentation, such as this document, aims to be technically accurate and complete with respect to the software documented. It is visible to everyone and covers all product features and examples of how to use them.

4. Using the ForgeRock.org Site

The ForgeRock.org site has links to source code for ForgeRock open source software, as well as links to the ForgeRock forums and technical blogs.

If you are a *ForgeRock customer*, raise a support ticket instead of using the forums. ForgeRock support professionals will get in touch to help you.

Chapter 1
Architectural Overview

This chapter introduces the OpenIDM architecture, and describes component modules and services.

In this chapter you will learn:

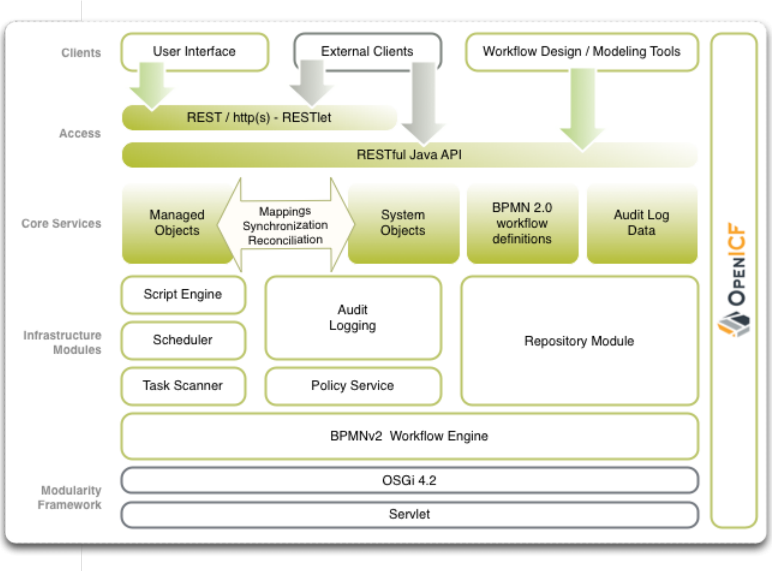
- How OpenIDM uses the OSGi framework as a basis for its modular architecture
- How the infrastructure modules provide the features required for OpenIDM's core services
- What those core services are and how they fit in to the overall architecture
- How OpenIDM provides access to the resources it manages

1.1. Modular Framework

OpenIDM implements infrastructure modules that run in an OSGi framework. It exposes core services through RESTful APIs to client applications.

The following figure provides an overview of the OpenIDM architecture, which is covered in more detail in subsequent sections of this chapter.

Modular Architecture



The OpenIDM framework is based on OSGi:

OSGi

OSGi is a module system and service platform for the Java programming language that implements a complete and dynamic component model. For a good introduction to OSGi, see the OSGi site. OpenIDM currently runs in Apache Felix, an implementation of *the OSGi Framework and Service Platform*.

Servlet

The Servlet layer provides RESTful HTTP access to the managed objects and services. OpenIDM embeds the Jetty Servlet Container, which can be configured for either HTTP or HTTPS access.

1.2. Infrastructure Modules

Infrastructure modules provide the underlying features needed for core services:

BPMN 2.0 Workflow Engine

OpenIDM provides an embedded workflow and business process engine based on Activiti and the Business Process Model and Notation (BPMN) 2.0 standard.

For more information, see "*Integrating Business Processes and Workflows*".

Task Scanner

OpenIDM provides a task-scanning mechanism that performs a batch scan for a specified property in OpenIDM data, on a scheduled interval. The task scanner then executes a task when the value of that property matches a specified value.

For more information, see "Scanning Data to Trigger Tasks".

Scheduler

The scheduler provides a **cron**-like scheduling component implemented using the Quartz library. Use the scheduler, for example, to enable regular synchronizations and reconciliations.

For more information, see "*Scheduling Tasks and Events*".

Script Engine

The script engine is a pluggable module that provides the triggers and plugin points for OpenIDM. OpenIDM currently supports JavaScript and Groovy.

Policy Service

OOpenIDM provides an extensible policy service that applies validation requirements to objects and properties, when they are created or updated.

For more information, see "*Using Policies to Validate Data*".

Audit Logging

Auditing logs all relevant system activity to the configured log stores. This includes the data from reconciliation as a basis for reporting, as well as detailed activity logs to capture operations on the internal (managed) and external (system) objects.

For more information, see "*Logging Audit Information*".

Repository

The repository provides a common abstraction for a pluggable persistence layer. OpenIDM supports reconciliation and synchronization with several major external repositories in production, including relational databases, LDAP servers, and even flat CSV and XML files.

The repository API uses a JSON-based object model with RESTful principles consistent with the other OpenIDM services. To facilitate testing, OpenIDM includes an embedded instance of OrientDB, a NoSQL database. You can then incorporate a supported internal repository, as described in "*Installing a Repository For Production*" in the *Installation Guide*.

1.3. Core Services

The core services are the heart of the OpenIDM resource-oriented unified object model and architecture:

Object Model

Artifacts handled by OpenIDM are Java object representations of the JavaScript object model as defined by JSON. The object model supports interoperability and potential integration with many applications, services, and programming languages.

OpenIDM can serialize and deserialize these structures to and from JSON as required. OpenIDM also exposes a set of triggers and functions that system administrators can define, in either JavaScript or Groovy, which can natively read and modify these JSON-based object model structures.

Managed Objects

A *managed object* is an object that represents the identity-related data managed by OpenIDM. Managed objects are configurable, JSON-based data structures that OpenIDM stores in its pluggable repository. The default managed object configuration includes users and roles, but you can define any kind of managed object, for example, groups or devices.

You can access managed objects over the REST interface with a query similar to the following:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/..."
```

System Objects

System objects are pluggable representations of objects on external systems. For example, a user entry that is stored in an external LDAP directory is represented as a system object in OpenIDM.

System objects follow the same RESTful resource-based design principles as managed objects. They can be accessed over the REST interface with a query similar to the following:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/system/..."
```

There is a default implementation for the OpenICF framework, that allows any connector object to be represented as a system object.

Mappings

Mappings define policies between source and target objects and their attributes during synchronization and reconciliation. Mappings can also define triggers for validation, customization, filtering, and transformation of source and target objects.

For more information, see "*Synchronizing Data Between Resources*".

Synchronization and Reconciliation

Reconciliation enables on-demand and scheduled resource comparisons between the OpenIDM managed object repository and source or target systems. Comparisons can result in different actions, depending on the mappings defined between the systems.

Synchronization enables creating, updating, and deleting resources from a source to a target system, either on demand or according to a schedule.

For more information, see "*Synchronizing Data Between Resources*".

1.4. Secure Commons REST Commands

Representational State Transfer (REST) is a software architecture style for exposing resources, using the technologies and protocols of the World Wide Web. For more information on the ForgeRock REST API, see "*REST API Reference*".

REST interfaces are commonly tested with a **curl** command. Many of these commands are used in this document. They work with the standard ports associated with Java EE communications, 8080 and 8443.

To run **curl** over the secure port, 8443, you must include either the **--insecure** option, or follow the instructions shown in "Restrict REST Access to the HTTPS Port". You can use those instructions with the self-signed certificate generated when OpenIDM starts, or with a `*.crt` file provided by a certificate authority.

In many examples in this guide, **curl** commands to the secure port are shown with a `--cacert self-signed.crt` option. Instructions for creating that `self-signed.crt` file are shown in "Restrict REST Access to the HTTPS Port".

1.5. Access Layer

The access layer provides the user interfaces and public APIs for accessing and managing the OpenIDM repository and its functions:

RESTful Interfaces

OpenIDM provides REST APIs for CRUD operations, for invoking synchronization and reconciliation, and to access several other services.

For more information, see "*REST API Reference*".

User Interfaces

User interfaces provide access to most of the functionality available over the REST API.

Chapter 2
Starting and Stopping the Server

This chapter covers the scripts provided for starting and stopping OpenIDM, and describes how to verify the *health* of a system, that is, that all requirements are met for a successful system startup.

2.1. To Start and Stop the Server

By default you start and stop OpenIDM in interactive mode.

To start OpenIDM interactively, open a terminal or command window, change to the `openidm` directory, and run the startup script:

- **startup.sh** (UNIX)

- **startup.bat** (Windows)

The startup script starts the server, and opens an OSGi console with a `->` prompt where you can issue console commands.

The default hostname and ports for OpenIDM are set in the `conf/boot/boot.properties` file found in the `openidm/` directory. OpenIDM is initially configured to run on `http` on port `8080`, `https` on port `8443`, with a hostname of `localhost`. For more information about changing ports and hostnames, see "*Host and Port Information*".

To stop the server interactively in the OSGi console, run the **shutdown** command:

```
-> shutdown
```

You can also start OpenIDM as a background process on UNIX and Linux. Follow these steps *before starting OpenIDM for the first time*.

1. If you have already started the server, shut it down and remove the Felix cache files under `openidm/felix-cache/`:

   ```
   -> shutdown
   ...
   $ rm -rf felix-cache/*
   ```

2. Start the server in the background. The **nohup** survives a logout and the **2>&1&** redirects standard output and standard error to the noted `console.out` file:

```
$ nohup ./startup.sh > logs/console.out 2>&1&
[1] 2343
```

To stop OpenIDM running as a background process, use the **shutdown.sh** script:

```
$ ./shutdown.sh
./shutdown.sh
Stopping OpenIDM (2343)
```

Incidentally, the process identifier (PID) shown during startup should match the PID shown during shutdown.

Note

Although installations on OS X systems are not supported in production, you might want to run OpenIDM on OS X in a demo or test environment. To run OpenIDM in the background on an OS X system, take the following additional steps:

- Remove the `org.apache.felix.shell.tui-*.jar` bundle from the `openidm/bundle` directory.
- Disable `ConsoleHandler` logging, as described in "Disabling Logs".

2.2. Specifying the Startup Configuration

By default, OpenIDM starts with the configuration, script, and binary files in the `openidm/conf`, `openidm/script`, and `openidm/bin` directories. You can launch OpenIDM with a different set of configuration, script, and binary files for test purposes, to manage different projects, or to run one of the included samples.

The **startup.sh** script enables you to specify the following elements of a running instance:

- `--project-location` or `-p /path/to/project/directory`

  The project location specifies the directory with OpenIDM configuration and script files.

  All configuration objects and any artifacts that are not in the bundled defaults (such as custom scripts) *must* be included in the project location. These objects include all files otherwise included in the `openidm/conf` and `openidm/script` directories.

  For example, the following command starts the server with the configuration of Sample 1, with a project location of `/path/to/openidm/samples/sample1`:

  ```
  $ ./startup.sh -p /path/to/openidm/samples/sample1
  ```

  If you do not provide an absolute path, the project location path is relative to the system property, `user.dir`. OpenIDM then sets `launcher.project.location` to that relative directory path. Alternatively, if you start OpenIDM without the **-p** option, OpenIDM sets `launcher.project.location` to `/path/to/openidm/conf`.

Note

In this documentation, "your project" refers to the value of `launcher.project.location`.

- `--working-location` or `-w /path/to/working/directory`

  The working location specifies the directory to which OpenIDM writes its database cache, audit logs, and felix cache. The working location includes everything that is in the default `db/` and `audit/`, and `felix-cache/` subdirectories.

  The following command specifies that OpenIDM writes its database cache and audit data to `/Users/admin/openidm/storage`:

  ```
  $ ./startup.sh -w /Users/admin/openidm/storage
  ```

  If you do not provide an absolute path, the path is relative to the system property, `user.dir`. If you do not specify a working location, OpenIDM writes this data to the `openidm/db`, `openidm/felix-cache` and `openidm/audit` directories.

  Note that this property does not affect the location of the OpenIDM system logs. To change the location of the OpenIDM logs, edit the `conf/logging.properties` file.

  You can also change the location of the Felix cache, by editing the `conf/config.properties` file, or by starting OpenIDM with the `-s` option, described later in this section.

- `--config` or `-c /path/to/config/file`

  A customizable startup configuration file (named `launcher.json`) enables you to specify how the OSGi Framework is started.

  Unless you are working with a highly customized deployment, you should not modify the default framework configuration. This option is therefore described in more detail in "*Advanced Configuration*".

- `--storage` or `-s /path/to/storage/directory`

  Specifies the OSGi storage location of the cached configuration files.

  You can use this option to redirect output if you are installing OpenIDM on a read-only filesystem volume. For more information, see "*Installing on a Read-Only Volume*" in the *Installation Guide*. This option is also useful when you are testing different configurations. Sometimes when you start OpenIDM with two different sample configurations, one after the other, the cached configurations are merged and cause problems. Specifying a storage location creates a separate `felix-cache` directory in that location, and the cached configuration files remain completely separate.

By default, properties files are loaded in the following order, and property values are resolved in the reverse order:

1. `system.properties`
2. `config.properties`
3. `boot.properties`

If both system and boot properties define the same attribute, the property substitution process locates the attribute in `boot.properties` and does not attempt to locate the property in `system.properties`.

You can use variable substitution in any `.json` configuration file with the install, working and project locations described previously. You can substitute the following properties:

`install.location`
`install.url`
`working.location`
`working.url`
`project.location`
`project.url`

Property substitution takes the following syntax:

```
&{launcher.property}
```

For example, to specify the location of the OrientDB database, you can set the `dbUrl` property in `repo.orientdb.json` as follows:

```
"dbUrl" : "local:&{launcher.working.location}/db/openidm",
```

The database location is then relative to a working location defined in the startup configuration.

You can find more examples of property substitution in many other files in your project's `conf/` subdirectory.

Note that property substitution does not work for connector reference properties. So, for example, the following configuration would not be valid:

```
"connectorRef" : {
    "connectorName" : "&{connectorName}",
    "bundleName" : "org.forgerock.openicf.connectors.ldap-connector",
    "bundleVersion" : "&{LDAP.BundleVersion}"
    ...
```

The `"connectorName"` must be the precise string from the connector configuration. If you need to specify multiple connector version numbers, use a range of versions, for example:

```
"connectorRef" : {
    "connectorName" : "org.identityconnectors.ldap.LdapConnector",
    "bundleName" : "org.forgerock.openicf.connectors.ldap-connector",
    "bundleVersion" : "[1.4.0.0,2.0.0.0)",
    ...
```

2.3. Monitoring Basic Server Health

Due to the highly modular, configurable nature of OpenIDM, it is often difficult to assess whether a system has started up successfully, or whether the system is ready and stable after dynamic configuration changes have been made.

OpenIDM includes a health check service, with options to monitor the status of internal resources.

To monitor the status of external resources such as LDAP servers and external databases, use the commands described in "Checking the Status of External Systems Over REST".

2.3.1. Basic Health Checks

The health check service reports on the state of the OpenIDM system and outputs this state to the OSGi console and to the log files. The system can be in one of the following states:

- `STARTING` - OpenIDM is starting up
- `ACTIVE_READY` - all of the specified requirements have been met to consider the OpenIDM system ready
- `ACTIVE_NOT_READY` - one or more of the specified requirements have not been met and the OpenIDM system is not considered ready
- `STOPPING` - OpenIDM is shutting down

You can verify the current state of an OpenIDM system with the following REST call:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/info/ping"
{
  "_id" : "",
  "state" : "ACTIVE_READY",
  "shortDesc" : "OpenIDM ready"
}
```

The information is provided by the following script: `openidm/bin/defaults/script/info/ping.js`.

2.3.2. Getting Current Session Information

You can get more information about the current OpenIDM session, beyond basic health checks, with the following REST call:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
"http://localhost:8080/openidm/info/login"
{
  "_id" : "",
  "class" : "org.forgerock.services.context.SecurityContext",
  "name" : "security",
  "authenticationId" : "openidm-admin",
  "authorization" : {
    "id" : "openidm-admin",
    "component" : "repo/internal/user",
    "roles" : [ "openidm-admin", "openidm-authorized" ],
    "ipAddress" : "127.0.0.1"
  },
  "parent" : {
    "class" : "org.forgerock.caf.authentication.framework.MessageContextImpl",
    "name" : "jaspi",
    "parent" : {
      "class" : "org.forgerock.services.context.TransactionIdContext",
      "id" : "2b4ab479-3918-4138-b018-1a8fa01bc67c-288",
      "name" : "transactionId",
      "transactionId" : {
        "value" : "2b4ab479-3918-4138-b018-1a8fa01bc67c-288",
        "subTransactionIdCounter" : 0
      },
      "parent" : {
        "class" : "org.forgerock.services.context.ClientContext",
        "name" : "client",
        "remoteUser" : null,
        "remoteAddress" : "127.0.0.1",
        "remoteHost" : "127.0.0.1",
        "remotePort" : 56534,
        "certificates" : ""
,
...
```

The information is provided by the following script: `openidm/bin/defaults/script/info/login.js`.

2.3.3. Monitoring Tuning and Health Parameters

You can extend OpenIDM monitoring beyond what you can check on the `openidm/info/ping` and `openidm/info/login` endpoints. Specifically, you can get more detailed information about the state of the:

- `Operating System` on the `openidm/health/os` endpoint
- `Memory` on the `openidm/health/memory` endpoint
- `JDBC Pooling`, based on the `openidm/health/jdbc` endpoint
- `Reconciliation`, on the `openidm/health/recon` endpoint.

You can regulate access to these endpoints as described in the following section: "Understanding the Access Configuration Script (`access.js`)".

2.3.3.1. Operating System Health Check

With the following REST call, you can get basic information about the host operating system:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/health/os"
{
    "_id" : "",
    "_rev" : "",
    "availableProcessors" : 1,
    "systemLoadAverage" : 0.06,
    "operatingSystemArchitecture" : "amd64",
    "operatingSystemName" : "Linux",
    "operatingSystemVersion" : "2.6.32-504.30.3.el6.x86_64"
}
```

From the output, you can see that this particular system has one 64-bit CPU, with a load average of 6 percent, on a Linux system with the noted kernel `operatingSystemVersion` number.

2.3.3.2. Memory Health Check

With the following REST call, you can get basic information about overall JVM memory use:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/health/memory"
{
    "_id" : "",
    "_rev" : "",
    "objectPendingFinalization" : 0,
    "heapMemoryUsage" : {
        "init" : 1073741824,
        "used" : 88538392,
        "committed" : 1037959168,
        "max" : 1037959168
    },
    "nonHeapMemoryUsage" : {
        "init" : 24313856,
        "used" : 69255024,
        "committed" : 69664768,
        "max" : 224395264
    }
}
```

The output includes information on JVM Heap and Non-Heap memory, in bytes. Briefly:

- JVM Heap memory is used to store Java objects.
- JVM Non-Heap Memory is used by Java to store loaded classes and related meta-data

2.3.3.3. JDBC Health Check

Running a health check on the JDBC repository is supported only if you are using the BoneCP connection pool. This is not the default connection pool, so you must make the following changes to your configuration before running this command:

- In your project's `conf/datasource.jdbc-default.json` file, change the `connectionPool` parameter as follows:

  ```
  "connectionPool" : {
      "type" : "bonecp"
  }
  ```

- In your project's `conf/boot/boot.properties` file, enable the statistics MBean for the BoneCP connection pool:

  ```
  openidm.bonecp.statistics.enabled=true
  ```

For a BoneCP connection pool, the following REST call returns basic information about the status of the JDBC repository:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/health/jdbc"
{
  "_id": "",
  "_rev": "",
  "com.jolbox.bonecp:type=BoneCP-4ffa60bd-5dfc-400f-850e-439c7aa27094": {
    "connectionWaitTimeAvg": 0.012701142857142857,
    "statementExecuteTimeAvg": 0.8084880967741935,
    "statementPrepareTimeAvg": 1.6652538867562894,
    "totalLeasedConnections": 0,
    "totalFreeConnections": 7,
    "totalCreatedConnections": 7,
    "cacheHits": 0,
    "cacheMiss": 0,
    "statementsCached": 0,
    "statementsPrepared": 31,
    "connectionsRequested": 28,
    "cumulativeConnectionWaitTime": 0,
    "cumulativeStatementExecutionTime": 25,
    "cumulativeStatementPrepareTime": 18,
    "cacheHitRatio": 0,
    "statementsExecuted": 31
  }
}
```

The BoneCP metrics are self-explanatory.

2.3.3.4. Reconciliation Health Check

With the following REST call, you can get basic information about the system demands related to reconciliation:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/health/recon"
{
    "_id" : "",
    "_rev" : "",
    "activeThreads" : 1,
    "corePoolSize" : 10,
    "largestPoolSize" : 1,
    "maximumPoolSize" : 10,
    "currentPoolSize" : 1
}
```

From the output, you can review the number of active threads used by the reconciliation, as well as the available thread pool.

2.3.4. Customizing Health Check Scripts

You can extend or override the default information that is provided by creating your own script file and its corresponding configuration file in `openidm/conf/info-name.json`. Custom script files can be located anywhere, although a best practice is to place them in `openidm/script/info`. A sample customized script file for extending the default ping service is provided in `openidm/samples/infoservice/script/info/customping.js`. The corresponding configuration file is provided in `openidm/samples/infoservice/conf/info-customping.json`.

The configuration file has the following syntax:

```
{
    "infocontext" : "ping",
    "type" : "text/javascript",
    "file" : "script/info/customping.js"
}
```

The parameters in the configuration file are as follows:

- `infocontext` specifies the relative name of the info endpoint under the info context. The information can be accessed over REST at this endpoint, for example, setting `infocontext` to `mycontext/myendpoint` would make the information accessible over REST at `http://localhost:8080/openidm/info/mycontext/myendpoint`.

- `type` specifies the type of the information source. JavaScript (`"type" : "text/javascript"`) and Groovy (`"type" : "groovy"`) are supported.

- `file` specifies the path to the JavaScript or Groovy file, if you do not provide a `"source"` parameter.

- `source` specifies the actual JavaScript or Groovy script, if you have not provided a `"file"` parameter.

Additional properties can be passed to the script as depicted in this configuration file (`openidm/samples/infoservice/conf/info-name.json`).

Script files in `openidm/samples/infoservice/script/info/` have access to the following objects:

- `request` - the request details, including the method called and any parameters passed.
- `healthinfo` - the current health status of the system.
- `openidm` - access to the JSON resource API.
- Any additional properties that are depicted in the configuration file (`openidm/samples/infoservice/conf/info-name.json`.)

2.3.5. Verifying the State of Health Check Service Modules

The configurable OpenIDM health check service can verify the status of required modules and services for an operational system. During system startup, OpenIDM checks that these modules and services are available and reports on whether any requirements for an operational system have not been met. If dynamic configuration changes are made, OpenIDM rechecks that the required modules and services are functioning, to allow ongoing monitoring of system operation.

Examples of Required Modules

OpenIDM checks all required modules. Examples of those modules are shown here:

```
    "org.forgerock.openicf.framework.connector-framework"
    "org.forgerock.openicf.framework.connector-framework-internal"
    "org.forgerock.openicf.framework.connector-framework-osgi"
    "org.forgerock.openidm.audit"
    "org.forgerock.openidm.core"
    "org.forgerock.openidm.enhanced-config"
    "org.forgerock.openidm.external-email"
    ...
    "org.forgerock.openidm.system"
    "org.forgerock.openidm.ui"
    "org.forgerock.openidm.util"
    "org.forgerock.commons.org.forgerock.json.resource"
    "org.forgerock.commons.org.forgerock.util"
    "org.forgerock.openidm.security-jetty"
    "org.forgerock.openidm.jetty-fragment"
    "org.forgerock.openidm.quartz-fragment"
    "org.ops4j.pax.web.pax-web-extender-whiteboard"
    "org.forgerock.openidm.scheduler"
    "org.ops4j.pax.web.pax-web-jetty-bundle"
    "org.forgerock.openidm.repo-jdbc"
    "org.forgerock.openidm.repo-orientdb"
    "org.forgerock.openidm.config"
    "org.forgerock.openidm.crypto"
```

Examples of Required Services

OpenIDM checks all required services. Examples of those services are shown here:

```
    "org.forgerock.openidm.config"
    "org.forgerock.openidm.provisioner"
    "org.forgerock.openidm.provisioner.openicf.connectorinfoprovider"
    "org.forgerock.openidm.external.rest"
    "org.forgerock.openidm.audit"
    "org.forgerock.openidm.policy"
    "org.forgerock.openidm.managed"
    "org.forgerock.openidm.script"
    "org.forgerock.openidm.crypto"
    "org.forgerock.openidm.recon"
    "org.forgerock.openidm.info"
    "org.forgerock.openidm.router"
    "org.forgerock.openidm.scheduler"
    "org.forgerock.openidm.scope"
    "org.forgerock.openidm.taskscanner"
```

You can replace the list of required modules and services, or add to it, by adding the following lines to your project's `conf/boot/boot.properties` file. Bundles and services are specified as a list of symbolic names, separated by commas:

- `openidm.healthservice.reqbundles` - overrides the default required bundles.
- `openidm.healthservice.reqservices` - overrides the default required services.
- `openidm.healthservice.additionalreqbundles` - specifies required bundles (in addition to the default list).
- `openidm.healthservice.additionalreqservices` - specifies required services (in addition to the default list).

By default, OpenIDM gives the system 15 seconds to start up all the required bundles and services, before the system readiness is assessed. Note that this is not the total start time, but the time required to complete the service startup after the framework has started. You can change this default by setting the value of the `servicestartmax` property (in milliseconds) in your project's `conf/boot/boot.properties` file. This example sets the startup time to five seconds:

```
openidm.healthservice.servicestartmax=5000
```

2.4. Displaying Information About Installed Modules

On a running OpenIDM instance, you can list the installed modules and their states by typing the following command in the OSGi console. (The output will vary by configuration):

```
-> scr list

   Id   State          Name
[  12] [active       ] org.forgerock.openidm.endpoint
[  13] [active       ] org.forgerock.openidm.endpoint
[  14] [active       ] org.forgerock.openidm.endpoint
[  15] [active       ] org.forgerock.openidm.endpoint
[  16] [active       ] org.forgerock.openidm.endpoint
      ...
[  34] [active       ] org.forgerock.openidm.taskscanner
[  20] [active       ] org.forgerock.openidm.external.rest
[   6] [active       ] org.forgerock.openidm.router
[  33] [active       ] org.forgerock.openidm.scheduler
[  19] [unsatisfied  ] org.forgerock.openidm.external.email
[  11] [active       ] org.forgerock.openidm.sync
[  25] [active       ] org.forgerock.openidm.policy
[   8] [active       ] org.forgerock.openidm.script
[  10] [active       ] org.forgerock.openidm.recon
[   4] [active       ] org.forgerock.openidm.http.contextregistrator
[   1] [active       ] org.forgerock.openidm.config
[  18] [active       ] org.forgerock.openidm.endpointservice
[  30] [unsatisfied  ] org.forgerock.openidm.servletfilter
[  24] [active       ] org.forgerock.openidm.infoservice
[  21] [active       ] org.forgerock.openidm.authentication
->
```

To display additional information about a particular module or service, run the following command, substituting the `Id` of that module from the preceding list:

```
-> scr info Id
```

The following example displays additional information about the router service:

```
-> scr info 9
ID: 9
Name: org.forgerock.openidm.router
Bundle: org.forgerock.openidm.api-servlet (127)
State: active
Default State: enabled
Activation: immediate
Configuration Policy: optional
Activate Method: activate (declared in the descriptor)
Deactivate Method: deactivate (declared in the descriptor)
Modified Method: -
Services: org.forgerock.json.resource.ConnectionFactory
          java.io.Closeable
          java.lang.AutoCloseable
Service Type: service
Reference: requestHandler
    Satisfied: satisfied
    Service Name: org.forgerock.json.resource.RequestHandler
    Target Filter: (org.forgerock.openidm.router=*)
    Multiple: single
    Optional: mandatory
    Policy: static
...
Properties:
    component.id = 9
    component.name = org.forgerock.openidm.router
    felix.fileinstall.filename = file:/path/to/openidm-latest/conf/router.json
    jsonconfig = {
    "filters" : [
        {
            "condition" : {
                "type" : "text/javascript",
                "source" : "context.caller.external === true || context.current.name === 'selfservice'"
            },
            "onRequest" : {
                "type" : "text/javascript",
                "file" : "router-authz.js"
            }
        },
        {
            "pattern" : "^(managed|system|repo/internal)($|(/.+))",
            "onRequest" : {
                "type" : "text/javascript",
                "source" : "require('policyFilter').runFilter()"
            },
            "methods" : [
                "create",
                "update"
            ]
        },
        {
            "pattern" : "repo/internal/user.*",
            "onRequest" : {
                "type" : "text/javascript",
                "source" : "request.content.password = require('crypto').hash(request.content.password);"
            },
            "methods" : [
                "create",
                "update"
            ]
        }
    ]
}
    maintenanceFilter.target = (service.pid=org.forgerock.openidm.maintenance)
    requestHandler.target = (org.forgerock.openidm.router=*)
    service.description = OpenIDM Common REST Servlet Connection Factory
    service.pid = org.forgerock.openidm.router
    service.vendor = ForgeRock AS.
->
```

2.5. Starting in Debug Mode

To debug custom libraries, you can start OpenIDM with the option to use the Java Platform Debugger Architecture (JPDA):

- Start OpenIDM with the `jpda` option:

```
$ cd /path/to/openidm
$ ./startup.sh jpda
Executing ./startup.sh...
Using OPENIDM_HOME:   /path/to/openidm
Using OPENIDM_OPTS:   -Xmx1024m -Xms1024m -Denvironment=PROD -Djava.compiler=NONE
   -Xnoagent -Xdebug -Xrunjdwp:transport=dt_socket,address=5005,server=y,suspend=n
Using LOGGING_CONFIG:
   -Djava.util.logging.config.file=/path/to/openidm/conf/logging.properties
Listening for transport dt_socket at address: 5005
Using boot properties at /path/to/openidm/conf/boot/boot
.properties
-> OpenIDM version "5.0.0" (revision: xxxx)
OpenIDM ready
```

The relevant JPDA options are outlined in the startup script (`startup.sh`).

- In your IDE, attach a Java debugger to the JVM via socket, on port 5005.

Caution

This interface is internal and subject to change. If you depend on this interface, contact ForgeRock support.

2.6. Running As a Service on Linux Systems

OpenIDM provides a script that generates an initialization script to run OpenIDM as a service on Linux systems. You can start the script as the root user, or configure it to start during the boot process.

When OpenIDM runs as a service, logs are written to the directory in which OpenIDM was installed.

To run OpenIDM as a service, take the following steps:

1. If you have not yet installed OpenIDM, follow the procedure described in "*Preparing to Install and Run Servers*" in the *Installation Guide*.

2. Run the RC script:

   ```
   $ cd /path/to/openidm/bin
   $ ./create-openidm-rc.sh
   ```

3. As a user with administrative privileges, copy the `openidm` script to the `/etc/init.d` directory:

   ```
   $ sudo cp openidm /etc/init.d/
   ```

4. If you run Linux with SELinux enabled, change the file context of the newly copied script with the following command:

   ```
   $ sudo restorecon /etc/init.d/openidm
   ```

You can verify the change to SELinux contexts with the `ls -Z /etc/init.d` command. For consistency, change the user context to match other scripts in the same directory with the `sudo chcon -u system_u /etc/init.d/openidm` command.

5. Run the appropriate commands to add OpenIDM to the list of RC services:

   - On Red Hat-based systems, run the following commands:

     ```
     $ sudo chkconfig --add openidm
     ```

     ```
     $ sudo chkconfig openidm on
     ```

   - On Debian/Ubuntu systems, run the following command:

     ```
     $ sudo update-rc.d openidm defaults
     Adding system startup for /etc/init.d/openidm ..
     .
     /etc/rc0.d/K20openidm -> ../init.d/
     openidm
     /etc/rc1.d/K20openidm -> ../init.d/
     openidm
     /etc/rc6.d/K20openidm -> ../init.d/
     openidm
     /etc/rc2.d/S20openidm -> ../init.d/
     openidm
     /etc/rc3.d/S20openidm -> ../init.d/
     openidm
     /etc/rc4.d/S20openidm -> ../init.d/
     openidm
     /etc/rc5.d/S20openidm -> ../init.d/openidm
     ```

     Note the output, as Debian/Ubuntu adds start and kill scripts to appropriate runlevels.

     When you run the command, you may get the following warning message: `update-rc.d: warning: /etc/init.d/openidm missing LSB information`. You can safely ignore that message.

6. As an administrative user, start the OpenIDM service:

   ```
   $ sudo /etc/init.d/openidm start
   ```

   Alternatively, reboot the system to start the OpenIDM service automatically.

7. (Optional) The following commands stops and restarts the service:

   ```
   $ sudo /etc/init.d/openidm stop
   ```

   ```
   $ sudo /etc/init.d/openidm restart
   ```

If you have set up a deployment of OpenIDM in a custom directory, such as `/path/to/openidm/production`, you can modify the `/etc/init.d/openidm` script.

Open the `openidm` script in a text editor and navigate to the `START_CMD` line.

At the end of the command, you should see the following line:

```
org.forgerock.commons.launcher.Main -c bin/launcher.json > logs/server.out 2>&1 &"
```

Include the path to the production directory. In this case, you would add **-p production** as shown:

```
org.forgerock.commons.launcher.Main -c bin/launcher.json -p production > logs/server.out 2>&1 &
```

Save the `openidm` script file in the `/etc/init.d` directory. The **sudo /etc/init.d/openidm start** command should now start OpenIDM with the files in your `production` subdirectory.

Chapter 3
Command-Line Interface

This chapter describes the basic command-line interface (CLI). The CLI includes a number of utilities for managing an OpenIDM instance.

All of the utilities are subcommands of the `cli.sh` (UNIX) or `cli.bat` (Windows) scripts. To use the utilities, you can either run them as subcommands, or launch the **cli** script first, and then run the utility. For example, to run the **encrypt** utility on a UNIX system:

```
$ cd /path/to/openidm
$ ./cli.sh
Using boot properties at /path/to/openidm/conf/boot/boot.properties
openidm# encrypt ....
```

or

```
$ cd /path/to/openidm
$ ./cli.sh encrypt ...
```

By default, the command-line utilities run with the properties defined in your project's `conf/boot/boot.properties` file.

If you run the **cli.sh** command by itself, it opens an OpenIDM-specific shell prompt:

```
openidm#
```

The startup and shutdown scripts are not discussed in this chapter. For information about these scripts, see "*Starting and Stopping the Server*".

The following sections describe the subcommands and their use. Examples assume that you are running the commands on a UNIX system. For Windows systems, use **cli.bat** instead of **cli.sh**.

For a list of subcommands available from the `openidm#` prompt, run the **cli.sh help** command. The **help** and **exit** options shown below are self-explanatory. The other subcommands are explained in the subsections that follow:

```
local:keytool  Export or import a SecretKeyEntry.
    The Java Keytool does not allow for exporting or importing SecretKeyEntries.
local:encrypt    Encrypt the input string.
local:secureHash   Hash the input string.
local:validate   Validates all json configuration files in the configuration
    (default: /conf) folder.
basic:help   Displays available commands.
basic:exit   Exit from the console.
remote:update              Update the system with the provided update file.
remote:configureconnector   Generate connector configuration.
remote:configexport         Exports all configurations.
remote:configimport         Imports the configuration set from local file/directory.
```

The following options are common to the **configexport**, **configimport**, and **configureconnector** subcommands:

-u or --user USER[:PASSWORD]

Allows you to specify the server user and password. Specifying a username is mandatory. If you do not specify a username, the following error is output to the OSGi console: `Remote operation failed: Unauthorized`. If you do not specify a password, you are prompted for one. This option is used by all three subcommands.

--url URL

The URL of the OpenIDM REST service. The default URL is `http://localhost:8080/openidm/`. This can be used to import configuration files from a remote running instance of OpenIDM. This option is used by all three subcommands.

-P or --port PORT

The port number associated with the OpenIDM REST service. If specified, this option overrides any port number specified with the **--url** option. The default port is 8080. This option is used by all three subcommands.

3.1. Using the **configexport** Subcommand

The **configexport** subcommand exports all configuration objects to a specified location, enabling you to reuse a system configuration in another environment. For example, you can test a configuration in a development environment, then export it and import it into a production environment. This subcommand also enables you to inspect the active configuration of an OpenIDM instance.

OpenIDM must be running when you execute this command.

Usage is as follows:

```
$ ./cli.sh configexport --user username:password export-location
```

For example:

```
$ ./cli.sh configexport --user openidm-admin:openidm-admin /tmp/conf
```

On Windows systems, the *export-location* must be provided in quotation marks, for example:

```
C:\openidm\cli.bat configexport --user openidm-admin:openidm-admin "C:\temp\openidm"
```

Configuration objects are exported as `.json` files to the specified directory. The command creates the directory if needed. Configuration files that are present in this directory are renamed as backup files, with a timestamp, for example, `audit.json.2014-02-19T12-00-28.bkp`, and are not overwritten. The following configuration objects are exported:

- The internal repository table configuration (`repo.orientdb.json` or `repo.jdbc.json`) and the datasource connection configuration, for JDBC repositories (`datasource.jdbc-default.json`)
- The script configuration (`script.json`)
- The log configuration (`audit.json`)
- The authentication configuration (`authentication.json`)
- The cluster configuration (`cluster.json`)
- The configuration of a connected SMTP email server (`external.email.json)`
- Custom configuration information (`info-`*`name`*`.json`)
- The managed object configuration (`managed.json`)
- The connector configuration (`provisioner.openicf-*.json`)
- The router service configuration (`router.json`)
- The scheduler service configuration (`scheduler.json`)
- Any configured schedules (`schedule-*.json`)
- Standard knowledge-based authentication questions (`selfservice.kba.json)`
- The synchronization mapping configuration (`sync.json`)
- If workflows are defined, the configuration of the workflow engine (`workflow.json`) and the workflow access configuration (`process-access.json`)
- Any configuration files related to the user interface (`ui-*.json`)
- The configuration of any custom endpoints (`endpoint-*.json`)
- The configuration of servlet filters (`servletfilter-*.json`)
- The policy configuration (`policy.json`)

3.2. Using the **configimport** Subcommand

The **configimport** subcommand imports configuration objects from the specified directory, enabling you to reuse a system configuration from another environment. For example, you can test a configuration in a development environment, then export it and import it into a production environment.

The command updates the existing configuration from the *import-location* over the OpenIDM REST interface. By default, if configuration objects are present in the *import-location* and not in the existing configuration, these objects are added. If configuration objects are present in the existing location but not in the *import-location*, these objects are left untouched in the existing configuration.

The subcommand takes the following options:

`-r`, `--replaceall`, `--replaceAll`

Replaces the entire list of configuration files with the files in the specified import location.

Note that this option wipes out the existing configuration and replaces it with the configuration in the *import-location*. Objects in the existing configuration that are not present in the *import-location* are deleted.

`--retries` **(integer)**

New in OpenIDM 5.0.0, this option specifies the number of times the command should attempt to update the configuration if OpenIDM is not ready.

Default value : 10

`--retryDelay` **(integer)**

New in OpenIDM 5.0.0, this option specifies the delay (in milliseconds) between configuration update retries if OpenIDM is not ready.

Default value : 500

Usage is as follows:

```
$ ./cli.sh configimport --user username:password [--replaceAll] [--retries integer] [--
retryDelay integer] import-location
```

For example:

```
$ ./cli.sh configimport --user openidm-admin:openidm-admin --retries 5 --retryDelay 250 --replaceAll /tmp/
conf
```

On Windows systems, the *import-location* must be provided in quotation marks, for example:

```
C:\openidm\cli.bat configimport --user openidm-admin:openidm-admin --replaceAll "C:\temp\openidm"
```

Configuration objects are imported as `.json` files from the specified directory to the `conf` directory. The configuration objects that are imported are the same as those for the **export** command, described in the previous section.

3.3. Using the **configureconnector** Subcommand

The **configureconnector** subcommand generates a configuration for an OpenICF connector.

Usage is as follows:

```
$ ./cli.sh configureconnector --user username:password --name connector-name
```

Select the type of connector that you want to configure. The following example configures a new XML connector:

```
$ ./cli.sh configureconnector --user openidm-admin:openidm-admin --name myXmlConnector
 Starting shell in /path/to/openidm
Using boot properties at /path/to/openidm/conf/boot/boot.properties
0. XML Connector version 1.1.0.3
1. SSH Connector version 1.4.1.0
2. LDAP Connector version 1.4.3.0
3. Kerberos Connector version 1.4.2.0
4. Scripted SQL Connector version 1.4.3.0
5. Scripted REST Connector version 1.4.3.0
6. Scripted CREST Connector version 1.4.3.0
7. Scripted Poolable Groovy Connector version 1.4.3.0
8. Scripted Groovy Connector version 1.4.3.0
9. Database Table Connector version 1.1.0.2
10. CSV File Connector version 1.5.1.4
11. Exit
Select [0..11]: 0
Edit the configuration file and run the command again. The configuration was
saved to /openidm/temp/provisioner.openicf-myXmlConnector.json
```

The basic configuration is saved in a file named `/openidm/temp/provisioner.openicf-connector-name.json`. Edit the `configurationProperties` parameter in this file to complete the connector configuration. For an XML connector, you can use the schema definitions in Sample 1 for an example configuration:

```

  "configurationProperties" : {
    "xmlFilePath" : "samples/sample1/data/resource-schema-1.xsd",
    "createFileIfNotExists" : false,
    "xsdFilePath" : "samples/sample1/data/resource-schema-extension.xsd",
    "xsdIcfFilePath" : "samples/sample1/data/xmlConnectorData.xml"
  },

```

For more information about the connector configuration properties, see "Configuring Connectors".

When you have modified the file, run the **configureconnector** command again so that OpenIDM can pick up the new connector configuration:

```
$ ./cli.sh configureconnector --user openidm-admin:openidm-admin --name myXmlConnector
Executing ./cli.sh...
Starting shell in /path/to/openidm
Using boot properties at /path/to/openidm/conf/boot/boot.properties
Configuration was found and read from: /path/to/openidm/temp/provisioner.openicf-myXmlConnector.json
```

You can now copy the new `provisioner.openicf-myXmlConnector.json` file to the `conf/` subdirectory.

You can also configure connectors over the REST interface, or through the Admin UI. For more information, see "Creating Default Connector Configurations" and "Adding New Connectors from the Admin UI".

3.4. Using the **encrypt** Subcommand

The **encrypt** subcommand encrypts an input string, or JSON object, provided at the command line. This subcommand can be used to encrypt passwords, or other sensitive data, to be stored in the OpenIDM repository. The encrypted value is output to standard output and provides details of the cryptography key that is used to encrypt the data.

Usage is as follows:

```
$ ./cli.sh encrypt [-j] string
```

If you do not enter the string as part of the command, the command prompts for the string to be encrypted. If you enter the string as part of the command, any special characters, for example quotation marks, must be escaped.

The `-j` option indicates that the string to be encrypted is a JSON object, and validates the object. If the object is malformed JSON and you use the `-j` option, the command throws an error. It is easier to input JSON objects in interactive mode. If you input the JSON object on the command-line, the object must be surrounded by quotes and any special characters, including curly braces, must be escaped. The rules for escaping these characters are fairly complex. For more information, see section 4.8.2 of the OSGi draft specification. For example:

```
$ ./cli.sh encrypt -j '\{\"password\":\"myPassw0rd\"\}'
```

The following example encrypts a normal string value:

```
$ ./cli.sh encrypt mypassword
Executing ./cli.sh...
Starting shell in /path/to/openidm
Using boot properties at /path/to/openidm/conf/boot/boot
.properties
-----BEGIN ENCRYPTED VALUE-----
{
  "$crypto" : {
    "type" : "x-simple-encryption",
    "value" : {
      "cipher" : "AES/CBC/PKCS5Padding",
      "salt" : "0pRncNLTJ6ZySHfV4DEtgA==",
      "data" : "pIrCCkLPhBt0rbGXiZBHkw==",
      "iv" : "l1Hau6nf2zizQSib8kkW0g==",
      "key" : "openidm-sym-default",
      "mac" : "SoqfhpvhBVuIkux8mztpeQ=="
    }
  }
}
------END ENCRYPTED VALUE------
```

The following example prompts for a JSON object to be encrypted:

```
$ ./cli.sh encrypt -j
Using boot properties at /path/to/openidm/conf/boot/boot.properties
Enter the Json value

> Press ctrl-D to finish input
Start data input:
{"password":"myPassw0rd"}
^D
-----BEGIN ENCRYPTED VALUE-----
{
  "$crypto" : {
    "type" : "x-simple-encryption",
    "value" : {
      "cipher" : "AES/CBC/PKCS5Padding",
      "salt" : "vdz6bUztiT6QsExNrZQAEA==",
      "data" : "RgMLRbX0guxF80nwrtaZkkoFFGqSQdNWF7Ve0zS+N1I=",
      "iv" : "R9w1TcWfbd9FPm0jfvMhZQ==",
      "key" : "openidm-sym-default",
      "mac" : "9pXtSKAt9+d03Mu0NlrJsQ=="
    }
  }
}
------END ENCRYPTED VALUE------
```

3.5. Using the **secureHash** Subcommand

The **secureHash** subcommand hashes an input string, or JSON object, using the specified hash algorithm. This subcommand can be used to hash password values, or other sensitive data, to be stored in the OpenIDM repository. The hashed value is output to standard output and provides details of the algorithm that was used to hash the data.

Usage is as follows:

```
$ ./cli.sh secureHash --algorithm [-j] string
```

The `-a` or `--algorithm` option specifies the hash algorithm to use. OpenIDM supports the following hash algorithms: `MD5`, `SHA-1`, `SHA-256`, `SHA-384`, and `SHA-512`. If you do not specify a hash algorithm, `SHA-256` is used.

If you do not enter the string as part of the command, the command prompts for the string to be hashed. If you enter the string as part of the command, any special characters, for example quotation marks, must be escaped.

The `-j` option indicates that the string to be hashed is a JSON object, and validates the object. If the object is malformed JSON and you use the `-j` option, the command throws an error. It is easier to input JSON objects in interactive mode. If you input the JSON object on the command-line, the object must be surrounded by quotes and any special characters, including curly braces, must be escaped. The rules for escaping these characters are fairly complex. For more information, see section 4.8.2 of the OSGi draft specification. For example:

```
$ ./cli.sh secureHash --algorithm SHA-1 '\{\"password\":\"myPassw0rd\"\}'
```

The following example hashes a password value (`mypassword`) using the `SHA-1` algorithm:

```
$ ./cli.sh secureHash --algorithm SHA-1 mypassword
Executing ./cli.sh...
Starting shell in /path/to/openidm
Using boot properties at /path/to/openidm/conf/boot/boot
.properties
-----BEGIN HASHED VALUE-----
{
  "$crypto" : {
    "value" : {
      "algorithm" : "SHA-1",
      "data" : "T9yf3dL7oepWvUPbC8kb4hEmKJ7g5Zd43ndORYQox3GiWAGU"
    },
    "type" : "salted-hash"
  }
}
------END HASHED VALUE------
```

The following example prompts for a JSON object to be hashed:

```
$ ./cli.sh secureHash --algorithm SHA-1 -j
Executing ./cli.sh...
Starting shell in /path/to/openidm
Using boot properties at /path/to/openidm/conf/boot/boot.properties
Enter the Json value

> Press ctrl-D to finish input
Start data input:
{"password":"myPassw0rd"}
^D
-----BEGIN HASHED VALUE-----
{
  "$crypto" : {
    "value" : {
      "algorithm" : "SHA-1",
      "data" : "PBsmFJZEVNHuYPZJwaF5oX0LtamUA2tikFCiQEfgIsqa/VHK"
    },
    "type" : "salted-hash"
  }
}
------END HASHED VALUE------
```

3.6. Using the **keytool** Subcommand

The **keytool** subcommand exports or imports secret key values.

The Java **keytool** command enables you to export and import public keys and certificates, but not secret or symmetric keys. The OpenIDM **keytool** subcommand provides this functionality.

Usage is as follows:

```
$ ./cli.sh keytool [--export, --import] alias
```

For example, to export the default OpenIDM symmetric key, run the following command:

```
$ ./cli.sh keytool --export openidm-sym-default
   Using boot properties at /openidm/conf/boot/boot.properties
Use KeyStore from: /openidm/security/keystore.jceks
Please enter the password:
[OK] Secret key entry with algorithm AES
AES:606d80ae316be58e94439f91ad8ce1c0
```

The default keystore password is `changeit`. For security reasons, you *must* change this password in a production environment. For information about changing the keystore password, see "Change the Default Keystore Password".

To import a new secret key named *my-new-key*, run the following command:

```
$ ./cli.sh keytool --import my-new-key
Using boot properties at /openidm/conf/boot/boot.properties
Use KeyStore from: /openidm/security/keystore.jceks
Please enter the password:
Enter the key:
AES:606d80ae316be58e94439f91ad8ce1c0
```

If a secret key of that name already exists, OpenIDM returns the following error:

```
"KeyStore contains a key with this alias"
```

3.7. Using the **validate** Subcommand

The **validate** subcommand validates all .json configuration files in your project's `conf/` directory.

Usage is as follows:

```
$ ./cli.sh validate
Executing ./cli.sh
Starting shell in /path/to/openidm
Using boot properties at /path/to/openidm/conf/boot/boot
.properties
...................................................................
[Validating] Load JSON configuration files from:
[Validating]  /path/to/openidm/conf
[Validating] audit.json .................................. SUCCESS
[Validating] authentication.json ......................... SUCCESS
    ...
[Validating] sync.json ................................... SUCCESS
[Validating] ui-configuration.json ....................... SUCCESS
[Validating] ui-countries.json ........................... SUCCESS
[Validating] workflow.json ............................... SUCCESS
```

3.8. Using the **update** Subcommand

The **update** subcommand supports updates of OpenIDM for patches and migrations. For an example of this process, see "*Updating Servers*" in the *Installation Guide*.

Chapter 4

Web-Based User Interfaces

OpenIDM includes a customizable, browser-based user interface. The functionality is subdivided into Administrative and Self-Service User Interfaces.

If you are administering OpenIDM, navigate to the Administrative User Interface, also known as the Admin UI. If OpenIDM is installed on the local system, you can get to the Admin UI at the following URL: `https://localhost:8443/admin`. In the Admin UI, you can configure connectors, customize managed objects, set up attribute mappings, manage accounts, and more.

The Self-Service User Interface, also known as the Self-Service UI, provides role-based access to tasks based on BPMN2 workflows, and allows users to manage certain aspects of their own accounts, including configurable self-service registration. When OpenIDM starts, you can access the Self-Service UI at `https://localhost:8443/`.

Warning

The default password for the administrative user, `openidm-admin`, is `openidm-admin`. To protect your deployment in production, change this password.

All users, including `openidm-admin`, can change their password through the Self-Service UI. After you have logged in, click Change Password.

4.1. Configuring the Server from the Admin UI

You can set up a basic configuration with the Administrative User Interface (Admin UI).

Through the Admin UI, you can connect to resources, configure attribute mapping and scheduled reconciliation, and set up and manage objects, such as users, groups, and devices.

You can configure OpenIDM through Quick Start cards, and from the Configure and Manage drop-down menus. Try them out, and see what happens when you select each option.

In the following sections, you will examine the default Admin UI dashboard, and learn how to set up custom Admin UI dashboards.

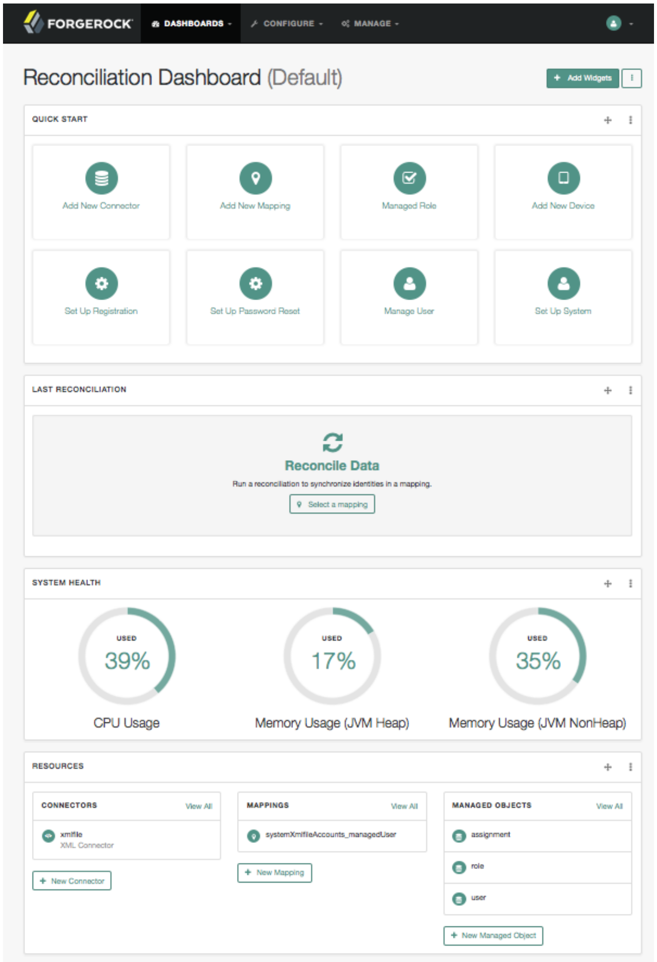
Caution

If your browser uses an AdBlock extension, it might inadvertently block some UI functionality, particularly if your configuration includes strings such as `ad`. For example, a connection to an Active Directory server might be configured at the endpoint `system/ad`. To avoid problems related to blocked UI functionality, either remove the AdBlock extension, or set up a suitable white list to ensure that none of the targeted endpoints are blocked.

4.1.1. Default Admin UI Dashboard

When you log into the Admin UI, the first screen you should see is the "Reconciliation Dashboard".

The Administrative UI Reconciliation Dashboard

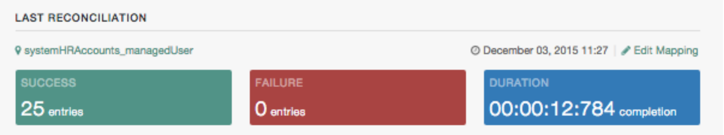


Run a reconciliation to synchronize identities in a mapping.

The Admin UI includes a fixed top menu bar. As you navigate around the Admin UI, you should see the same menu bar throughout. You can click the Dashboards > Reconciliation Dashboard to return to that screen.

The default dashboard is split into four sections, based on widgets.

- Quick Start cards support one-click access to common administrative tasks, and are described in detail in the following section.
- Last Reconciliation includes data from the most recent reconciliation between data stores. After you run a reconciliation, you should see data similar to:



LAST RECONCILIATION

systemHRAccounts_managedUser

December 03, 2015 11:27 | Edit Mapping

SUCCESS
25 entries

FAILURE
0 entries

DURATION
00:00:12:784 completion

- System Health includes data on current CPU and memory usage.
- Resources include an abbreviated list of configured connectors, mappings, and managed objects.

The `Quick Start` cards allow quick access to the labeled configuration options, described here:

- `Add Connector`

  Use the Admin UI to connect to external resources. For more information, see "Adding New Connectors from the Admin UI".

- `Create Mapping`

  Configure synchronization mappings to map objects between resources. For more information, see "Mapping Source Objects to Target Objects".

- `Manage Role`

  Set up managed provisioning or authorization roles. For more information, see "Working With Managed Roles".

- `Add Device`

  Use the Admin UI to set up managed objects, including users, groups, roles, or even Internet of Things (IoT) devices. For more information, see "Managing Accounts".

- `Set Up Registration`

  Configure User Self-Registration. You can set up the Self-Service UI login screen, with a link that allows new users to start a verified account registration process. For more information, see "*Configuring User Self-Service*".

- `Set Up Password Reset`

  Configure user self-service Password Reset, allowing end-users to reset forgotten passwords. For more information, see "*Configuring User Self-Service*".

- `Manage User`

Allows management of users in the repository. You may have to run a reconciliation from an external repository first. For more information, see "Working with Managed Users".

- `Set Up System`

  Configure the following server elements:

  - Authentication, as described in "Supported Authentication and Session Modules".
  - Audit, as described in "*Logging Audit Information*".
  - Self-Service UI, as described in "Changing the UI Path".
  - Email, as described in "*Configuring Outbound Email*".
  - Updates, as described in "*Updating Servers*" in the *Installation Guide*.

4.1.2. Creating and Modifying Dashboards

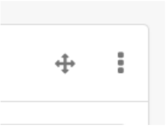
To create a new dashboard, click Dashboards > New Dashboard. You're prompted for a dashboard name, and whether to set it as the default. You can then add widgets.

Alternatively, you can start with an existing dashboard. In the upper-right corner of the UI, next to the Add Widgets button, click the vertical ellipsis. In the menu that appears, you can take the following actions on the current dashboard:

- Rename
- Duplicate
- Set as Default
- Delete

To add a widget to a dashboard, click Add Widgets and add the widget of your choice in the window that appears.

To modify the position of a widget in a dashboard, click and drag on the move icon for the widget. You can find that four arrow icon in the upper right corner of the widget window, next to the three dot vertical ellipsis.



If you add a new Quick Start widget, select the vertical ellipsis in the upper right corner of the widget, and click Settings. You can configure an Admin UI sub-widget to embed in the Quick Start widget in the pop-up menu that appears.

Click Add a Link. You can then enter a name, a *destination URL*, and an icon for the widget.

If you are linking to a specific page in the OpenIDM Admin UI, the destination URL can be the part of the address after the main page for the Admin UI, such as `https://localhost:8443/admin`

For example, if you want to create a quick start link to the Audit configuration tab, at `https://localhost:8443/admin/#settings/audit/`, you could enter `#settings/audit` in the destination URL text box.

OpenIDM writes the changes you make to the `ui-dashboard.json` file for your project.

For example, if you add a Last Reconciliation and Embed Web Page widget to a new dashboard named Test, you'll see the following excerpt in your `ui-dashboard.json` file:

```
            {
    "name" : "Test",
    "isDefault" : false,
    "widgets" : [
        {
            "type" : "frame",
            "size" : "large",
            "frameUrl" : "http://example.com",
            "height" : "100px",
            "title" : "Example.com"
        },
        {
            "type" : "lastRecon",
            "size" : "large",
            "barchart" : "true"
        },
        {
            "type" : "quickStart",
             "size" : "large",
            "cards" : [
                {
                    "name" : "Audit",
                    "icon" : "fa-align-justify",
                    "href" : "#settings/audit"
                }
            ]
        },
    ]
}
```

For more information on each property, see the following table:

Admin UI Widget Properties in `ui-dashboard.json`

Property	Options	Description
`name`	User entry	Dashboard name
`isDefault`	`true` or `false`	Default dashboard; can set one default
`widgets`	Different options for `type`	Code blocks that define a widget

Property	Options	Description
`type`	`lifeCycleMemoryHeap`, `lifeCycleMemoryNonHeap`, `systemHealthFull`, `cpuUsage`, `lastRecon`, `resourceList`, `quickStart`, `frame`, `userRelationship`	Widget name
`size`	`x-small`, `small`, `medium`, or `large`	Width of widget, based on a 12-column grid system, where x-small=4, small=6, medium=8, and large=12; for more information, see Bootstrap CSS
`height`	Height, in units such as `cm`, `mm`, `px`, and `in`	Height; applies only to Embed Web Page widget
`frameUrl`	URL	Web page to embed; applies only to Embed Web Page widget
`title`	User entry	Label shown in the UI; applies only to Embed Web Page widget
`barchart`	`true` or `false`	Reconciliation bar chart; applies only to Last Reconciliation widget

When complete, you can select the name of the new dashboard under the Dashboards menu.

You can modify the options for each dashboard and widget. Select the vertical ellipsis in the upper right corner of the object, and make desired choices from the pop-up menu that appears.

4.2. Working With the Self-Service UI

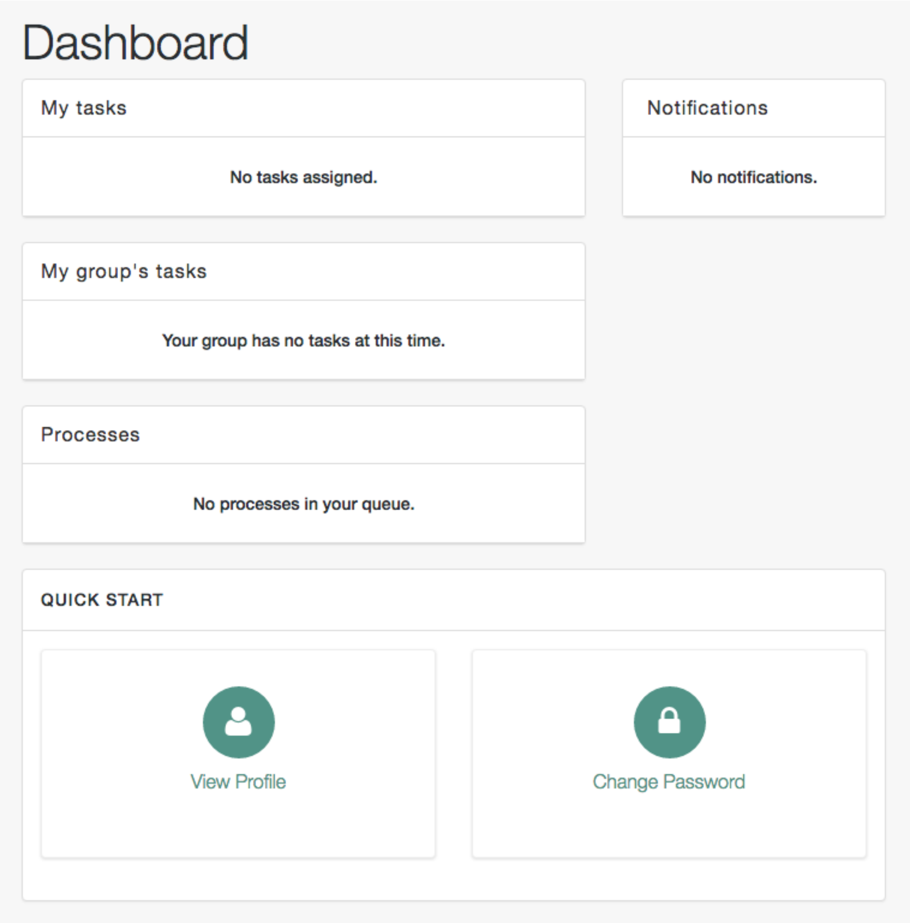
For all users, the Self-Service UI includes Dashboard and Profile links in the top menu bar.

To access the Self-Service UI, start OpenIDM, then navigate to `https://localhost:8443/`. If you have not installed a certificate that is trusted by a certificate authority, you are prompted with an Untrusted Connection warning the first time you log in to the UI.

4.2.1. The Self-Service UI Dashboard

The Dashboard includes a list tasks assigned to the user who has logged in, tasks assigned to the relevant group, processes available to be invoked, current notifications for that user, along with Quick Start cards for that user's profile and password.

The OpenIDM Self-Service UI Dashboard



Dashboard

My tasks

No tasks assigned.

Notifications

No notifications.

My group's tasks

Your group has no tasks at this time.

Processes

No processes in your queue.

QUICK START

For examples of these tasks, processes, and notifications, see "*Workflow Samples*" in the *Samples Guide*.

4.2.2. The Self-Service UI Profile

Every user who logs into the Self-Service UI has a profile, with Basic Info and Password Tabs. Users other than `openidm-admin` may see additional information, including Preferences, Social Identities, and Security Questions tabs.

You'll see the following information under each tab:

Basic Info

Specifies basic account information, including username, first name, last name, and email address.

Password

Supports password changes; for more information on password policy criteria, see "Enforcing Password Policy".

Preferences

Allows selection of preferences, as defined in the `managed.json` file, and the Managed Object User property Preferences tab. The default preferences relate to updates and special offers.

Social Identities

Lists social ID providers that have been enabled in the Admin UI. If you have registered with one provider, you can enable logins to this account with additional social ID providers. For more information on configuring and linking each provider, see "*Configuring Social ID Providers*".

Security Questions

Shown if KBA is enabled. Includes security questions and answers for this account, created when a new user goes through the registration process. For more information on KBA, see "Configuring Self-Service Questions".

4.3. Customizing a UI Template

You may want to customize information included in the Self-Service UI.

These procedures do not address actual data store requirements. If you add text boxes in the UI, it is your responsibility to set up associated properties in your repositories.

To do so, you should copy existing default template files in the `openidm/ui/selfservice/default` subdirectory to associated `extension/` subdirectories.

To simplify the process, you can copy some or all of the content from the `openidm/ui/selfservice/default/templates` to the `openidm/ui/selfservice/extension/templates` directory.

You can use a similar process to modify what is shown in the Admin UI.

4.3.1. Customizing User Self-Service Screens

In the following procedure, you will customize the screen that users see during the User Registration process. You can use a similar process to customize what a user sees during the Password Reset and Forgotten Username processes.

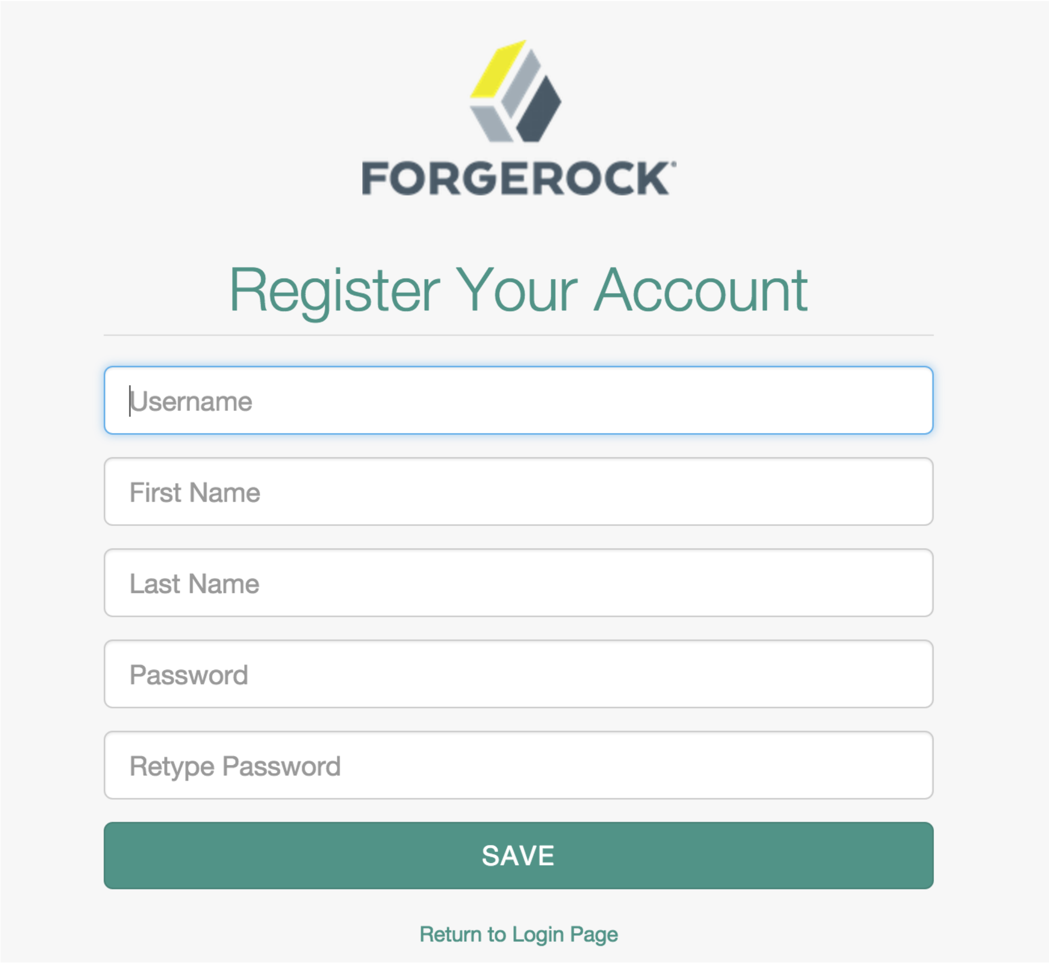
For user Self-Service features, you can customize options in three files. Navigate to the `extension/templates/user/process` subdirectory, and examine the following files:

- User Registration: `registration/userDetails-initial.html`
- Password Reset: `reset/userQuery-initial.html`
- Forgotten Username: `username/userQuery-initial.html`

The following procedure demonstrates the process for User Registration.

Customizing the User Registration Page

1. When you configure user self-service, as described in "*Configuring User Self-Service*", anonymous users who choose to register will see a screen similar to:



Register Your Account

Username

First Name

Last Name

Password

Retype Password

2. The screen you see is from the following file: `userDetails-initial.html`, in the `selfservice/extension/templates/user/process/registration` subdirectory. Open that file in a text editor.

3. Assume that you want new users to enter an employee ID number when they register.

   Create a new `form-group` stanza for that number. For this procedure, the stanza appears after the stanza for Last Name (or surname) `sn`:

   ```
   <div class="form-group">
       <label class="sr-only" for="input-employeeNum">{{t 'common.user.employeeNum'}}</label>
       <input type="text" placeholder="{{t 'common.user.employeeNum'}}" id="input-employeeNum"
    name="user.employeeNum" class="form-control input-lg" />
   </div>
   ```

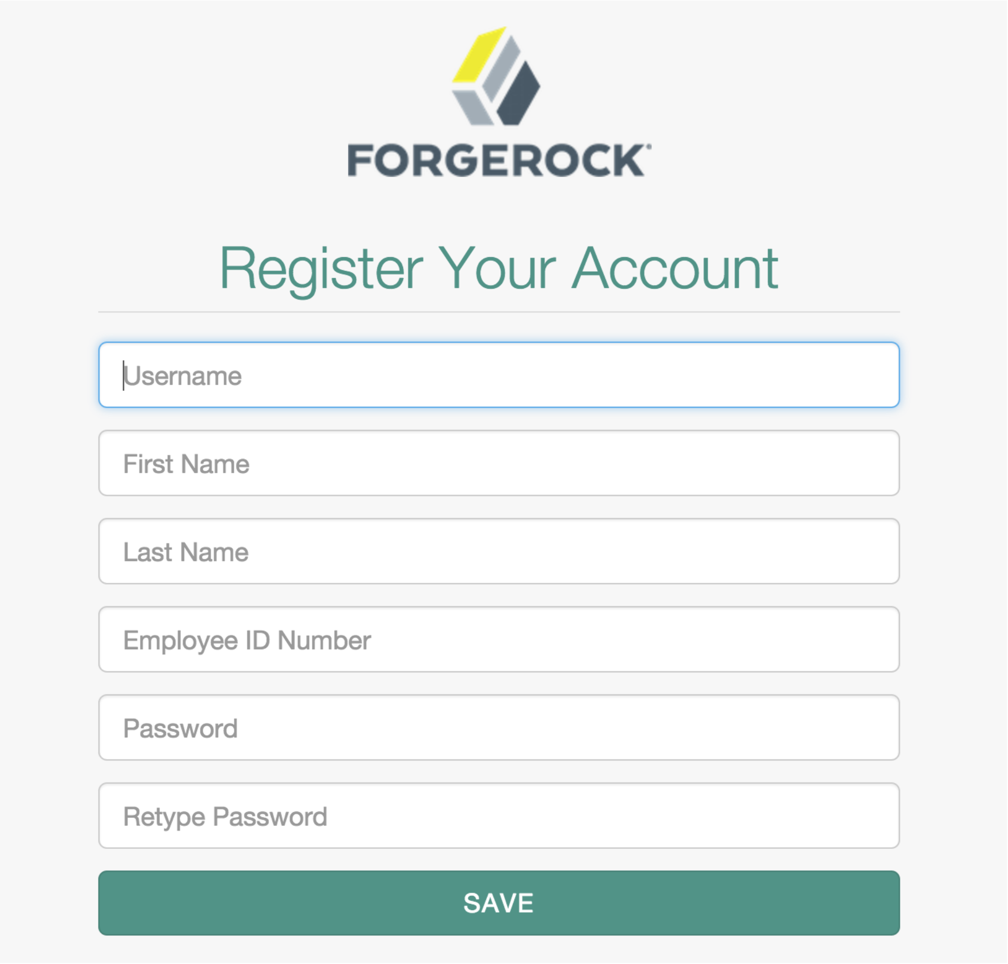
4. Edit the relevant `translation.json` file. As this is the customized file for the Self-Service UI, you will find it in the `selfservice/extension/locales/en` directory that you set up in "Customizing the UI".

   You need to find the right place to enter text associated with the `employeeNum` property. Look for the other properties in the `userDetails-initial.html` file.

   The following excerpt illustrates the `employeeNum` property as added to the `translation.json` file.

   ```
   ...
   "givenName" : "First Name",
   "sn" : "Last Name",
   "employeeNum" : "Employee ID Number",
   ...
   ```

5. The next time an anonymous user tries to create an account, that user should see a screen similar to:



FORGEROCK

Register Your Account

Username

First Name

Last Name

Employee ID Number

Password

Retype Password

In the following procedure, you will customize what users can modify when they navigate to their User Profile page:

Adding a Custom Tab to the User Profile Page

If you want to allow users to modify additional data on their profiles, this procedure is for you.

1. Log in to the Self-Service UI. Click the Profile tab. You should see at least the following tabs: `Basic Info` and `Password`. In this procedure, you will add a `Mobile Phone` tab.

2. OpenIDM generates the user profile page from the following file: `UserProfileTemplate.html`. Assuming you set up custom `extension` subdirectories, as described in "Customizing a UI

Template", you should find a copy of this file in the following directory: `selfservice/extension/templates/user`.

3. Examine the first few lines of that file. Note how the `tablist` includes the tabs in the Self-Service UI user profile: Basic Info and Password, associated with the `common.user.basicInfo` and `common.user.password` properties.

   The following excerpt includes a third tab, with the `mobilePhone` property:

```
<div class="container">
 <div class="page-header">
  <h1>{{t "common.user.userProfile"}}</h1>
 </div>
 <div class="tab-menu">
  <ul class="nav nav-tabs" role="tablist">
   <li class="active"><a href="#userDetailsTab" role="tab" data-toggle="tab">
     {{t "common.user.basicInfo"}}</a></li>
   <li><a href="#userPasswordTab" role="tab" data-toggle="tab">
     {{t "common.user.password"}}</a></li>
   <li><a href="#userMobilePhoneNumberTab" role="tab" data-toggle="tab">
     {{t "common.user.mobilePhone"}}</a></li>
  </ul>
 </div>
...
```

4. Next, you should provide information for the tab. Based on the comments in the file, and the entries in the `Password` tab, the following code sets up a Mobile Phone number entry:

```
<div role="tabpanel" class="tab-pane panel
     panel-default fr-panel-tab" id="userMobilePhoneNumberTab">
 <form class="form-horizontal" id="password">
  <div class="panel-body">
   <div class="form-group">
    <label class="col-sm-3 control-label" for="input-telephoneNumber">
     {{t "common.user.mobilePhone"}}</label>
    <div class="col-sm-6">
     <input class="form-control" type="telephoneNumber" id="input-mobilePhone"
     name="mobilePhone" value="" />
    </div>
   </div>
  </div>
  <div class="panel-footer clearfix">
   {{> form/_basicSaveReset}}
  </div>
 </form>
</div>
    ...
```

> **Note**
>
> For illustration, this procedure uses the HTML tags found in the `UserProfileTemplate.html` file. You can use any standard HTML content within `tab-pane` tags, as long as they include a standard `form` tag and standard `input` fields. OpenIDM picks up this information when the tab is saved, and uses it to `PATCH` user content.

5. Review the `managed.json` file. Make sure it is `viewable` and `userEditable` as shown in the following excerpt:

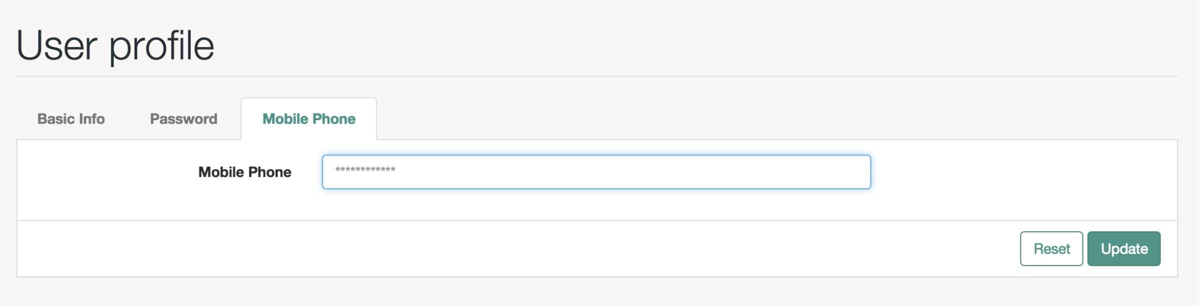
```
"telephoneNumber" : {
     "type" : "string",
     "title" : "Mobile Phone",
     "viewable" : true,
     "userEditable" : true,
     "pattern" : "^\\+?([0-9\\- \\(\\)])*$"
},
```

6. Open the applicable `translation.json` file. You should find a copy of this file in the following subdirectory: `selfservice/extension/locales/en/`.

   Search for the line with `basicInfo`, and add an entry for `mobilePhone`:

```
"basicInfo": "Basic Info",
"mobilePhone": "Mobile Phone",
```

7. Review the result. Log in to the Self-Service UI, and click Profile. Note the entry for the Mobile Phone tab.



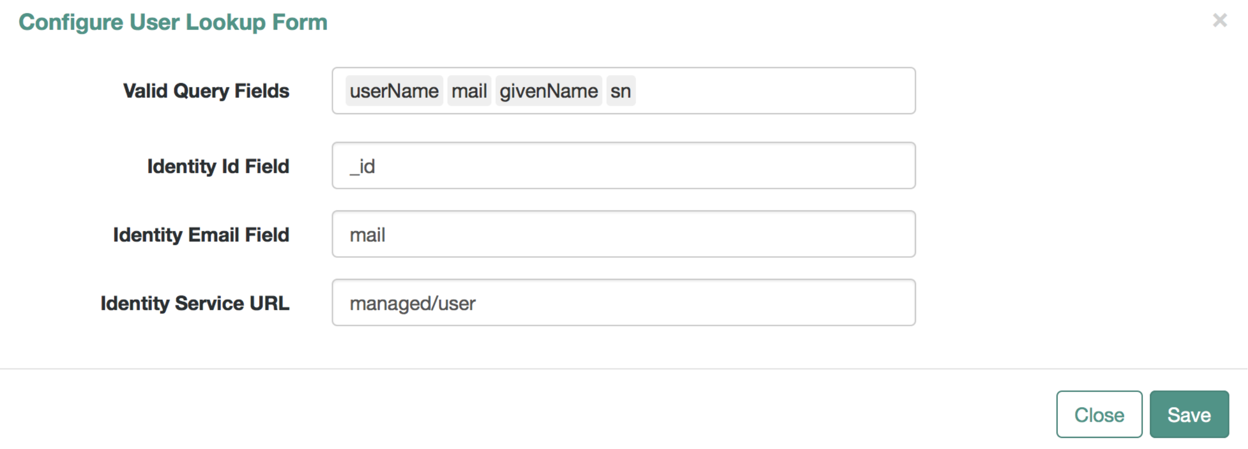
User profile

Mobile Phone: ************

4.3.2. Modifying Valid Query Fields

For Password Reset and Forgotten Username functionality, you may choose to modify Valid Query Fields, such as those described in "*Configuring User Self-Service*".

For example, if you click Configure > Password Reset > User Query Form, you can make changes to *Valid Query Fields*.



Configure User Lookup Form

Valid Query Fields: userName mail givenName sn

Identity Id Field: _id

Identity Email Field: mail

Identity Service URL: managed/user

If you add, delete, or modify any Valid Query Fields, you will have to change the corresponding `userQuery-initial.html` file.

Assuming you set up custom `extension` subdirectories, as described in "Customizing a UI Template", you can find this file in the following directory: `selfservice/extension/templates/user/process`.

If you change any Valid Query Fields, you should make corresponding changes.

- For Forgotten Username functionality, you would modify the `username/userQuery-initial.html` file.
- For Password Reset functionality, you would modify the `reset/userQuery-initial.html` file.

For a model of how you can change the `userQuery-initial.html` file, see "Customizing the User Registration Page".

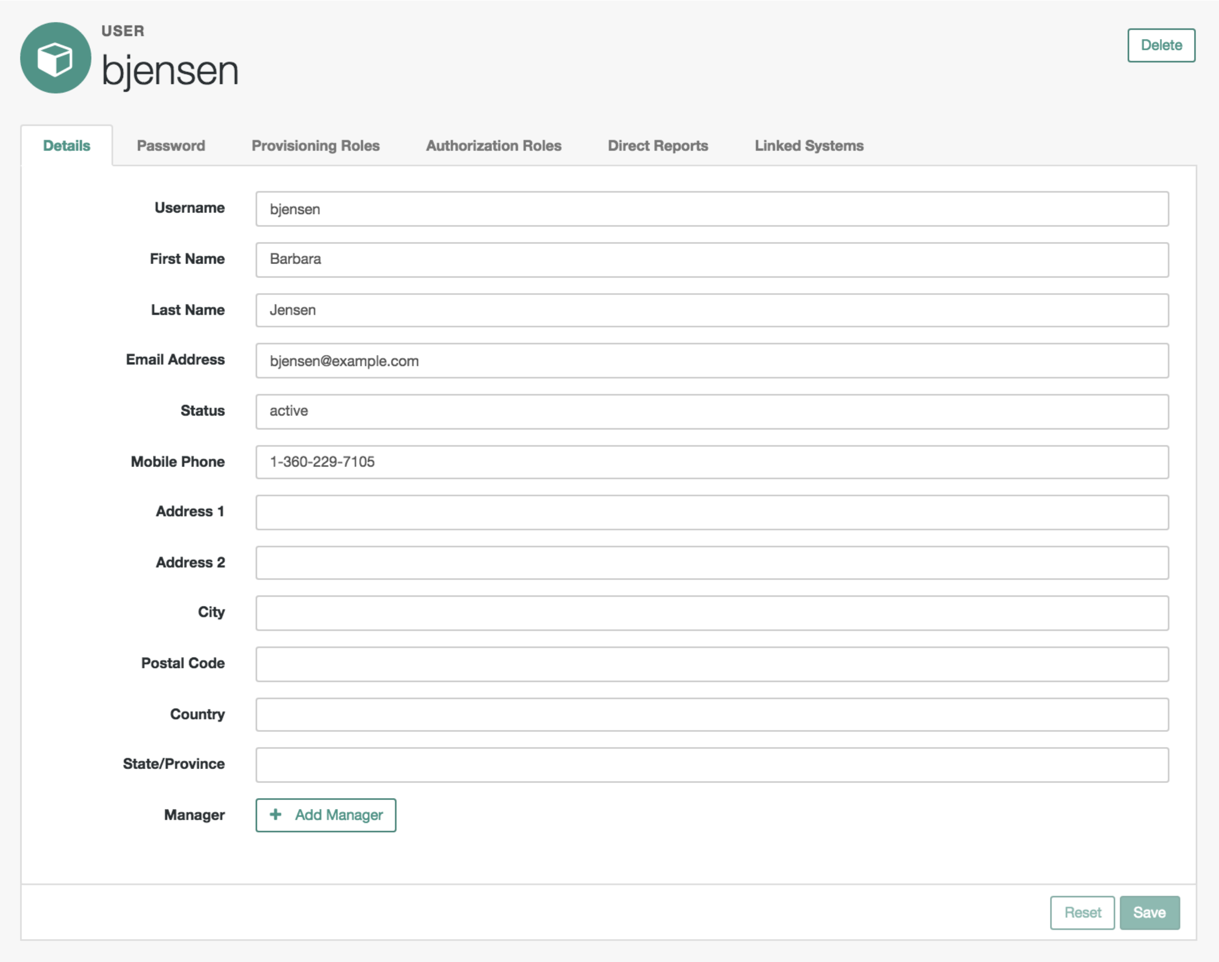
4.4. Managing Accounts

Only administrative users (with the role `openidm-admin`) can add, modify, and delete accounts from the Admin UI. Regular users can modify certain aspects of their own accounts from the Self-Service UI.

4.4.1. Account Configuration

In the Admin UI, you can manage most details associated with an account, as shown in the following screenshot.

Account, UI Configuration



Username	bjensen
First Name	Barbara
Last Name	Jensen
Email Address	bjensen@example.com
Status	active
Mobile Phone	1-360-229-7105
Address 1	
Address 2	
City	
Postal Code	
Country	
State/Province	
Manager	+ Add Manager

You can configure different functionality for an account under each tab:

Details

The Details tab includes basic identifying data for each user, with two special entries:

Status

By default, accounts are shown as *active*. To suspend an account, such as for a user who has taken a leave of absence, set that user's status to *inactive*.

Manager

You can assign a manager from the existing list of managed users.

Password

As an administrator, you can create new passwords for users in the managed user repository.

Provisioning Roles

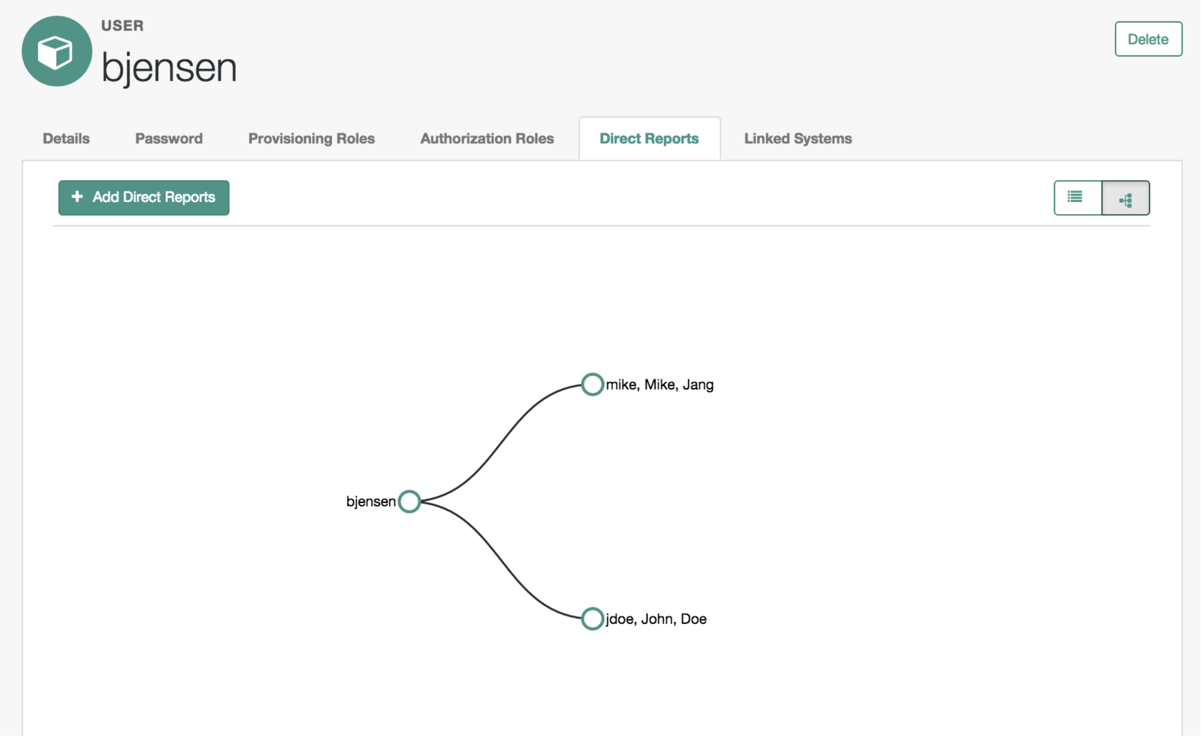
Used to specify how objects are provisioned to an external system. For more information, see "Working With Managed Roles".

Authorization Roles

Used to specify the authorization rights of a managed user within OpenIDM. For more information, see "Working With Managed Roles".

Direct Reports

Users who are listed as managers of others have entries under the Direct Reports tab, as shown in the following illustration:



Linked Systems

Used to display account information reconciled from external systems.

4.4.2. Procedures for Managing Accounts

With the following procedures, you can add, update, and deactivate accounts for managed objects such as users.

The managed object does not have to be a user. It can be a role, a group, or even be a physical item such as an IoT device. The basic process for adding, modifying, deactivating, and deleting other objects is the same as it is with accounts. However, the details may vary; for example, many IoT devices do not have telephone numbers.

To Add a User Account

1. Log in to the Admin UI at `https://localhost:8443/admin`.
2. Click Manage > User.
3. Click New User.
4. Complete the fields on the New User page.

   Most of these fields are self-explanatory. Be aware that the user interface is subject to policy validation, as described in "*Using Policies to Validate Data*". So, for example, the email address must be a valid email address, and the password must comply with the password validation settings that appear if you enter an invalid password.

In a similar way, you can create accounts for other managed objects.

You can review new managed object settings in the `managed.json` file of your `project-dir/conf` directory.

In the following procedures, you learn how:

- "To Update a User Account"
- "To Delete a User Account"
- "To View an Account in External Resources"

To Update a User Account

1. Log in to the Admin UI at `https://localhost:8443/admin` as an administrative user.
2. Click Manage > User.

3. Click the Username of the user that you want to update.
4. On the profile page for the user, modify the fields you want to change and click Update.

   The user account is updated in the OpenIDM repository.

To Delete a User Account

1. Log in to the Admin UI at `https://localhost:8443/admin` as an administrative user.
2. Click Manage > User.
3. Select the checkbox next to the desired Username.
4. Click the Delete Selected button.
5. Click OK to confirm the deletion.

   The user is deleted from the internal repository.

To View an Account in External Resources

The Admin UI displays the details of the account in the OpenIDM repository (managed/user). When a mapping has been configured between the repository and one or more external resources, you can view details of that account in any external system to which it is linked. As this view is read-only, you cannot update a user record in a linked system from within the Self-Service UI.

By default, *implicit synchronization* is enabled for mappings *from* the `managed/user` repository *to* any external resource. This means that when you update a managed object, any mappings defined in the `sync.json` file that have the managed object as the source are automatically executed to update the target system. You can see these changes in the Linked Systems section of a user's profile.

To view a user's linked accounts:

1. Log in to the Admin UI at `https://localhost:8443/admin`.
2. Click Manage User > *Username* > Linked Systems.
3. The Linked Systems panel indicates the external mapped resource or resources.
4. Select the resource in which you want to view the account, from the Linked Resource list.

   The user record in the linked resource is displayed.

4.5. Configuring Account Relationships

This section will help you set up relationships between human users and devices, such as IoT devices.

You'll set this up with the help of the Admin UI schema editor, which allows you to create and customize managed objects such as `Users` and `Devices` as well as relationships between managed objects. You can also create these options in the `managed.json` file for your project.

When complete, you will have users who can own multiple unique devices. If you try to assign the same device to more than one owner, OpenIDM will stop you with an error message.

This section assumes that you have started OpenIDM with "Sample 2b - LDAP Two Way" in the *Samples Guide*.

After you have started OpenIDM with "Sample 2b", go through the following procedures, where you will:

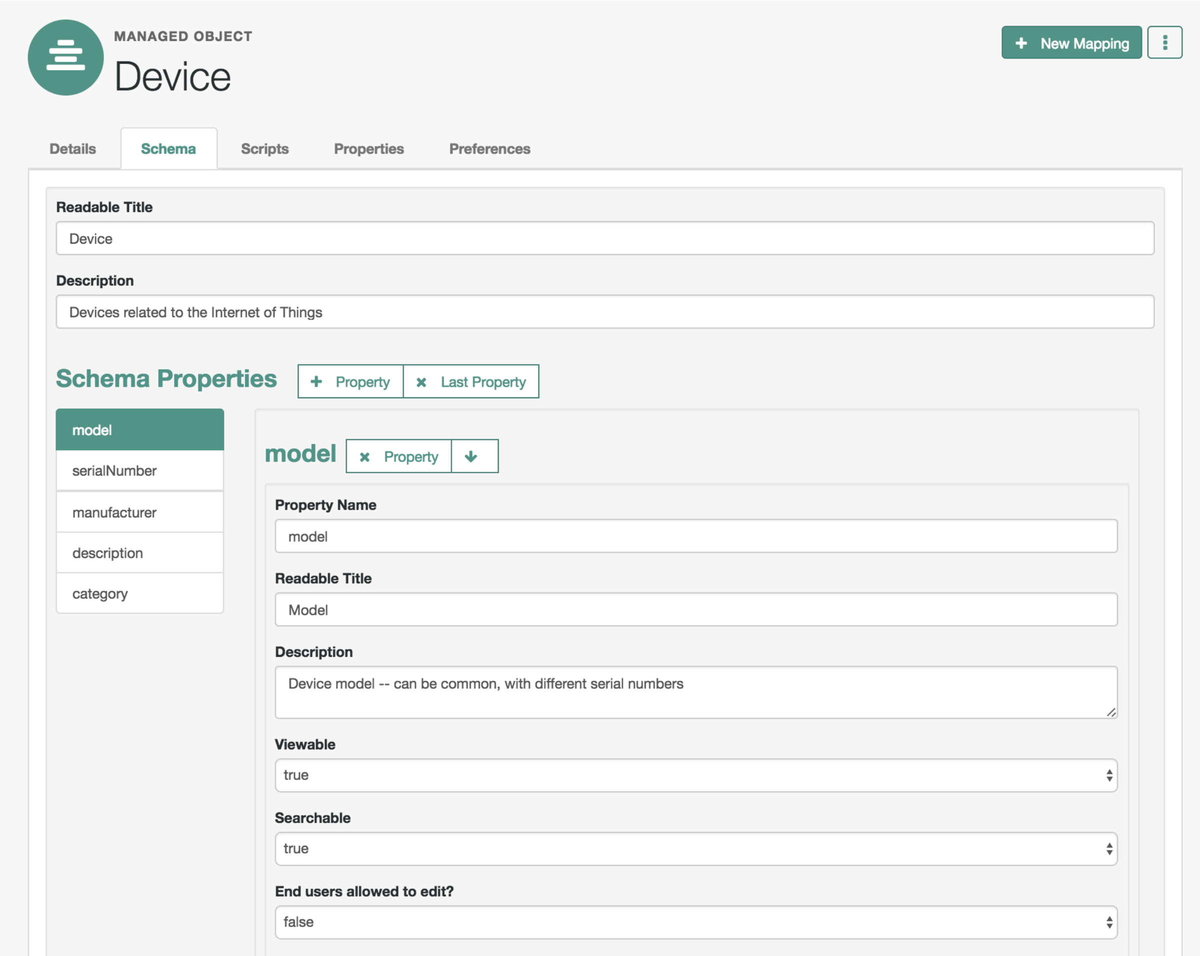
- Set up a managed object named `Device`, with unique serial numbers for each device. You can configure the searchable schema of your choice. See "Configuring Schema for a Device" for details.
- Set up a relationship from the Device to the User managed object. See "Configure a Relationship from the Device Managed Object" for details.
- Set up a Two-way from the User to the Device managed object. See "Configure a Relationship From the User Managed Object" for details.
- Demonstrate the relationships. Assign users to devices. See what happens when you try to assign a device to more than one user. For details, see "Demonstrating an IoT Relationship".

Configuring Schema for a Device

This procedure illustrates how you might set up a Device managed object, with schema that configures relationships to users.

After you configure the schema for the Device managed object, you can collect information such as model, manufacturer, and serial number for each device. In the next procedure, you'll set up an `owner` schema property that includes a relationship to the User managed object.

1. Click Configure > Managed Objects > New Managed Object. Give that object an appropriate IoT name. For this procedure, specify `Device`. You should also select a managed object icon. Click Save.
2. You should now see five tabs: Details, Schema, Scripts, Properties, and Preferences. Select the Schema tab.



Readable Title: Device

Description: Devices related to the Internet of Things

Property Name: model

Readable Title: Model

Description: Device model -- can be common, with different serial numbers

Viewable: true

Searchable: true

End users allowed to edit?: false

3. The items that you can add to the new managed object depend on the associated properties.

   The Schema tab includes the `Readable Title` of the device; in this case, set it to `Device`.

4. You can add schema properties as needed in the UI. Click the Property button. Include the properties shown in the illustration: model, serialNumber, manufacturer, description, and category.

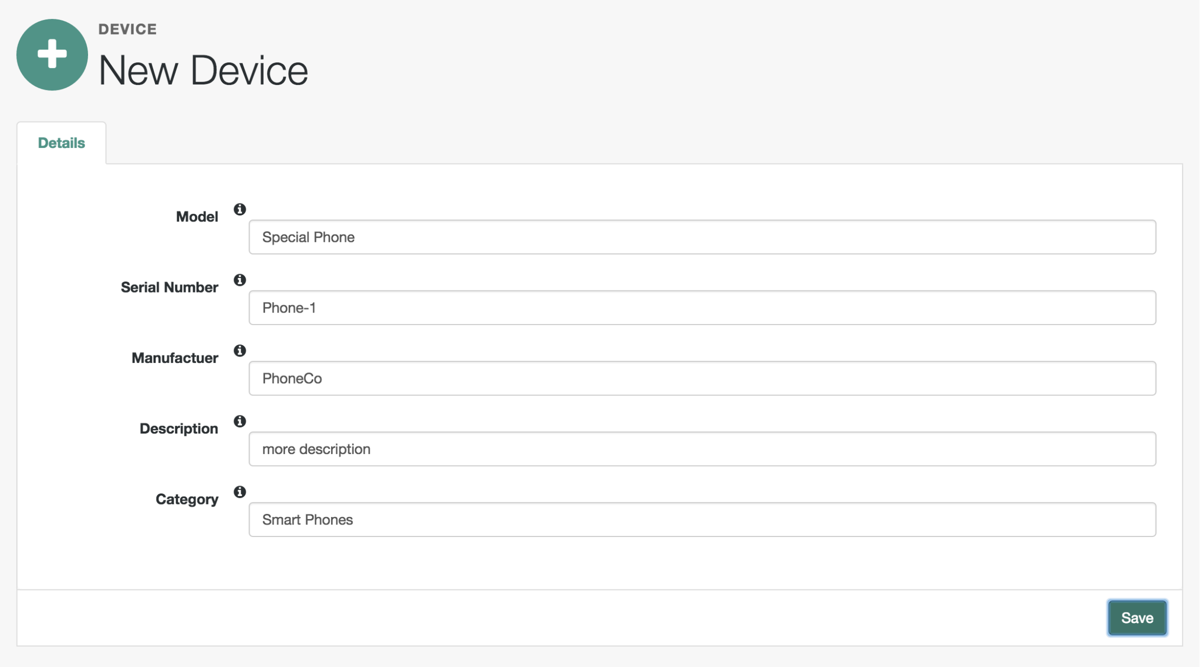
5. Initially, the new property is named `Property 1`. As soon as you enter a property name such as `model`, OpenIDM changes that property name accordingly.

6. To support UI-based searches of devices, make sure to set the Searchable option to true for all configured schema properties, unless it includes extensive text, In this case, you should set Searchable to false for the `description` property.

The Searchable option is used in the data grid for the given object. When you click Manage > Device (or another object such as User), OpenIDM displays searchable properties for that object.

7. After you save the properties for the new managed object type, OpenIDM saves those entries in the `managed.json` file in the *`project-dir`*`/conf` directory.

8. Now click Manage > Device > New Device. Add a device as shown in the following illustration.



DEVICE

New Device

Details

Model: Special Phone

Serial Number: Phone-1

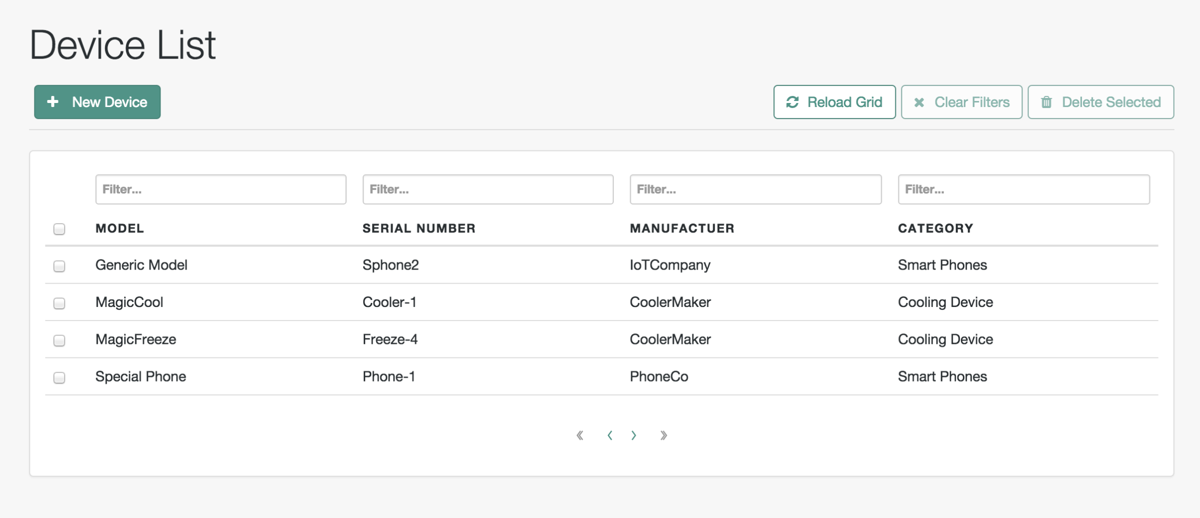
Manufactuer: PhoneCo

Description: more description

Category: Smart Phones

9. You can continue adding new devices to the managed object, or reconcile that managed object with another data store. The other procedures in this section assume that you have set up the devices as shown in the next illustration.

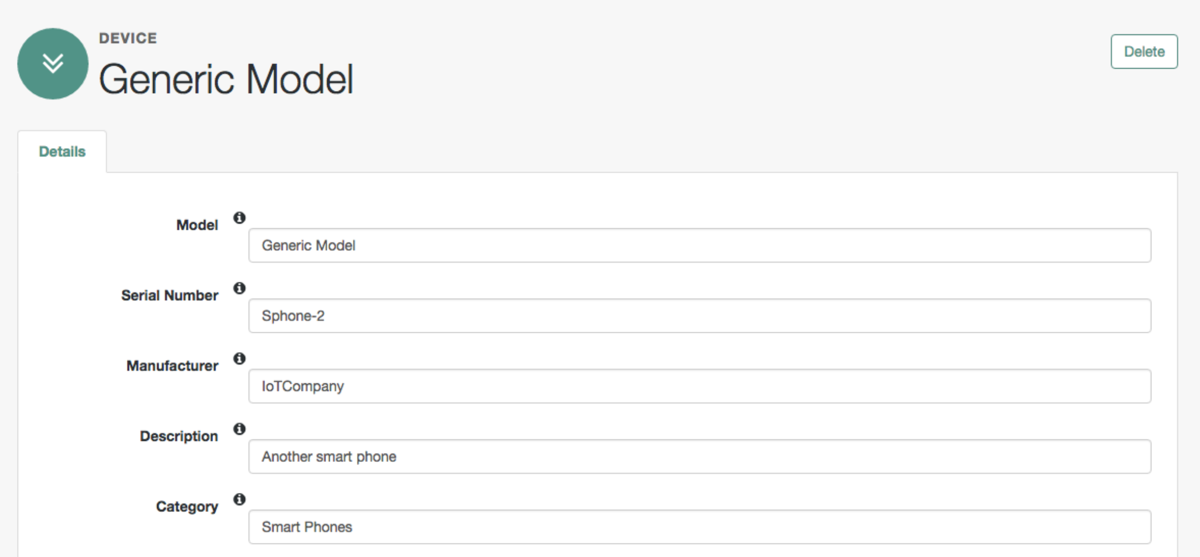
10. When complete, you can review the list of devices. Based on this procedure, click Manage > Device.



Device List

□	MODEL	SERIAL NUMBER	MANUFACTUER	CATEGORY
□	Generic Model	Sphone2	IoTCompany	Smart Phones
□	MagicCool	Cooler-1	CoolerMaker	Cooling Device
□	MagicFreeze	Freeze-4	CoolerMaker	Cooling Device
□	Special Phone	Phone-1	PhoneCo	Smart Phones

11. Select one of the listed devices. You'll note that the label for the device in the Admin UI matches the name of the first property of the device.



DEVICE
Generic Model

Model: Generic Model

Serial Number: Sphone-2

Manufacturer: IoTCompany

Description: Another smart phone

Category: Smart Phones

You can change the order of schema properties for the Device managed object by clicking Configure > Managed Objects > Device > Schema, and select the property that you want to move up or down the list.

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

Configure a Relationship from the Device Managed Object

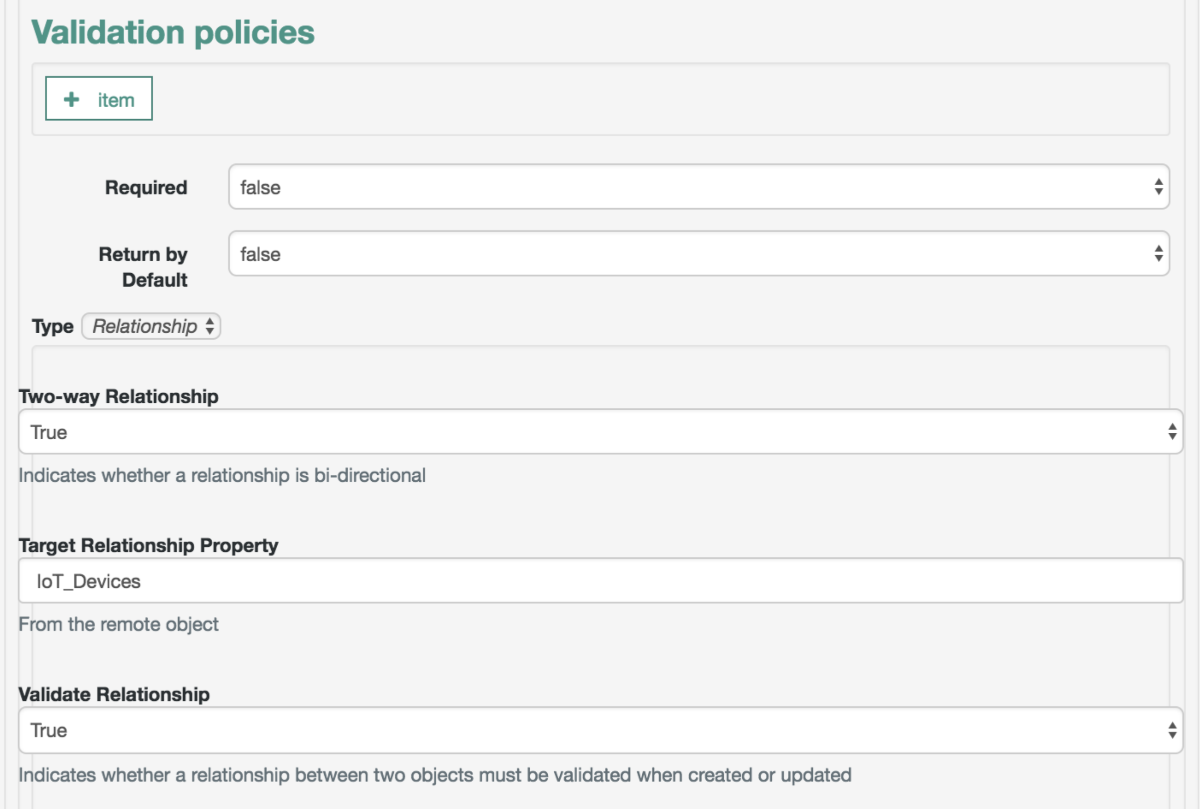
In this procedure, you will add a property to the schema of the Device managed object.

1. In the Admin UI, click Configure > Managed Objects > Device > Schema.

2. Under the Schema tab, add a new property. For this procedure, we call it *owner*. Unlike other schema properties, set the Searchable property to false.

3. Scroll down to Validation Policies; click the Type box and select Relationship. This opens additional relationship options.

4. Set up a Target Property Name of `IoT_Devices`. You'll use that property name in the next "Configure a Relationship From the User Managed Object".



Validation policies

Required: false

Return by Default: false

Type: Relationship

Two-way Relationship: True

Indicates whether a relationship is bi-directional

Target Relationship Property: IoT_Devices

From the remote object

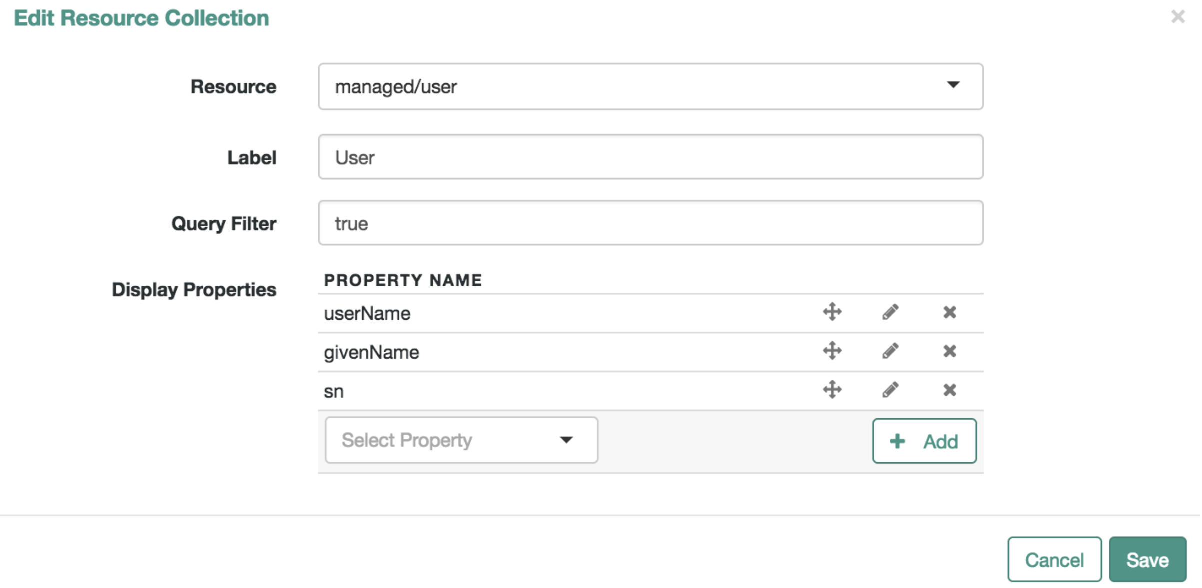
Validate Relationship: True

Indicates whether a relationship between two objects must be validated when created or updated

Be sure to set the Two-way Relationship and Validate Relationship options to `true`, which ensures that each device is associated with no more than one user.

5. Scroll down and add a Resource Collection. Set up a link to the `managed/user` resource, with a label that matches the `User` managed object.

6. Enable queries of the User managed object by setting Query Filter to true. The Query Filter value for this Device object allows you to identify the user who "owns" each device. For more information, see "Common Filter Expressions".



Edit Resource Collection

Resource: managed/user

Label: User

Query Filter: true

Display Properties

PROPERTY NAME
userName
givenName
sn

7. Set up Display Properties from `managed/user` properties. The properties shown in the illustration are just examples, based on "Sample 2b - LDAP Two Way" in the *Samples Guide*.

8. Press Save to exit the Resource Collection pop-up. Press Save again in the Manage Device window.

Configure a Relationship From the User Managed Object

In this procedure, you will configure an existing User Managed Object with schema to match what was created in "Configure a Relationship from the Device Managed Object".

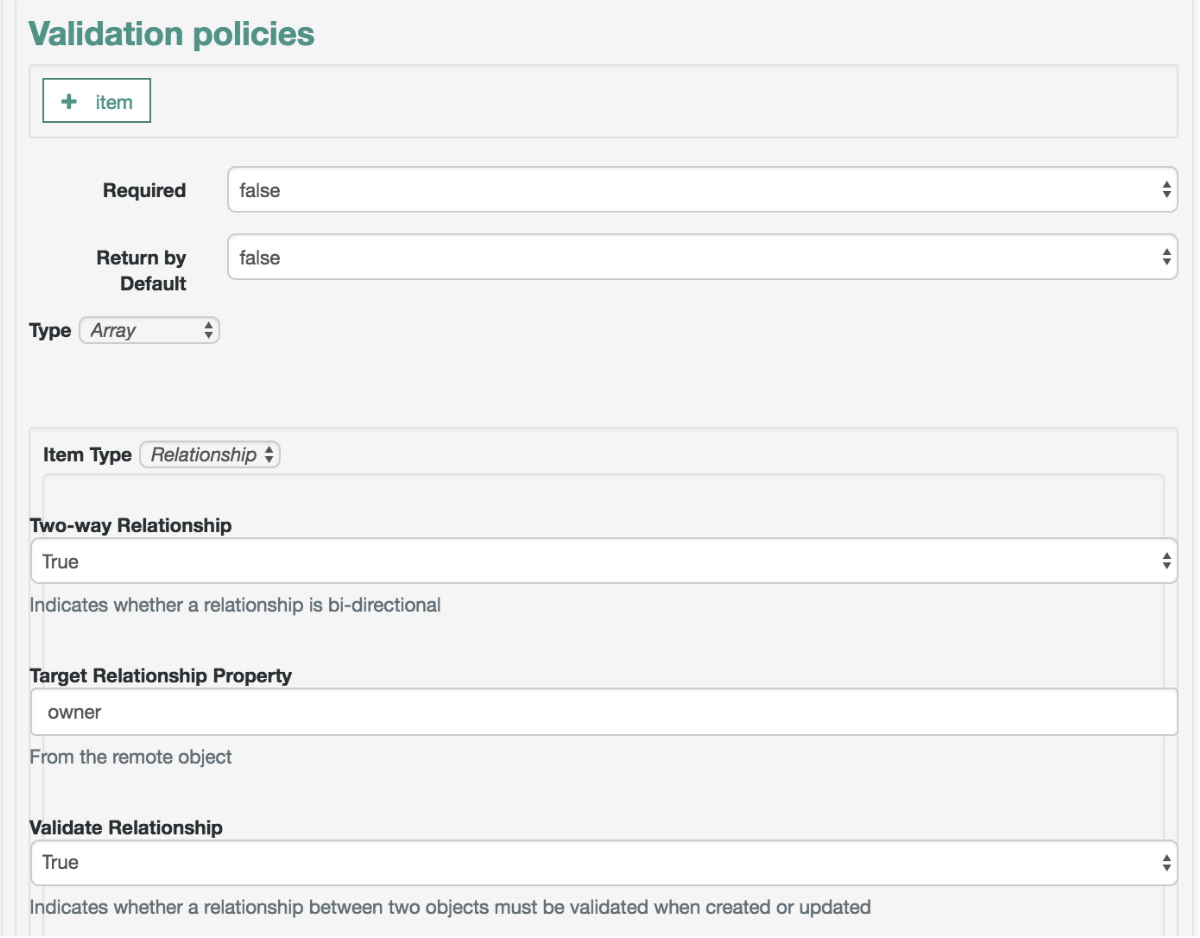
With the settings you create, OpenIDM supports a relationship between a single user and multiple devices. In addition, this procedure prevents multiple users from "owning" any single device.

1. In the Admin UI, click Configure > Managed Objects > User > Schema.

2. Under the Schema tab, add a new property, called IoT_Devices.

3. Make sure the searchable property is set to false, to minimize confusion in the relationship. Otherwise, you'll see every device owned by every user, when you select Manage > User.

4. For validation policies, you'll set up an *array* with a relationship. Note how the reverse property name matches the property that you configured in "Configure a Relationship from the Device Managed Object".



Validation policies

Required: false

Return by Default: false

Type: Array

Item Type: Relationship

Two-way Relationship: True

Indicates whether a relationship is bi-directional

Target Relationship Property: owner

From the remote object

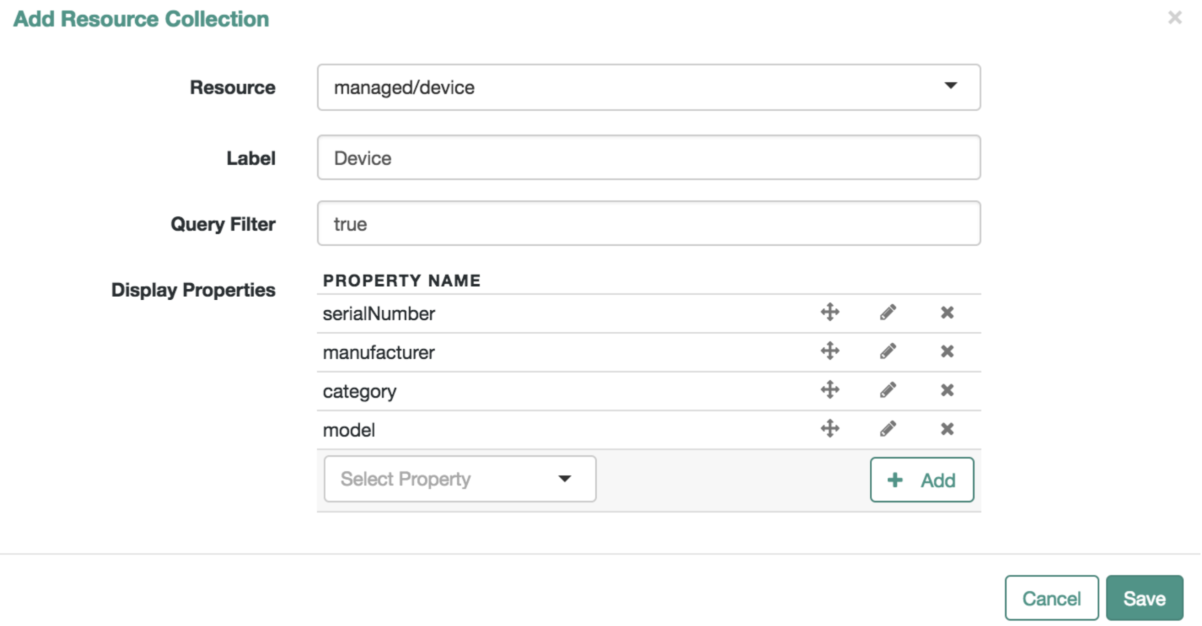
Validate Relationship: True

Indicates whether a relationship between two objects must be validated when created or updated

Be sure to set the Two-way Relationship and Validate Relationship options to `true`, which ensures that no more than one user gets associated with a specific device.

5. Scroll down to Resource Collection, and add references to the `managed/device` resource, as shown in the next illustration.

6. Enter `true` in the Query Filter text box. In this relationship, OpenIDM will read all information from the `managed/device` managed object, with information from the device fields and sort keys that you configured in "Configure a Relationship from the Device Managed Object".



Add Resource Collection

Resource: managed/device

Label: Device

Query Filter: true

Display Properties:

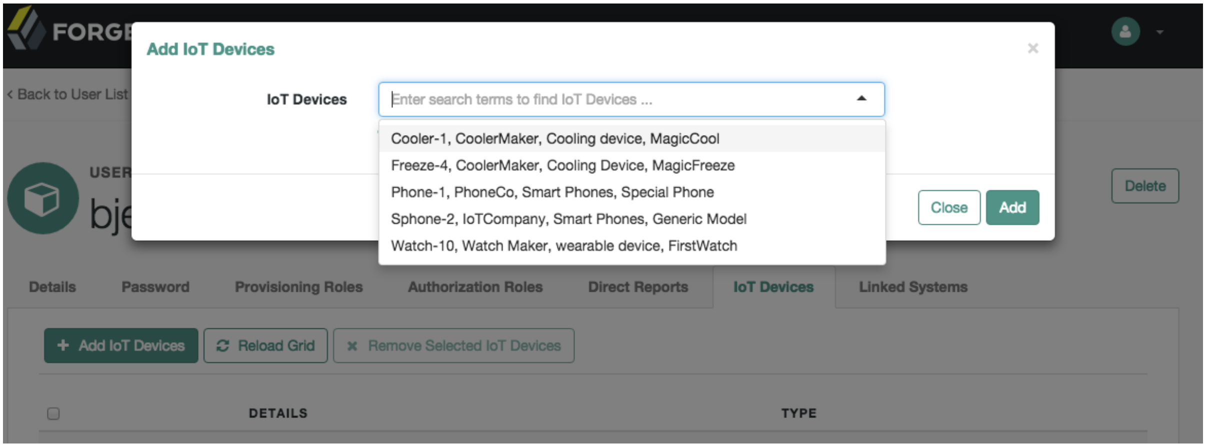
PROPERTY NAME
serialNumber
manufacturer
category
model

Demonstrating an IoT Relationship

This procedure assumes that you have already taken the steps described in the previous procedures in this section, specifically, "Configuring Schema for a Device", "Configure a Relationship from the Device Managed Object", and "Configure a Relationship From the User Managed Object".

This procedure also assumes that you started OpenIDM with "Sample 2b - LDAP Two Way" in the *Samples Guide*, and have reconciled to set up users.

1. From the Admin UI, click Manage > User. Select a user, and in this case, click the IoT Devices tab. See how you can select any of the devices that you may have added in "Configuring Schema for a Device".



Add IoT Devices

IoT Devices: Enter search terms to find IoT Devices ...

- Cooler-1, CoolerMaker, Cooling device, MagicCool
- Freeze-4, CoolerMaker, Cooling Device, MagicFreeze
- Phone-1, PhoneCo, Smart Phones, Special Phone
- Sphone-2, IoTCompany, Smart Phones, Generic Model
- Watch-10, Watch Maker, wearable device, FirstWatch

2. Alternatively, try to assign a device to an owner. To do so, click Manage > Device, and select a device. You'll see either an `Add Owner` or `Update Owner` button, which allows you to assign a device to a specific user.

   If you try to assign a device already assigned by a different user, you'll get the following message: `Conflict with Existing Relationship`.

4.6. Customizing the UI

OpenIDM allows you to customize both the Admin and Self-Service UIs. When you install OpenIDM, you can find the default UI configuration files in two directories:

- Admin UI: `openidm/ui/admin/default`
- Self-Service UI: `openidm/ui/selfservice/default`

OpenIDM looks for custom themes and templates in the following directories:

- Admin UI: `openidm/ui/admin/extension`
- Self-Service UI: `openidm/ui/selfservice/extension`

Before starting the customization process, you should create these directories. If you are running UNIX/Linux, the following commands create a copy of the appropriate subdirectories:

```
$ cd /path/to/openidm/ui
$ cp -r selfservice/default/. selfservice/extension
$ cp -r admin/default/. admin/extension
```

OpenIDM also includes templates that may help, in two other directories:

- Admin UI: `openidm/ui/admin/default/templates`

- Self-Service UI: `openidm/ui/selfservice/default/templates`

If you want to customize workflows in the UI, see "Managing User Access to Workflows".

4.7. Changing the UI Theme

You can customize the theme of the user interface. OpenIDM uses the *Bootstrap* framework. You can download and customize the OpenIDM UI with the Bootstrap themes of your choice. OpenIDM is also configured with the *Font Awesome* CSS toolkit.

> **Note**
>
> If you use *Brand Icons from the Font Awesome CSS Toolkit*, be aware of the following statement:
>
> All brand icons are trademarks of their respective owners. The use of these trademarks does not indicate endorsement of the trademark holder by ForgeRock, nor vice versa.

4.7.1. OpenIDM UI Themes and Bootstrap

You can configure a few features of the OpenIDM UI in the `ui-themeconfig.json` file in your project's `conf/` subdirectory. However, to change most theme-related features of the UI, you must copy target files to the appropriate `extension` subdirectory, and then modify them as discussed in "Customizing the UI".

The default configuration files for the Admin and Self-Service UIs are identical for theme configuration.

By default the UI reads the stylesheets and images from the respective `openidm/ui/`*`function`*`/default` directories. Do not modify the files in this directory. Your changes may be overwritten the next time you update or even patch your system.

To customize your UI, first set up matching subdirectories for your system (`openidm/ui/admin/extension` and `openidm/ui/selfservice/extension`). For example, assume you want to customize colors, logos, and so on.

You can set up a new theme, primarily through custom Bootstrap CSS files, in appropriate `extension/` subdirectories, such as `openidm/ui/selfservice/extension/libs` and `openidm/ui/selfservice/extension/css`.

You may also need to update the `"stylesheets"` listing in the `ui-themeconfig.json` file for your project, in the *`project-dir`*`/conf` directory.

```
"stylesheets" : [
    "css/bootstrap-3.4.1-custom.css",
    "css/structure.css",
    "css/theme.css"
],
```

You can find these `stylesheets` in the `/css` subdirectory.

- `bootstrap-3.4.1-custom.css`: Includes custom settings that you can get from various Bootstrap configuration sites, such as the Bootstrap *Customize and Download* website.

  You may find the ForgeRock version of this in the `config.json` file in the `ui/selfservice/default/css/common/structure/` directory.

- `structure.css`: Supports configuration of structural elements of the UI.

- `theme.css`: Includes customizable options for UI themes such as colors, buttons, and navigation bars.

If you want to set up custom versions of these files, copy them to the `extension/css` subdirectories.

4.7.2. Changing the Default Logo

For the Self-Service UI, you can find the default logo in the `openidm/ui/selfservice/default/images` directory. To change the default logo, copy desired files to the `openidm/ui/selfservice/extension/images` directory. You should see the changes after refreshing your browser.

To specify a different file name, or to control the size, and other properties of the image file that is used for the logo, adjust the `logo` property in the UI theme configuration file for your project: *`project-dir`*`/conf/ui-themeconfig.json`).

The following change to the UI theme configuration file points to an image file named `example-logo.png`, in the `openidm/ui/extension/images` directory:

```
...
"loginLogo" : {
     "src" : "images/example-logo.png",
     "title" : "Example.com",
     "alt" : "Example.com",
     "height" : "104px",
     "width" : "210px"
},
...
```

Refresh your browser window for the new logo to appear.

4.7.3. Changing the Language of the UI

Currently, the UI is provided only in US English. You can translate the UI and specify that your own locale is used. The following example shows how to translate the UI into French:

1. Assuming you set up custom `extension` subdirectories, as described in "Customizing the UI", you can copy the default (`en`) locale to a new (`fr`) subdirectory as follows:

   ```
   $ cd /path/to/openidm/ui/selfservice/extension/locales
   $ cp -R en fr
   ```

The new locale (`fr`) now contains the default `translation.json` file:

```
$ ls fr/
translation.json
```

2. Translate the values of the properties in the `fr/translate.json` file. Do *not* translate the property names. For example:

```
...
"UserMessages" : {
   "changedPassword" : "Mot de passe a été modifié",
   "profileUpdateFailed" : "Problème lors de la mise à jour du profil",
   "profileUpdateSuccessful" : "Profil a été mis à jour",
   "userNameUpdated" : "Nom d'utilisateur a été modifié",
....
```

3. Change the UI configuration to use the new locale by setting the value of the `lang` property in the *`project-dir`*`/conf/ui-configuration.json` file, as follows:

```
"lang" : "fr",
```

4. Refresh your browser window, and OpenIDM applies your change.

You can also change the labels for accounts in the UI. To do so, navigate to the `Schema Properties` for the managed object to be changed.

To change the labels for user accounts, navigate to the Admin UI. Click Configure > Managed Objects > User, and scroll down to Schema.

Under Schema Properties, select a property and modify the `Readable Title`. For example, you can modify the `Readable Title` for `userName` to a label in another language, such as `Nom d'utilisateur`.

4.7.4. Creating a Project-Specific UI Theme

You can create specific UI themes for different projects and then point a particular UI instance to use a defined theme on startup. To create a complete custom theme, follow these steps:

1. Shut down the OpenIDM instance, if it is running. In the OSGi console, type:

```
shutdown
->
```

2. Copy the entire default Self-Service UI theme to an accessible location. For example:

```
$ cd /path/to/openidm/ui/selfservice
$ cp -r default /path/to/openidm/new-project-theme
```

3. If desired, repeat the process with the Admin UI; just remember to copy files to a different directory:

```
$ cd /path/to/openidm/ui/admin
$ cp -r default /path/to/openidm/admin-project-theme
```

4. In the copied theme, modify the required elements, as described in the previous sections. Note that nothing is copied to the extension folder in this case - changes are made in the copied theme.

5. In the `conf/ui.context-selfservice.json` file, modify the values for `defaultDir` and `extensionDir` to the directory with your `new-project-theme`:

```
{
    "enabled" : true,
    "urlContextRoot" : "/",
    "defaultDir" : "&{launcher.install.location}/ui/selfservice/default",
    "extensionDir" : "&{launcher.install.location}/ui/selfservice/extension",
    "responseHeaders" : {
        "X-Frame-Options" : "DENY"
    }
}
```

6. If you want to repeat the process for the Admin UI, make parallel changes to the *`project-dir`*`/conf/ui.context-admin.json` file.

7. Restart OpenIDM.

```
$ cd /path/to/openidm
$ ./startup.sh
```

8. Relaunch the UI in your browser. The UI is displayed with the new custom theme.

4.8. Resetting User Passwords

When working with end users, administrators frequently have to reset their passwords. OpenIDM allows you to do so directly, through the Admin UI. Alternatively, you can configure an external system for that purpose.

4.8.1. Resetting a User Password Through the Admin UI

From the Admin UI, you can reset the passwords of accounts in the internal Managed User datastore. If you haven't already done so, start by configuring the outbound email service, as described in "*Configuring Outbound Email*". Then take the following steps in the Admin UI:

1. Select Manage > User. Choose a specific user from the list that appears.

2. Select the Password tab for that user; you should see a Reset Password option.

When you select Reset Password, OpenIDM by default generates a random 16 character password with at least one of each of the following types of characters:

- Uppercase letters: `A-Z`

- Lowercase letters: `a-z`

- Integers: `0-9`
- Special characters: `: ; < = > ? @`

OpenIDM then uses its configured outgoing email service to send that password to the specified user. For example, user `mike` might an email message with the following subject line:

```
Your password has been reset by an administrator
```

along with the following message:

```
mike's new password is: <generated_password>
```

If desired, you can configure that message (along with password complexity) by modifying the following code block in your project's `managed.json` file:

```
"actions" : {
    "resetPassword": {
        "type": "text/javascript",
        "source": "require('ui/resetPassword').sendMail(object, subject, message, passwordRules,
 passwordLength);",
        "globals": {
            "subject": "Your password has been reset by an administrator",
            "message": "<html><body><p>{{object.userName}}'s new password is: {{password}}</p></body></
html>",
            "passwordRules": [
                { "rule": "UPPERCASE", "minimum": 1 },
                { "rule": "LOWERCASE", "minimum": 1 },
                { "rule": "INTEGERS", "minimum": 1 },
                { "rule": "SPECIAL", "minimum": 1 }
            ],
        "passwordLength": 16
    }
}
```

4.8.2. Using an External System for Password Reset

By default, the Password Reset mechanism is handled internally, in OpenIDM. You can reroute Password Reset in the event that a user has forgotten their password, by specifying an external URL to which Password Reset requests are sent. Note that this URL applies to the Password Reset link on the login page only, not to the security data change facility that is available after a user has logged in.

To set an external URL to handle Password Reset, set the `passwordResetLink` parameter in the UI configuration file (`conf/ui-configuration.json`) file. The following example sets the `passwordResetLink` to `https://accounts.example.com/account/reset-password`:

```
passwordResetLink: "https://accounts.example.com/reset-password"
```

The `passwordResetLink` parameter takes either an empty string as a value (which indicates that no external link is used) or a full URL to the external system that handles Password Reset requests.

Note

External Password Reset and security questions for internal Password Reset are mutually exclusive. Therefore, if you set a value for the `passwordResetLink` parameter, users will not be prompted with any security questions, regardless of the setting of the `securityQuestions` parameter.

4.9. Providing a Logout URL to External Applications

By default, a UI session is invalidated when a user clicks on the Log out link. In certain situations your external applications might require a distinct logout URL to which users can be routed, to terminate their UI session.

The logout URL is `#logout`, appended to the UI URL, for example, `https://localhost:8443/#logout/`.

The logout URL effectively performs the same action as clicking on the Log out link of the UI.

4.10. Changing the UI Path

By default, the Self-Service UI is registered at the root context and is accessible at the URL `https://localhost:8443`. To specify a different URL, edit the `project-dir/conf/ui.context-selfservice.json` file, setting the `urlContextRoot` property to the new URL.

For example, to change the URL of the Self-Service UI to `https://localhost:8443/exampleui`, edit the file as follows:

```
"urlContextRoot" : "/exampleui",
```

Alternatively, to change the Self-Service UI URL in the Admin UI, follow these steps:

1. Log in to the Admin UI.
2. Select Configure > System Preferences, and select the Self-Service UI tab.
3. Specify the new context route in the Relative URL field.

4.11. API Explorer

OpenIDM includes an API Explorer, an implementation of the *OpenAPI Initiative Specification*, also known as Swagger.

To access the API Explorer, log into the Admin UI, select the question mark in the upper right corner, and choose API Explorer from the drop-down menu.

Note

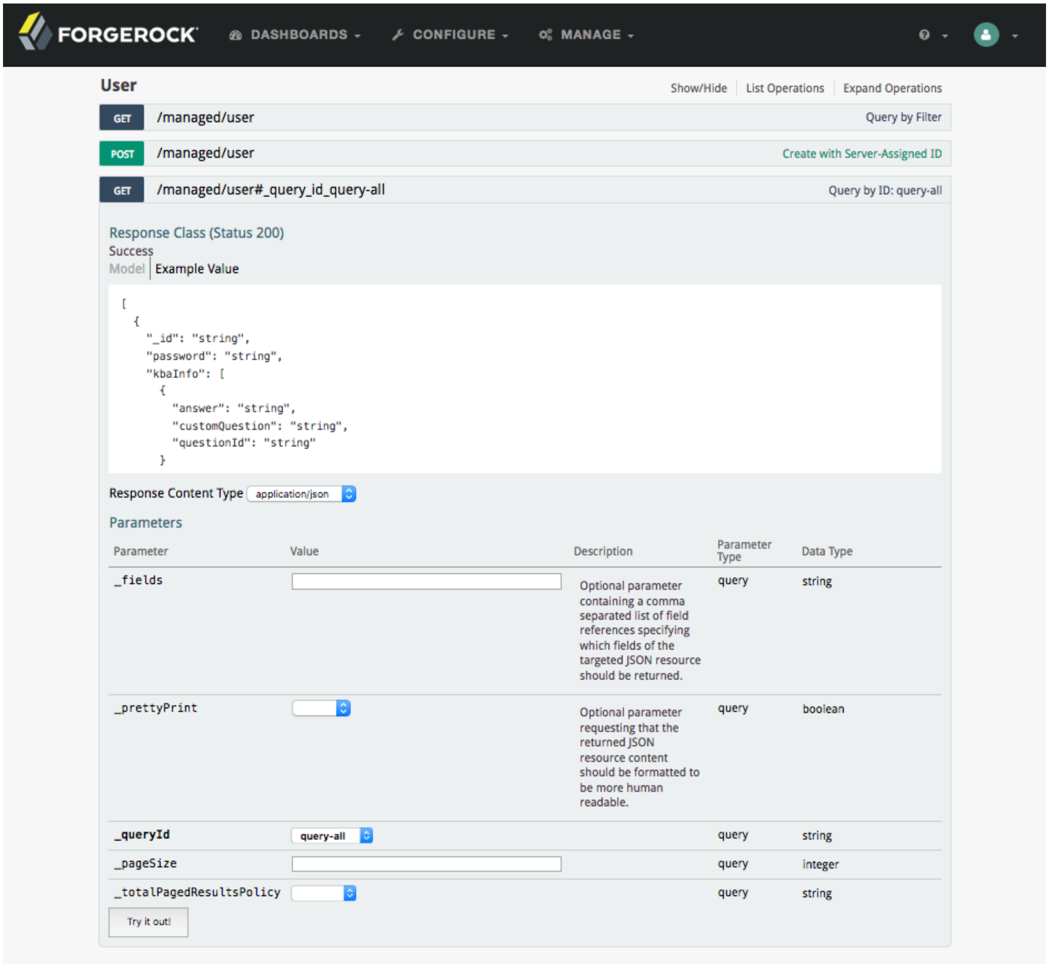
If the API Explorer does not appear, you may need to enable it in your project's `conf/boot/boot.properties` file, specifically with the `openidm.apidescriptor.enabled` property. For more information see, "Disable the API Explorer".

In the API Explorer, you'll find several navigable endpoints, including:

- `/managed/assignment`
- `/managed/role`
- `/managed/user`

Each endpoint lists supported HTTP methods such as POST and GET. When custom actions are available, the API Explorer lists them as *HTTP Method* `/path/to/endpoint?_action=something`.

To see how this works, navigate to the `User` endpoint, select List Operations, and choose the GET option associated with the `/managed/user#_query_id_query-all` endpoint.



GET	/managed/user	Query by Filter
POST	/managed/user	Create with Server-Assigned ID
GET	/managed/user#_query_id_query-all	Query by ID: query-all

Response Class (Status 200)
Success
Model | Example Value

```
[
  {
    "_id": "string",
    "password": "string",
    "kbaInfo": [
      {
        "answer": "string",
        "customQuestion": "string",
        "questionId": "string"
      }
```

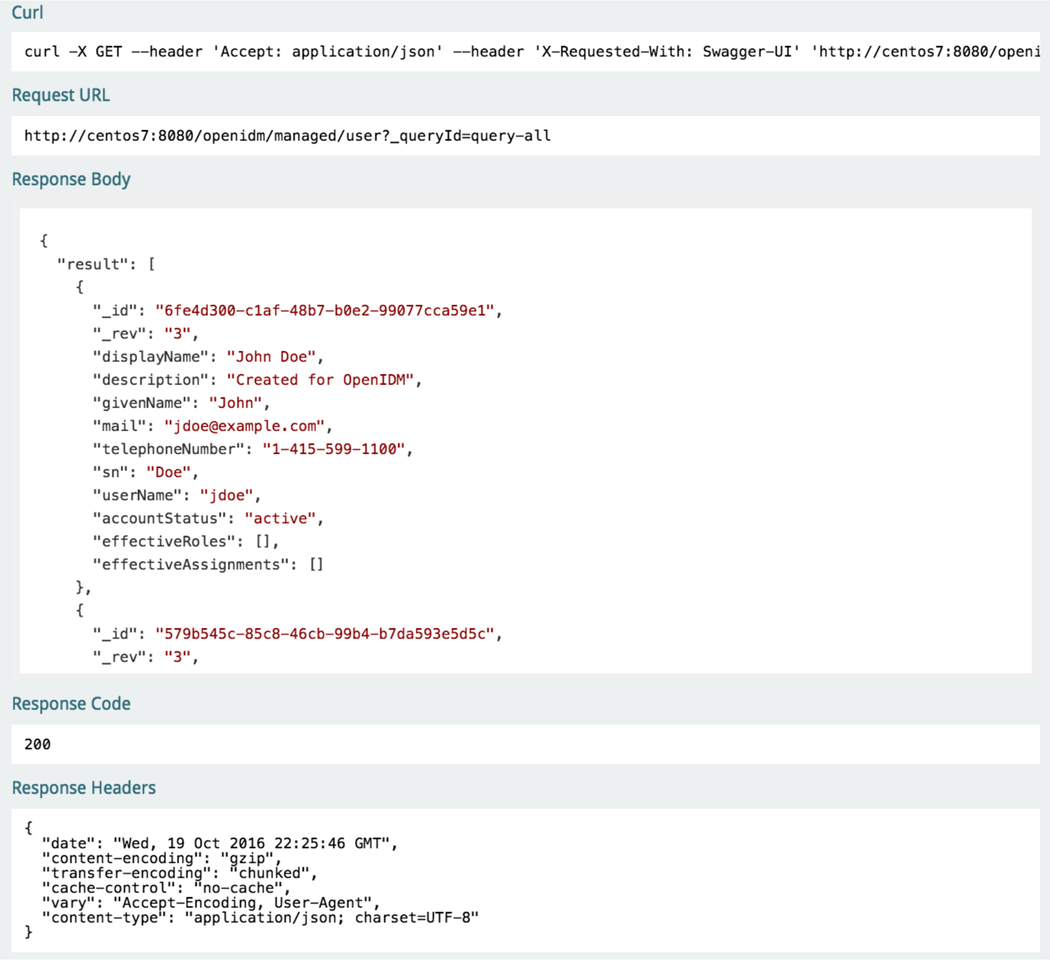
Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
_fields		Optional parameter containing a comma separated list of field references specifying which fields of the targeted JSON resource should be returned.	query	string
_prettyPrint		Optional parameter requesting that the returned JSON resource content should be formatted to be more human readable.	query	boolean
_queryId	query-all		query	string
_pageSize			query	integer
_totalPagedResultsPolicy			query	string

In this case, the defaults are set, and all you need to do is select the `Try it out!` button. The output you see includes:

- The REST call, in the form of the **curl** command.
- The request URL, which specifies the endpoint and associated parameters.
- The response body, which contains the data that you requested.
- The HTTP response code; if everything works, this should be `200`.
- Response headers.



Curl

```
curl -X GET --header 'Accept: application/json' --header 'X-Requested-With: Swagger-UI' 'http://centos7:8080/openi
```

Request URL

```
http://centos7:8080/openidm/managed/user?_queryId=query-all
```

Response Body

```
{
  "result": [
    {
      "_id": "6fe4d300-c1af-48b7-b0e2-99077cca59e1",
      "_rev": "3",
      "displayName": "John Doe",
      "description": "Created for OpenIDM",
      "givenName": "John",
      "mail": "jdoe@example.com",
      "telephoneNumber": "1-415-599-1100",
      "sn": "Doe",
      "userName": "jdoe",
      "accountStatus": "active",
      "effectiveRoles": [],
      "effectiveAssignments": []
    },
    {
      "_id": "579b545c-85c8-46cb-99b4-b7da593e5d5c",
      "_rev": "3",
```

Response Code

```
200
```

Response Headers

```
{
  "date": "Wed, 19 Oct 2016 22:25:46 GMT",
  "content-encoding": "gzip",
  "transfer-encoding": "chunked",
  "cache-control": "no-cache",
  "vary": "Accept-Encoding, User-Agent",
  "content-type": "application/json; charset=UTF-8"
}
```

If you're familiar with "Sample 2b - LDAP Two Way" in the *Samples Guide*, you may recognize the output as users in the OpenIDM managed user object, after reconciliation.

Tip

If you see a `401 Access Denied` code in the response body, your OpenIDM session may have timed out, and you'll have to log into the Admin UI again.

For details on common ForgeRock REST parameters, see "About ForgeRock Common REST".

You'll see examples of REST calls throughout ForgeRock OpenIDM documentation. You can now try these calls with the API Explorer.

You can generate an OpenAPI-compliant descriptor of the REST API to provide API reference documentation specific to your deployment. The following command saves the API descriptor of the managed/user endpoint to a file named `my-openidm-api.json`:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 --output "my-openidm-api.json" \
 "http://localhost:8080/openidm/managed/user?_api"
```

For information about publishing reference documentation using the API descriptor, see "To Publish OpenAPI Documentation".

4.12. Disabling the UI

The UI is packaged as a separate bundle that can be disabled in the configuration before server startup. To disable the registration of the UI servlet, edit the `project-dir/conf/ui.context-selfservice.json` file, setting the `enabled` property to false:

```
"enabled" : false,
```

Chapter 5

Configuring User Self-Service

The following sections describe how you can configure three functions of user self-service: User Registration, Forgotten Username, and Password Reset.

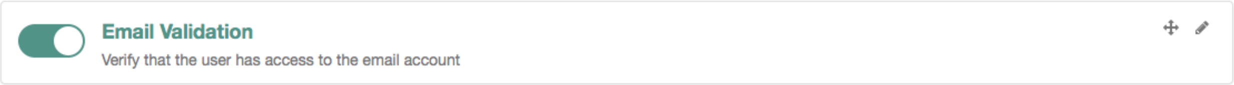
- User Registration: You can configure limited access that allows a current anonymous user to create their own accounts. To aid in this process, you can configure reCAPTCHA, email validation, and KBA questions.

  If you have configured one or more social identity providers, as described in "*Configuring Social ID Providers*", you can enable the use of those providers for User Registration. You can also configure the user terms and conditions of your choice, typically a license and/or a privacy agreement.

- Forgotten Username: You can set up OpenIDM to allow users to recover forgotten usernames via their email addresses or first and last names. OpenIDM can then display that username on the screen, and/or email such information to that user.

- Password Reset: You can set up OpenIDM to verify user identities via KBA questions. If email configuration is included, OpenIDM would email a link that allows users to reset their passwords.

If you enable email functionality, the one solution that works for all three self-service functions is to configure an outgoing email service for OpenIDM, as described in "*Configuring Outbound Email*".



Email Validation

Verify that the user has access to the email account

Note

If you disable email validation only for user registration, you should perform one of the following actions:

- Disable validation for `mail` in the managed user schema. Click Configure > Managed Objects > User > Schema. Under Schema Properties, click Mail, scroll down to Validation Policies, and set Required to `false`.
- Configure the User Registration template to support user email entries. To do so, use "Customizing the User Registration Page", and substitute `mail` for `employeeNum`.

Without these changes, users who try to register accounts will see a `Forbidden Request Error`.

You can configure user self-service through the UI and through configuration files.

- In the UI, log into the Admin UI. You can enable these features when you click Configure > User Registration, Configure > Forgotten Username, and Configure > Password Reset.

- In the command-line interface, copy the following files from `samples/misc` to your working *`project-dir/conf`* directory:

  User Registration: `selfservice-registration.json`
  Forgotten username: `selfservice-username.json`
  Password reset: `selfservice-reset.json`

  Examine the `ui-configuration.json` file in the same directory. You can activate or deactivate User Registration and Password Reset by changing the value associated with the `selfRegistration` and `passwordReset` properties:

  ```
  {
      "configuration" : {
      "selfRegistration" : true,
      "passwordReset" : true,
      "forgotUsername" : true,
      ...
  ```

For each of these functions, you can configure several options, including:

reCAPTCHA

> Google reCAPTCHA helps prevent bots from registering users or resetting passwords on your system. For Google documentation, see *Google reCAPTCHA*. For directions on how to configure reCAPTCHA for user self-service, see "Configuring Google reCAPTCHA".

Email Validation / Email Username

> You can configure the email messages that OpenIDM sends to users, as a way to verify identities for user self-service. For more information, see "Configuring Self-Service Email Messages".
>
> If you configure email validation, you must also configure an outgoing email service in OpenIDM. To do so, click Configure > System Preferences > Email. For more information, read "*Configuring Outbound Email*".

User Details

> You can modify the Identity Email Field associated with user registration; by default, it is set to `mail`.

User Query

> When configuring password reset and forgotten username functionality, you can modify the fields that a user is allowed to query. If you do, you may need to modify the HTML templates that appear to users who request such functionality. For more information, see "Modifying Valid Query Fields".
>
> **Valid Query Fields**
>
> > Property names that you can use to help users find their usernames or verify their identity, such as `userName`, `mail`, or `givenName`.

Identity ID Field

Property name associated with the User ID, typically `_id`.

Identity Email Field

Property name associated with the user email field, typically something like `mail` or `email`.

Identity Service URL

The path associated with the identity data store, such as `managed/user`.

KBA Stage

You can modify the list of Knowledge-based Authentication (KBA) questions in the `conf/selfservice.kba.json` file. Users can then select the questions they will use to help them verify their own identities. For directions on how to configure KBA questions, see "Configuring Self-Service Questions". For User Registration, you cannot configure these questions in the Admin UI.

Password Reset Form

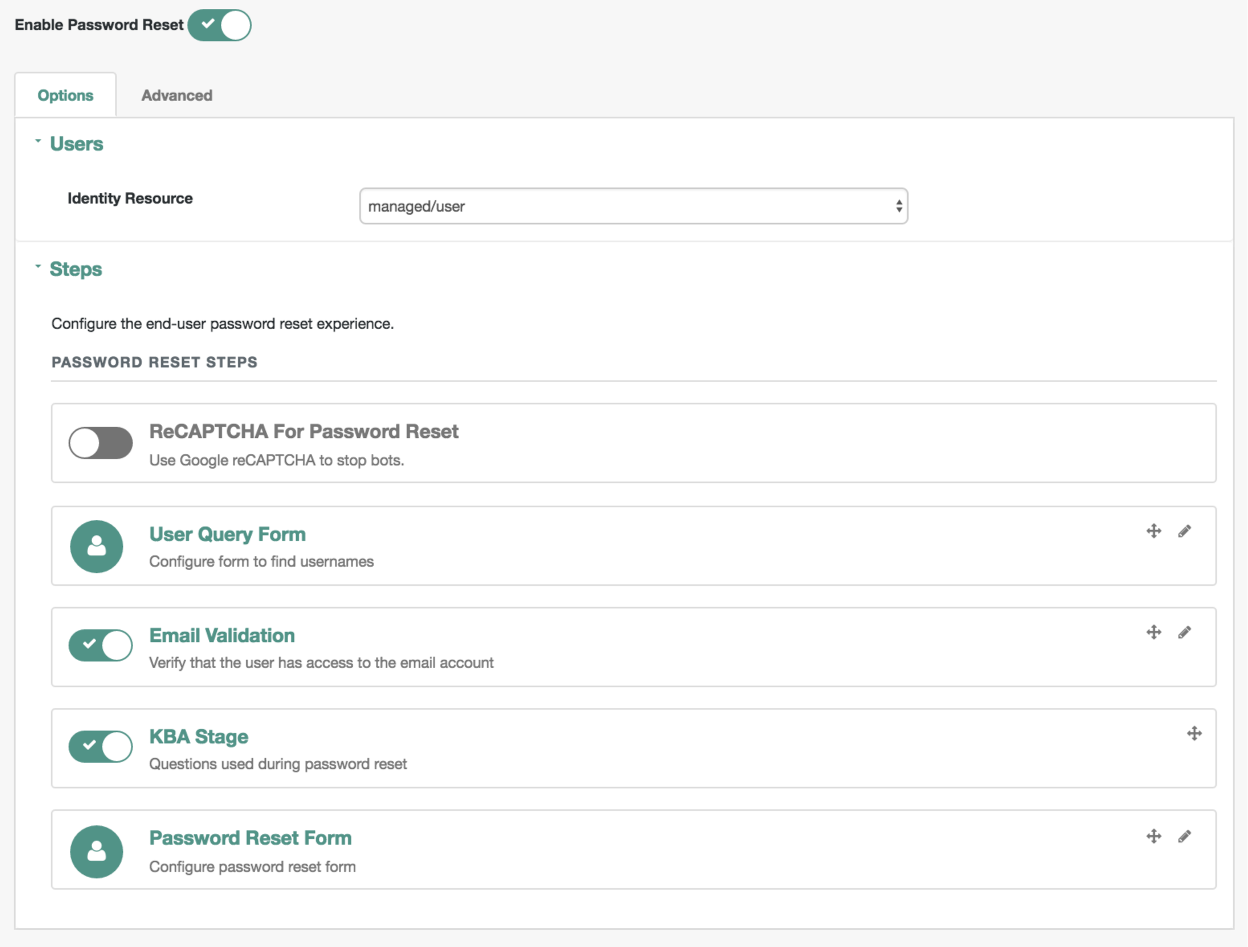
You can change the Password Field for the Password Reset feature to specify a relevant password property such as `password`, `pwd`, or `userPassword`. Make sure the property you select matches the canonical form for user passwords.

Snapshot Token

OpenIDM User Self-Service uses JWT tokens, with a default token lifetime of 1800 seconds.

You can reorder how OpenIDM works with relevant self-service options, specifically reCAPTCHA, KBA stage questions, and email validation. Based on the following screen, users who need to reset their passwords will go through reCAPTCHA, followed by email validation, and then answer any configured KBA questions.

OpenIDM Self-Service UI - Password Reset Sequence



Enable Password Reset ✓

Users

Identity Resource: managed/user

Steps

Configure the end-user password reset experience.

PASSWORD RESET STEPS

- ReCAPTCHA For Password Reset (off): Use Google reCAPTCHA to stop bots.
- User Query Form: Configure form to find usernames
- Email Validation ✓: Verify that the user has access to the email account
- KBA Stage ✓: Questions used during password reset
- Password Reset Form: Configure password reset form

To reorder the steps, either "drag and drop" the options in the Admin UI, or change the sequence in the associated configuration file, in the `project-dir/conf` directory.

OpenIDM generates a token for each process. For example, users who forget their usernames and passwords go through two steps:

- The user goes through the User Registration process, gets a JWT token, and has the token lifetime (default = 1800 seconds) to get to the next step in the process.
- With username in hand, that user may then start the Password Reset process. That user gets a second JWT token, with the token lifetime configured for that process.

5.1. Common Configuration Details

This section describes configuration details common to OpenIDM Self-Service features: User Registration, Password Reset, and Forgotten Username.

5.1.1. Configuring Self-Service Email Messages

When a user requests a new account, a Password Reset, or a reminder of their username, you can configure OpenIDM to send that user an email message, to confirm the request.

You can configure that email message either through the UI or the associated configuration files, as illustrated in the following excerpt of the `selfservice-registration.json` file.

```
{
  "stageConfigs" : {
    {
      "name" : "emailValidation",
      "identityEmailField" : "mail",
      "emailServiceUrl" : "external/email",
      "from" : "admin@example.net",
      "subject" : "Register new account",
      "mimeType" : "text/html",
      "subjectTranslations" : {
        "en" : "Register new account",
        "fr" : "Créer un nouveau compte"
      },
      "messageTranslations" : {
        "en" : "<h3>This is your registration email.</h3><h4><a href=\"%link%\">Email verification link</
a></h4>",
        "fr" : "<h3>Ceci est votre mail d'inscription.</h3><h4><a href=\"%link%\">Lien de vérification
 email</a></h4>",
      "verificationLinkToken" : "%link%",
      "verificationLink" : "https://localhost:8443/#register/"
    }
...
```

Note the two languages in the `subjectTranslations` and `messageTranslations` code blocks. You can add translations for languages other than US English `en` and French `fr`. Use the appropriate two-letter code based on ISO 639. End users will see the message in the language configured in their web browsers.

You can set up similar emails for password reset and forgotten username functionality, in the `selfservice-reset.json` and `selfservice-username.json` files. For templates, see the `/path/to/openidm/samples/misc` directory.

One difference between User Registration and Password Reset is in the `verificationLink`; for Password Reset, the corresponding URL is:

```
...
"verificationLink" : "https://localhost:8443/#passwordReset/"
...
```

Substitute the IP address or FQDN where you've deployed OpenIDM for `localhost`.

5.1.2. Configuring Google reCAPTCHA

To use Google reCAPTCHA, you will need a Google account and your domain name (RFC 2606-compliant URLs such as `localhost` and `example.com` are acceptable for test purposes). Google then provides a Site key and a Secret key that you can include in the self-service function configuration.

For example, you can add the following reCAPTCHA code block (with appropriate keys as defined by Google) into the `selfservice-registration.json`, `selfservice-reset.json` or the `selfservice-username.json` configuration files.

```
{
    "stageConfigs" : [
        {
            "name" : "captcha",
            "recaptchaSiteKey" : "< Insert Site Key Here >",
            "recaptchaSecretKey" : "< Insert Secret Key Here >",
            "recaptchaUri" : "https://www.google.com/recaptcha/api/siteverify"
        },
```

You may also add the reCAPTCHA keys through the UI.

5.1.3. Configuring Self-Service Questions

OpenIDM uses Knowledge-based Authentication (KBA) to help users prove their identity when they perform the noted functions. In other words, they get a choice of questions configured in the following file: `selfservice.kba.json`.

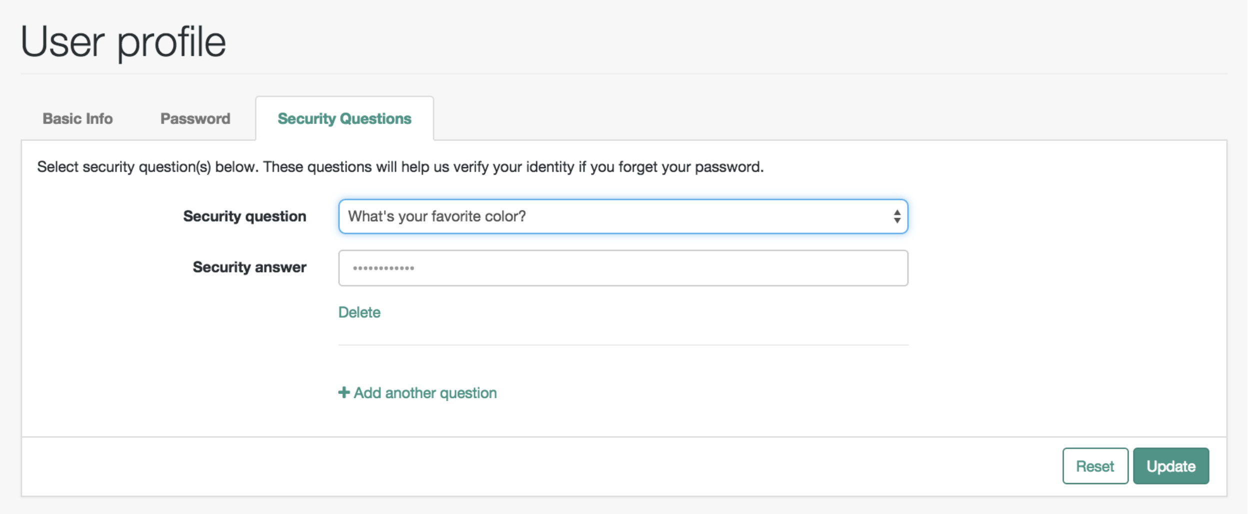
The default version of this file is straightforward:

```
{
    "kbaPropertyName" : "kbaInfo",
    "questions" : {
        "1" : {
            "en" : "What's your favorite color?",
            "en_GB" : "What's your favorite colour?",
            "fr" : "Quelle est votre couleur préférée?"
        },
        "2" : {
            "en" : "Who was your first employer?"
        }
    }
}
```

You may change or add the questions of your choice, in JSON format.

At this time, OpenIDM supports editing KBA questions only through the noted configuration file. However, individual users can configure their own questions and answers, during the User Registration process.

After a regular user logs into the Self-Service UI, that user can modify, add, and delete KBA questions under the Profile tab:



User profile

Select security question(s) below. These questions will help us verify your identity if you forget your password.

Security question: What's your favorite color?

Security answer: ••••••••••••

Note

The Self-Service KBA modules do not preserve the case of the answers when they hash the value. All answers are first converted to lowercase. If you intend to pre-populate KBA answer strings by using a mapping, or any other means that uses the `openidm.hash` function or the CLI `secureHash` mechanism, you must provide the KBA string in lowercase for the value to be matched correctly.

5.1.4. Setting a Minimum Number of Self-Service Questions

In addition, you can set a minimum number of questions that users have to define to register for their accounts. To do so, open the associated configuration file, `selfservice-registration.json`, in your *`project-dir/conf`* directory. Look for the code block that starts with `kbaSecurityAnswerDefinitionStage`:

```
{
    "name" : "kbaSecurityAnswerDefinitionStage",
    "numberOfAnswersUserMustSet" : 1,
    "kbaConfig" : null
},
```

In a similar fashion, you can set a minimum number of questions that users have to answer before OpenIDM allows them to reset their passwords. The associated configuration file is `selfservice-reset.json`, and the relevant code block is:

```
{
    "name" : "kbaSecurityAnswerVerificationStage",
    "kbaPropertyName" : "kbaInfo",
    "identityServiceUrl" : "managed/user",
    "numberOfQuestionsUserMustAnswer" : "1",
    "kbaConfig" : null
},
```

5.1.5. Enabling Social Identities in User Self-Registration

If you've configured a social identity provider as described in "*Configuring Social ID Providers*", you can enable those providers in the Admin UI. To do so, select Configure > User Registration, and enable the option associated with Social Registration.

When you activate the Social Registration option, that changes one line in the `selfservice-registration.json` file, from:

```
"name" : "userDetails",
```

to:

```
"name" : "socialUserDetails",
```

5.2. The End User and Commons User Self-Service

When all self-service features are enabled, OpenIDM includes three links on the self-service login page: `Reset your password`, `Register`, and `Forgot Username?`.

When the account registration page is used to create an account, OpenIDM creates a managed object in the OpenIDM repository, and applies default policies for managed objects.

Chapter 6
Managing the Repository

OpenIDM stores managed objects, internal users, and configuration objects in a repository. By default, the server uses OrientDB for its internal repository. In production, you must replace OrientDB with a supported JDBC repository, as described in "*Installing a Repository For Production*" in the *Installation Guide*.

This chapter describes the JDBC repository configuration, the use of mappings in the repository, and how to configure a connection to the repository over SSL. It also describes how to interact with the repository over the REST interface.

6.1. Understanding the JDBC Repository Configuration File

OpenIDM provides configuration files for each supported JDBC repository, as well as example configurations for other repositories. These configuration files are located in the `/path/to/openidm/db/*database*/conf` directory. The configuration is defined in two files:

- `datasource.jdbc-default.json`, which specifies the connection details to the repository.
- `repo.jdbc.json`, which specifies the mapping between OpenIDM resources and the tables in the repository, and includes a number of predefined queries.

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

6.1.1. Understanding the JDBC Connection Configuration File

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

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

The configuration file includes the following properties:

`driverClass`, `jndiName`, or `jtaName`

Depending on the mechanism you use to acquire the data source, set *one* of these properties:

- `"driverClass" : string`

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

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

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

- `"jndiName" : string`

  If you use JNDI to acquire the data source, set this property to the JNDI name of the data source.

  This option might be relevant if you want to run OpenIDM inside your own web container.

  Example: `"jndiName" : "jdbc/my-datasource"`

- `"jtaName" : string`

  If you use an OSGi service to acquire the data source, set this property to a stringified version of the OsgiName.

  This option would only be relevant in a highly customized deployment, for example, if you wanted to develop your own connection pool.

  Example: `"jtaName" : "osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc/openidm)"`

`jdbcUrl`

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

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

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

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

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

```
$ export OPENIDM_OPTS="-Xmx1024m -Xms1024m -Dopenidm.repo.host=localhost -Dopenidm.repo.port=3306"
$ ./startup.sh
Executing ./startup.sh...
Using OPENIDM_HOME:   /path/to/openidm
Using PROJECT_HOME:   /path/to/openidm
Using OPENIDM_OPTS:   -Xmx1024m -Xms1024m -Dopenidm.repo.host=localhost -Dopenidm.repo.port=3306
Using LOGGING_CONFIG: -Djava.util.logging.config.file=/path/to/openidm/conf/logging.properties
Using boot properties at /path/to/openidm/conf/boot/boot
.properties
-> OpenIDM version "5.0.0"
OpenIDM ready
```

`databaseName`

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

`username`

The username with which to access the JDBC database.

`password`

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

`connectionTimeout`

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

`connectionPool`

Database connection pooling configuration. The default connection pool library is Hikari (`"type" : "hikari"`).

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

- `minimumIdle`

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

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

- `maximumPoolSize`

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

By default, Hikari sets the maximum number of connections to `10`. The connection configuration files provided with OpenIDM set the maximum number of connections to `50`.

For information about the Hikari configuration parameters, see the Hikari Project Page.

OpenIDM also supports the BoneCP connection pool library. To use BoneCP, change the configuration as follows:

```
"connectionPool" : {
        "type" : "bonecp"
}
```

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

- `partitionCount`

  The partition count determines the lock segmentation in the connection pool. Each incoming connection request acquires a connection from a pool that has thread-affinity. Threads are dispatched to the appropriate lock by using a value of `threadId % partitionCount`. A partition count that is greater than 1 protects the connection pool with more than a single lock, thereby reducing lock contention.

  By default, BoneCP creates a single partition. The JDBC Connection Configuration Files provided with OpenIDM set the partition count to `4`.

- `maxConnectionsPerPartition`

  The maximum number of connections to create per partition. The maximum number of database connections is equal to `partitionCount * maxConnectionsPerPartition`. BoneCP does not create all these connections at once, but starts off with the `minConnectionsPerPartition` and gradually increases connections as required.

  By default, BoneCP creates a maximum of `20` connections per partition. The JDBC Connection Configuration Files provided with OpenIDM set the maximum connections per partition to `25`.

- `minConnectionsPerPartition`

  The number of connections to start off with, per partition. The minimum number of database connections is equal to `partitionCount * minConnectionsPerPartition`.

  By default, BoneCP starts with a minimum of `1` connection per partition. The JDBC Connection Configuration Files provided with OpenIDM set the minimum connections per partition to `5`.

For more information about the BoneCP configuration parameters, see http://www.jolbox.com/configuration.html.

6.1.2. Understanding the Database Table Configuration

An excerpt from an database table configuration file follows:

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

The configuration file includes the following properties:

`"dbType"` : string, optional

The type of database. The database type might affect the queries used and other optimizations. Supported database types include MYSQL, SQLSERVER, ORACLE, MS SQL, and DB2.

`"useDataSource"` : string, optional

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

`"maxBatchSize"`

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

`"maxTxRetry"`

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

`"queries"`

Enables you to create predefined queries that can be referenced from the configuration. For more information about predefined queries, see "Parameterized Queries". The queries are divided between those for `"genericTables"` and those for `"explicitTables"`.

The following sample extract from the default MySQL configuration file shows two credential queries, one for a generic mapping, and one for an explicit mapping. Note that the lines have been broken here for legibility only. In a real configuration file, the query would be all on one line.

```
"queries" : {
    "genericTables" : {
        "credential-query" : "SELECT fullobject FROM ${_dbSchema}.${_mainTable}
          obj INNER JOIN ${_dbSchema}.${_propTable} prop ON
          obj.id = prop.${_mainTable}_id INNER JOIN ${_dbSchema}.objecttypes
          objtype ON objtype.id = obj.objecttypes_id WHERE prop.propkey='/userName'
          AND prop.propvalue = ${username} AND objtype.objecttype = ${_resource}",
        ...
    "explicitTables" : {
        "credential-query" : "SELECT * FROM ${_dbSchema}.${_table}
          WHERE objectid = ${username} and accountStatus = 'active'",
        ...
    }
}
```

Options supported for query parameters include the following:

- A default string parameter, for example:

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

  For more information about the query function, see "openidm.query(resourceName, params, fields)".

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

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

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

- An integer parameter (`${int:propName}`).

  Use this parameter if you need query for non-string values in the database. This is particularly useful with explicit tables.

`"commands"`

Specific commands configured for to managed the database over the REST interface. Currently, only two default commands are included in the configuration:

- `purge-by-recon-expired`
- `purge-by-recon-number-of`

Both of these commands assist with removing stale reconciliation audit information from the repository, and preventing the repository from growing too large. For more information about repository commands, see "Running Queries and Commands on the Repository".

`"resourceMapping"`

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

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

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

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

6.2. Using Explicit or Generic Object Mapping With a JDBC Repository

For JDBC repositories, there are two ways of mapping OpenIDM objects to the database tables:

- *Generic mapping*, which allows arbitrary objects to be stored without special configuration or administration.

- *Explicit mapping*, which allows for optimized storage and queries by explicitly mapping objects to tables and columns in the database.

These two mapping strategies are discussed in the following sections.

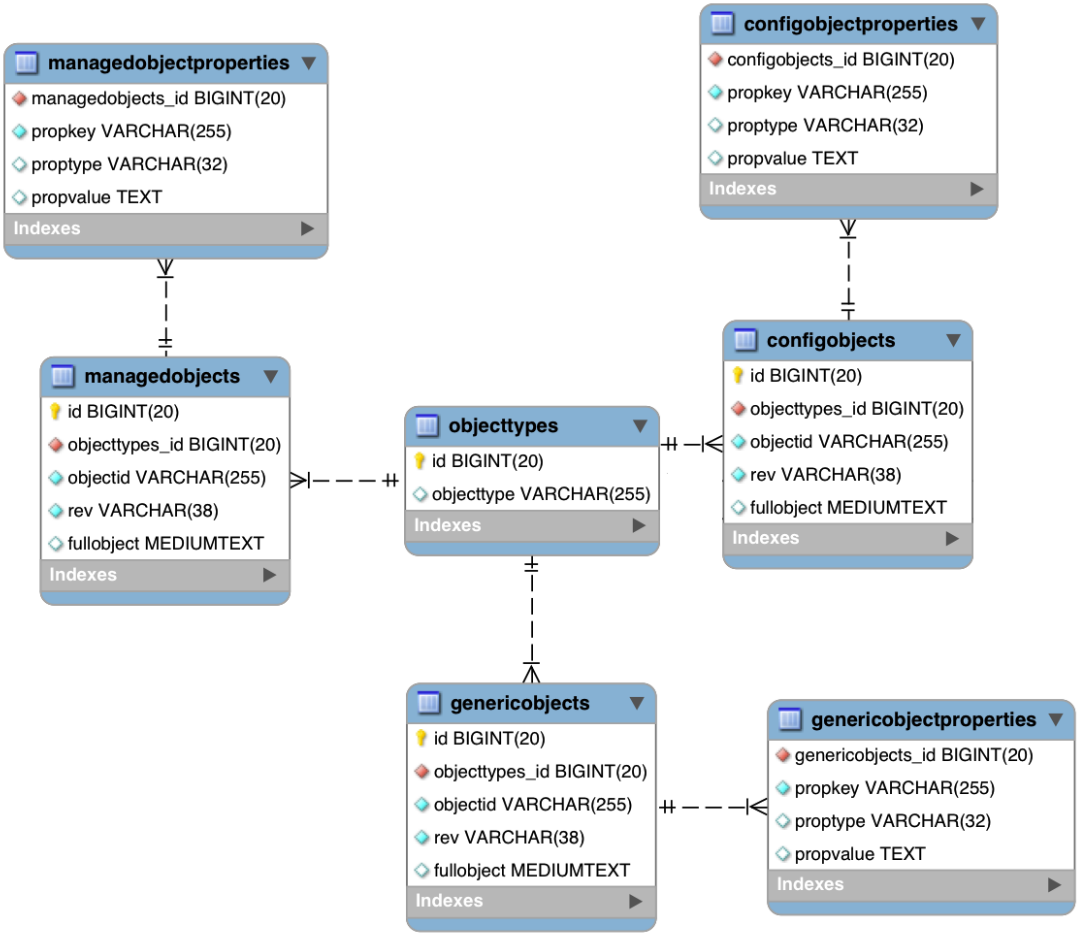
6.2.1. Using Generic Mappings

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

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

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

Generic Tables Entity Relationship Diagram



These separate tables can make the query syntax particularly complex. For example, a simple query to return user entries based on a user name would need to be implemented as follows:

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

The query can be broken down as follows:

1. Select the full object from the main table:

```
SELECT fullobject FROM ${_dbSchema}.${_mainTable} obj
```

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

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

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

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

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

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

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

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

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

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

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

`mainTable` (string, mandatory)

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

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

`propertiesTable` (string, mandatory)

Indicates the properties table, used for searches.

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

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

The properties table includes the following columns:

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

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

`searchableDefault` (boolean, optional)

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

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

`properties`

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

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

6.2.2. Improving Search Performance for Generic Mappings

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

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

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

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

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

6.2.3. Using Explicit Mappings

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

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

Note that it is possible to have a generic mapping configuration for most managed objects, *and* to have an explicit mapping that overrides the default generic mapping in certain cases. The sample configuration provided in `/path/to/openidm/db/mysql/conf/repo.jdbc-mysql-explicit-managed-user.json` has a generic mapping for managed objects, but an explicit mapping for managed user objects.

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

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

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

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

`<resource-uri>` (string, mandatory)

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

`table` (string, mandatory)

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

`objectToColumn` (string, mandatory)

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

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

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

or

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

Caution

Support for data types in columns is restricted to `String` (`VARCHAR` in the case of MySQL). If you use a different data type, such as `DATE` or `TIMESTAMP`, your database must attempt to convert from `String` to the other data type. This conversion is not guaranteed to work.

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

6.3. Configuring SSL with a JDBC Repository

To configure SSL with a JDBC repository, you need to import the CA certificate file for the server into the OpenIDM truststore. That certificate file could have a name like `ca-cert.pem`. If you have a different genuine or self-signed certificate file, substitute accordingly.

To import the CA certificate file into the OpenIDM truststore, use the **keytool** command native to the Java environment, typically located in the `/path/to/jre-version/bin` directory. On some UNIX-based systems, **/usr/bin/keytool** may link to that command.

Preparing OpenIDM for SSL with a JDBC Repository

1. Import the `ca-cert.pem` certificate into the OpenIDM truststore file with the following command:

   ```
   $ keytool \
    -importcert \
    -trustcacerts \
    -file ca-cert.pem \
    -alias "DB cert" \
    -keystore /path/to/openidm/security/truststore
   ```

   You are prompted for a keystore password. You must use the same password as is shown in the your project's `conf/boot/boot.properties` file. The default truststore password is:

   ```
   openidm.truststore.password=changeit
   ```

   After entering a keystore password, you are prompted with the following question. Assuming you have included an appropriate `ca-cert.pem` file, enter `yes`.

   ```
   Trust this certificate? [no]:
   ```

2. Open the repository connection configuration file, `datasource.jdbc-default.json` and locate the `jdbcUrl` property.

   Append `&useSSL=true` to the end of that URL.

   The value of the `jdbcUrl` property depends on your JDBC repository. The following example shows a MySQL repository, configured for SSL:

   ```
   "jdbcUrl" : "jdbc:mysql://&{openidm.repo.host}:&{openidm.repo.port}/openidm?
   allowMultiQueries=true&characterEncoding=utf8&useSSL=true"
   ```

3. Open your project's `conf/config.properties` file. Find the `org.osgi.framework.bootdelegation` property. Make sure that property includes a reference to the `javax.net.ssl` option. If you started with the default version of `config.properties` that line should now read as follows:

```
org.osgi.framework.bootdelegation=sun.*,com.sun.*,apple.*,com.apple.*,javax.net.ssl
```

4. Open your project's `conf/system.properties` file. Add the following line to that file. If appropriate, substitute the path to your own truststore:

   ```
   # Set the truststore
   javax.net.ssl.trustStore=&{launcher.install.location}/security/truststore
   ```

   Even if you are setting up this instance of OpenIDM as part of a cluster, you still need to configure this initial truststore. After this instance joins a cluster, the SSL keys in this particular truststore are replaced. For more information on clustering, see "*Clustering, Failover, and Availability*".

6.4. Interacting With the Repository Over REST

The OpenIDM repository is accessible over the REST interface, at the `openidm/repo` endpoint.

In general, you must ensure that external calls to the `openidm/repo` endpoint are protected. Native queries and free-form command actions on this endpoint are disallowed by default, as the endpoint is vulnerable to injection attacks. For more information, see "Running Queries and Commands on the Repository".

6.4.1. Changing the Repository Password

In the case of an embedded OrientDB repository, the default username and password are `admin` and `admin`. You can change the default password, by sending the following POST request on the `repo` endpoint:

```
$ curl \
 --cacert self-signed.crt \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "https://localhost:8443/openidm/repo?_action=updateDbCredentials&user=admin&password=newPassword"
```

You must restart OpenIDM for the change to take effect.

6.4.2. Running Queries and Commands on the Repository

Free-form commands and native queries on the repository are disallowed by default and should remain so in production to reduce the risk of injection attacks.

Common filter expressions, called with the `_queryFilter` keyword, enable you to form arbitrary queries on the repository, using a number of supported filter operations. For more information on these filter operations, see "Constructing Queries". Parameterized or predefined queries and commands (using the `_queryId` and `_commandId` keywords) can be authorized on the repository for external calls if necessary. For more information, see "Parameterized Queries".

Running commands on the repository is supported primarily from scripts. Certain scripts that interact with the repository are provided by default, for example, the scripts that enable you to purge the repository of reconciliation audit records.

You can define your own commands, and specify them in the database table configuration file (either `repo.orientdb.json` or `repo.jdbc.json`). In the following simple example, a command is called to clear out UI notification entries from the repository, for specific users.

The command is defined in the repository configuration file, as follows:

```
"commands" : {
"delete-notifications-by-id" : "DELETE FROM ui_notification WHERE receiverId = ${username}"
...
},
```

The command can be called from a script, as follows:

```
openidm.action("repo/ui/notification", "command", {},
{ "commandId" : "delete-notifications-by-id", "userName" : "scarter"});
```

Exercise caution when allowing commands to be run on the repository over the REST interface, as there is an attached risk to the underlying data.

Chapter 7
Configuring the Server

This chapter describes how OpenIDM loads and stores its configuration, how the configuration can be changed, and specific configuration recommendations in a production environment.

The configuration is defined in a combination of `.properties` files, container configuration files, and dynamic configuration objects. Most of the configuration files are stored in your project's `conf/` directory. Note that you might see files with a `.patch` extension in the `conf/` and `db/repo/conf/` directories. These files specify differences relative to the last released version of OpenIDM and are used by the update mechanism. They do not affect your current configuration.

When the same configuration object is declared in more than one location, the configuration is loaded with the following order of precedence:

1. System properties passed in on startup through the `OPENIDM_OPTS` environment variable
2. Properties declared in the `project-dir/conf/system.properties` file
3. Properties declared in the `project-dir/conf/boot/boot.properties` file
4. Properties set explicitly in the various `project-dir/conf/*.json` files

Properties that are set using the first three options are not stored in the repository. You can therefore use these mechanisms to set different configurations for multiple nodes participating in a cluster.

To set the configuration in the `OPENIDM_OPTS` environment variable, export that variable before startup. The following example starts OpenIDM with a different keystore and truststore:

```
$ export OPENIDM_OPTS="-Xmx1024m -Xms1024m \
 -Dopenidm.keystore.location=/path/to/keystore.jceks -Dopenidm.truststore.location=/path/to/truststore"
$ ./startup.sh
Executing ./startup.sh...
Using OPENIDM_HOME:   /path/to/openidm
Using PROJECT_HOME:   /path/to/openidm
Using OPENIDM_OPTS:   -Xmx1024m -Xms1024m -Dopenidm.keystore.location=/path/to/keystore.jceks
                      -Dopenidm.truststore.location=/path/to/truststore
Using LOGGING_CONFIG: -Djava.util.logging.config.file=/path/to/openidm/conf/logging.properties
Using boot properties at /path/to/openidm/conf/boot/boot
.properties
-> OpenIDM version "5"
OpenIDM ready
```

Configuration properties that are explicitly set in `project-dir/conf/*.json` files are stored in the internal repository. You can manage these configuration objects by using the REST interface or by using the

JSON files themselves. Several aspects of the configuration can also be managed by using the Admin UI, as described in "Configuring the Server from the Admin UI".

7.1. Configuration Objects

OpenIDM exposes internal configuration objects in JSON format. Configuration elements can be either single instance or multiple instance for an OpenIDM installation.

7.1.1. Single Instance Configuration Objects

Single instance configuration objects correspond to services that have at most one instance per installation. JSON file views of these configuration objects are named `object-name.json`.

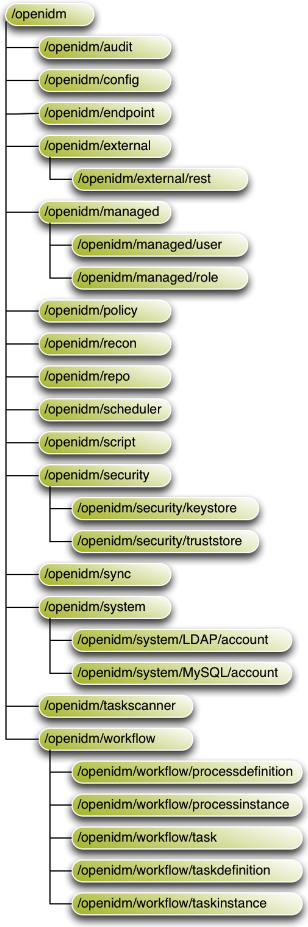
The following list describes the single instance configuration objects:

- The `audit` configuration specifies how audit events are logged.
- The `authentication` configuration controls REST access.
- The `cluster` configuration defines how one OpenIDM instance can be configured in a cluster.
- The `endpoint` configuration controls any custom REST endpoints.
- The `info` configuration points to script files for the customizable information service.
- The `managed` configuration defines managed objects and their schemas.
- The `policy` configuration defines the policy validation service.
- The `process access` configuration defines access to configured workflows.
- The `repo.repo-type` configuration such as `repo.orientdb` or `repo.jdbc` configures the internal repository.
- The `router` configuration specifies filters to apply for specific operations.
- The `script` configuration defines the parameters that are used when compiling, debugging, and running JavaScript and Groovy scripts.
- The `sync` configuration defines the mappings that OpenIDM uses when it synchronizes and reconciles managed objects.
- The `ui` configuration defines the configurable aspects of the default user interfaces.
- The `workflow` configuration defines the configuration of the workflow engine.

OpenIDM stores managed objects in the repository, and exposes them under `/openidm/managed`. System objects on external resources are exposed under `/openidm/system`.

The following image shows the paths to objects in the OpenIDM namespace.

OpenIDM Namespaces and Object Paths



7.1.2. Multiple Instance Configuration Objects

Multiple instance configuration objects correspond to services that can have many instances per installation. Multiple instance configuration objects are named `objectname/instancename`, for example, `provisioner.openicf/xml`.

JSON file views of these configuration objects are named `objectname-instancename.json`, for example, `provisioner.openicf-xml.json.`

OpenIDM provides the following multiple instance configuration objects:

- Multiple `schedule` configurations can run reconciliations and other tasks on different schedules.
- Multiple `provisioner.openicf` configurations correspond to the resources connected to OpenIDM.
- Multiple `servletfilter` configurations can be used for different servlet filters such as the Cross Origin and GZip filters.

7.2. Changing the Default Configuration

When you change OpenIDM's configuration objects, take the following points into account:

- OpenIDM's authoritative configuration source is the internal repository. JSON files provide a view of the configuration objects, but do not represent the authoritative source.

  OpenIDM updates JSON files after making configuration changes, whether those changes are made through REST access to configuration objects, or through edits to the JSON files.

- OpenIDM recognizes changes to JSON files when it is running. OpenIDM *must* be running when you delete configuration objects, even if you do so by editing the JSON files.
- Avoid editing configuration objects directly in the internal repository. Rather, edit the configuration over the REST API, or in the configuration JSON files to ensure consistent behavior and that operations are logged.
- OpenIDM stores its configuration in the internal database by default. If you remove an OpenIDM instance and do not specifically drop the repository, the configuration remains in effect for a new OpenIDM instance that uses that repository. For testing or evaluation purposes, you can disable this *persistent configuration* in the `conf/system.properties` file by uncommenting the following line:

  ```
  # openidm.config.repo.enabled=false
  ```

  Disabling persistent configuration means that OpenIDM will store its configuration in memory only. You should not disable persistent configuration in a production environment.

7.3. Configuring the Server for Production

Out of the box, OpenIDM is configured to make it easy to install and evaluate. Specific configuration changes are required before you deploy the server in a production environment.

7.3.1. Configuring a Production Repository

By default, OpenIDM uses OrientDB for its internal repository so that you do not have to install a database in order to evaluate OpenIDM. Before you use OpenIDM in production, you must replace OrientDB with a supported repository.

For more information, see "*Installing a Repository For Production*" in the *Installation Guide*.

7.3.2. Disabling Automatic Configuration Updates

By default, OpenIDM polls the JSON files in the `conf` directory periodically for any changes to the configuration. In a production system, it is recommended that you disable automatic polling for updates to prevent untested configuration changes from disrupting your identity service.

To disable automatic polling for configuration changes, edit the `conf/system.properties` file for your project, and uncomment the following line:

```
# openidm.fileinstall.enabled=false
```

This setting also disables the file-based configuration view, which means that OpenIDM reads its configuration only from the repository.

Before you disable automatic polling, you must have started the OpenIDM instance at least once to ensure that the configuration has been loaded into the repository. Be aware, if automatic polling is enabled, OpenIDM immediately uses changes to scripts called from a JSON configuration file.

When your configuration is complete, you can disable writes to configuration files. To do so, add the following line to the `conf/config.properties` file for your project:

```
felix.fileinstall.enableConfigSave=false
```

7.3.3. Communicating Through a Proxy Server

To set up OpenIDM to communicate through a proxy server, you must use JVM parameters that identify the proxy host system, and the OpenIDM port number.

If you have configured OpenIDM behind a proxy server, include JVM properties from the following table, in the OpenIDM startup script:

JVM Proxy Properties

JVM Property	Example Values	Description
`-Dhttps.proxyHost`	proxy.example.com, 192.168.0.1	Hostname or IP address of the proxy server
`-Dhttps.proxyPort`	8443, 9443	Port number used by OpenIDM

If an insecure port is acceptable, you can also use the `-Dhttp.proxyHost` and `-Dhttp.proxyPort` options. You can add these JVM proxy properties to the value of `OPENIDM_OPTS` in your startup script (`startup.sh` or `startup.bat`):

```
# Only set OPENIDM_OPTS if not already set
[ -z "$OPENIDM_OPTS" ] && OPENIDM_OPTS="-Xmx1024m -Xms1024m -Dhttps.proxyHost=localhost -
Dhttps.proxyPort=8443"
```

7.4. Configuring the Server Over REST

OpenIDM exposes configuration objects under the `/openidm/config` context path.

You can list the configuration on the local host by performing a GET request on `http://localhost:8080/openidm/config`. The examples shown in this section are based on first OpenIDM sample, described in "Reconciling an XML File Resource" in the *Samples Guide*.

The following REST call includes excerpts of the default configuration for an OpenIDM instance started with Sample 1:

```
$ curl \
 --request GET \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 http://localhost:8080/openidm/config
{
  "_id" : "",
  "configurations" : [ {
    "_id" : "endpoint/usernotifications",
    "pid" : "endpoint.95b46fcd-f0b7-4627-9f89-6f3180c826e4",
    "factoryPid" : "endpoint"
  }, {
    "_id" : "router",
    "pid" : "router",
    "factoryPid" : null
  },
   ...
  {
    "_id" : "endpoint/reconResults",
    "pid" : "endpoint.ad3f451c-f34e-4096-9a59-0a8b7bc6989a",
    "factoryPid" : "endpoint"
  }, {
    "_id" : "endpoint/gettasksview",
    "pid" : "endpoint.bc400043-f6db-4768-92e5-ebac0674e201",
    "factoryPid" : "endpoint"
  },
  ...
  {
    "_id" : "workflow",
    "pid" : "workflow",
    "factoryPid" : null
  }, {
    "_id" : "ui.context/selfservice",
    "pid" : "ui.context.537a5838-217b-4f67-9301-3fde19a51784",
    "factoryPid" : "ui.context"
  } ]
}
```

Single instance configuration objects are located under `openidm/config/object-name`. The following example shows the Sample 1 `audit` configuration. The output has been cropped for legibility:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 "http://localhost:8080/openidm/config/audit"
{
  "_id": "audit",
  "auditServiceConfig": {
    "handlerForQueries": "repo",
    "availableAuditEventHandlers": [
      "org.forgerock.audit.handlers.csv.CsvAuditEventHandler",
      "org.forgerock.audit.handlers.elasticsearch.ElasticsearchAuditEventHandler",
      "org.forgerock.audit.handlers.jms.JmsAuditEventHandler",
      "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
      "org.forgerock.openidm.audit.impl.RepositoryAuditEventHandler",
      "org.forgerock.openidm.audit.impl.RouterAuditEventHandler",
      "org.forgerock.audit.handlers.splunk.SplunkAuditEventHandler",
      "org.forgerock.audit.handlers.syslog.SyslogAuditEventHandler"
    ],
    "filterPolicies": {
      "value": {
        "excludeIf": [
          "/access/http/request/headers/Authorization",
          "/access/http/request/headers/X-OpenIDM-Password",
          "/access/http/request/cookies/session-jwt",
          "/access/http/response/headers/Authorization",
          "/access/http/response/headers/X-OpenIDM-Password"
        ],
        "includeIf": []
      }
    }
  },
  "eventHandlers": [
    {
      "class": "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
      "config": {
        "name": "json",
        "logDirectory": "/path/to/openidm/audit",
        "topics": [
          "access",
          "activity",
          "recon",
          "sync",
          "authentication",
          "config"
        ]
      }
    }
,
...
}
```

Multiple instance configuration objects are found under `openidm/config/object-name/instance-name`.

The following example shows the configuration for the XML connector shown in the first OpenIDM sample. The output has been cropped for legibility:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 "http://localhost:8080/openidm/config/provisioner.openicf/xml"
{
  "_id": "provisioner.openicf/xml",
  "name": "xmlfile",
  "connectorRef": {
    "bundleName": "org.forgerock.openicf.connectors.xml-connector",
    "bundleVersion": "[1.1.0.3,1.2.0.0)",
    "connectorName": "org.forgerock.openicf.connectors.xml.XMLConnector"
  },
  "producerBufferSize": 100,
  "connectorPoolingSupported": true,
  "poolConfigOption": {
    "maxObjects": 1,
    "maxIdle": 1,
    "maxWait": 150000,
    "minEvictableIdleTimeMillis": 120000,
    "minIdle": 1
  },
  "operationTimeout": {
    "CREATE": -1,
    "TEST": -1,
    "AUTHENTICATE": -1,
    "SEARCH": -1,
    "VALIDATE": -1,
    "GET": -1,
    "UPDATE": -1,
    "DELETE": -1,
    "SCRIPT_ON_CONNECTOR": -1,
    "SCRIPT_ON_RESOURCE": -1,
    "SYNC": -1,
    "SCHEMA": -1
  },
  "configurationProperties": {
    "xsdIcfFilePath": "/path/to/openidm/samples/sample1/data/resource-schema-1.xsd",
    "xsdFilePath": "/path/to/openidm/samples/sample1/data/resource-schema-extension.xsd",
    "xmlFilePath": "/path/to/openidm/samples/sample1/data/xmlConnectorData.xml"
  }
,
...
}
```

You can change the configuration over REST by using an HTTP PUT or HTTP PATCH request to modify the required configuration object.

The following example uses a PUT request to modify the configuration of the scheduler service, increasing the maximum number of threads that are available for the concurrent execution of scheduled tasks:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PUT \
 --data '{
    "threadPool": {
        "threadCount": "20"
    },
    "scheduler": {
        "executePersistentSchedules": "&{openidm.scheduler.execute.persistent.schedules}"
    }
}' \
 "http://localhost:8080/openidm/config/scheduler"
{
  "_id" : "scheduler",
  "threadPool": {
    "threadCount": "20"
  },
  "scheduler": {
    "executePersistentSchedules": "true"
  }
}
```

The following example uses a PATCH request to reset the number of threads to their original value.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PATCH \
 --data '[
    {
      "operation" : "replace",
      "field" : "/threadPool/threadCount",
      "value" : "10"
    }
 ]' \
 "http://localhost:8080/openidm/config/scheduler"
{
  "_id": "scheduler",
  "threadPool": {
    "threadCount": "10"
  },
  "scheduler": {
    "executePersistentSchedules": "true"
  }
}
```

Note

Multi-version concurrency control (MVCC) is not supported for configuration objects so you do not need to specify a revision during updates to the configuration, and no revision is returned in the output.

For more information about using the REST API to update objects, see "*REST API Reference*".

7.5. Using Property Value Substitution In the Configuration

In an environment where you have more than one OpenIDM instance, you might require a configuration that is similar, but not identical, across the different OpenIDM hosts. OpenIDM supports variable replacement in its configuration which means that you can modify the effective configuration according to the requirements of a specific environment or OpenIDM instance.

Property substitution enables you to achieve the following:

- Define a configuration that is specific to a single OpenIDM instance, for example, setting the location of the keystore on a particular host.
- Define a configuration whose parameters vary between different environments, for example, the URLs and passwords for test, development, and production environments.
- Disable certain capabilities on specific nodes. For example, you might want to disable the workflow engine on specific instances.

When OpenIDM starts up, it combines the system configuration, which might contain specific environment variables, with the defined OpenIDM configuration properties. This combination makes up the effective configuration for that OpenIDM instance. By varying the environment properties, you can change specific configuration items that vary between OpenIDM instances or environments.

Property references are contained within the construct `&{ }`. When such references are found, OpenIDM replaces them with the appropriate property value, defined in the `boot.properties` file.

For properties that would usually be encrypted, such as passwords, OpenIDM does not encrypt the property reference. You can therefore reference an obfuscated property value as shown in the following example:

Specify the reference in the configuration file:

```
{
...
"password" : "&{openidm.repo.password}",
...
}
```

Provide the encrypted or obfuscated property value in the `boot.properties` file:

```
openidm.repo.password=OBF:1jmv1usdf1t3b1vuz1sfgsb1t2v1ufs1jkn
```

The following examples demonstrate additional use cases for property value substitution.

Using Separate OpenIDM Environments

The following example defines two separate OpenIDM environments - a development environment and a production environment. You can specify the environment at startup time and, depending on the environment, the database URL is set accordingly.

The environments are defined by adding the following lines to the `conf/boot.properties` file:

```
PROD.location=production
DEV.location=development
```

The database URL is then specified as follows in the `repo.orientdb.json` file:

```
{
    "dbUrl" : "plocal:./db/&{&{environment}.location}-openidm",
    ...
}
```

The effective database URL is determined by setting the `OPENIDM_OPTS` environment variable when you start OpenIDM. To use the production environment, start OpenIDM as follows:

```
$ export OPENIDM_OPTS="-Xmx1024m -Xms1024m -Denvironment=PROD"
$ ./startup.sh
```

To use the development environment, start OpenIDM as follows:

```
$ export OPENIDM_OPTS="-Xmx1024m -Xms1024m -Denvironment=DEV"
$ ./startup.sh
```

7.5.1. Using Property Value Substitution With System Properties

You can use property value substitution in conjunction with the system properties, to modify the configuration according to the system on which the OpenIDM instance runs.

Custom Audit Log Location

The following example modifies the `audit.json` file so that the JSON audit logs are written to the user's directory. The `user.home` property is a default Java System property:

```
"eventHandlers" : [
    {
        "class" : "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
        "config" : {
            "name" : "json",
            "logDirectory" : "&{user.home}/audit",
            ...
        }
    },
...
```

You can define *nested* properties (that is a property definition within another property definition) and you can combine system properties and boot properties.

Defining Different Ports in the Configuration

The following example uses the `user.country` property, a default Java system property. The example defines specific LDAP ports, depending on the country (identified by the country code) in the `boot`

`.properties` file. The value of the LDAP port (set in the `provisioner.openicf-ldap.json` file) depends on the value of the `user.country` system property.

The port numbers are defined in the `boot.properties` file as follows:

```
openidm.NO.ldap.port=2389
openidm.EN.ldap.port=3389
openidm.US.ldap.port=1389
```

The following excerpt of the `provisioner.openicf-ldap.json` file shows how the value of the LDAP port is eventually determined, based on the system property:

```
"configurationProperties" :
   {
      "credentials" : "Passw0rd",
      "port" : "&{openidm.&{user.country}.ldap.port}",
      "principal" : "cn=Directory Manager",
      "baseContexts" :
         [
            "dc=example,dc=com"
         ],
      "host" : "localhost"
   }
```

7.5.2. Limitations of Property Value Substitution

Note the following limitations when you use property value substitution:

- You cannot reference complex objects or properties with syntaxes other than string. Property values are resolved from the `boot.properties` file or from the system properties and the value of these properties is always in string format.

  Property substitution of boolean values is currently only supported in stringified format, that is, resulting in `"true"` or `"false"`.

7.6. Setting the Script Configuration

The script configuration file (`conf/script.json`) enables you to modify the parameters that are used when compiling, debugging, and running JavaScript and Groovy scripts.

The default `script.json` file includes the following parameters:

properties

Any custom properties that should be provided to the script engine.

ECMAScript

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

- `javascript.recompile.minimumInterval` - minimum time after which a script can be recompiled.

  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).

Groovy

Specifies 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 log level for Groovy scripts. Possible values are `none`, `likely`, `possible`, and `paranoia`.
- `groovy.source.encoding` - the encoding format for Groovy scripts. Possible values are `UTF-8` and `US-ASCII`.
- `groovy.target.directory` - the directory to which compiled Groovy classes will be output. The default directory is `*install-dir*/classes`.
- `groovy.target.bytecode` - the bytecode version that is used to compile Groovy scripts. The default version is `1.5`.
- `groovy.classpath` - the directory in which 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" : "/&{launcher.install.location}/lib/http-builder-0.7.1.jar:
         /&{launcher.install.location}/lib/json-lib-2.3-jdk15.jar:
         /&{launcher.install.location}/lib/xml-resolver-1.2.jar:
         /&{launcher.install.location}/lib/commons-collections-3.2.1.jar",
```

- `groovy.output.verbose` - specifies the verbosity of stack traces. Boolean, `true` or `false`.
- `groovy.output.debug` - specifies whether debugging messages are output. Boolean, `true` or `false`.
- `groovy.errors.tolerance` - sets the number of non-fatal errors that can occur before a compilation is aborted. The default is `10` errors.
- `groovy.script.extension` - specifies the file extension for Groovy scripts. The default is `.groovy`.
- `groovy.script.base` - defines the base class for Groovy scripts. By default any class extends `groovy.lang.Script`.
- `groovy.recompile` - indicates whether scripts can be recompiled. Boolean, `true` or `false`, with default `true`.

- `groovy.recompile.minimumInterval` - sets the minimum time between which Groovy scripts can be recompiled.

  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).

- `groovy.target.indy` - specifies whether a Groovy indy test can be used. Boolean, `true` or `false`, with default `true`.

- `groovy.disabled.global.ast.transformations` - specifies a list of disabled Abstract Syntax Transformations (ASTs).

sources

Specifies the locations in which OpenIDM expects to find JavaScript and Groovy scripts that are referenced in the configuration.

The following excerpt of the `script.json` file shows the default locations:

```
...
"sources" : {
    "default" : {
        "directory" : "&{launcher.install.location}/bin/defaults/script"
    },
    "install" : {
        "directory" : "&{launcher.install.location}"
    },
    "project" : {
        "directory" : "&{launcher.project.location}"
    },
    "project-script" : {
        "directory" : "&{launcher.project.location}/script"
    }
...
```

> **Note**
>
> The order in which locations are listed in the `sources` property is important. Scripts are loaded from the *bottom up* in this list, that is, scripts found in the last location on the list are loaded first.

> **Note**
>
> By default, debug information (such as file name and line number) is excluded from JavaScript exceptions. To troubleshoot script exceptions, you can include debug information by changing the following setting to `true` in your project's `conf/boot/boot.properties` file:
>
> ```
> javascript.exception.debug.info=false
> ```

Including debug information in a production environment is not recommended.

7.7. Calling a Script From a Configuration File

You can call a script from within a configuration file by providing the script source, or by referencing a file that contains the script source. For example:

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

or

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

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 following sample excerpts from configuration files indicate how scripts can be called.

The following example (included in the `sync.json` file) returns `true` if the `employeeType` is equal to `external`, otherwise returns `false`. This script can be useful during reconciliation to establish whether a target object should be included in the reconciliation process, or should be ignored:

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

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

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

The following example (included in the `router.json` file) shows 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"
        }
    }
}
```

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"
    }
}
```

Script variables are not necessarily simple `key:value` pairs. A script variable can be any arbitrarily complex JSON object.

Chapter 8 Accessing Data Objects

OpenIDM supports a variety of objects that can be addressed via a URL or URI. You can access data objects by using scripts (through the Resource API) or by using direct HTTP calls (through the REST API).

The following sections describe these two methods of accessing data objects, and provide information on constructing and calling data queries.

8.1. Accessing Data Objects By Using Scripts

OpenIDM's uniform programming model means that all objects are queried and manipulated in the same way, using the Resource API. The URL or URI that is used to identify the target object for an operation depends on the object type. For an explanation of object types, see "*Data Models and Objects Reference*". For more information about scripts and the objects available to scripts, see "*Scripting Reference*".

You can use the Resource API to obtain managed, system, configuration, and repository objects, as follows:

```
val = openidm.read("managed/organization/mysampleorg")
val = openidm.read("system/mysystem/account")
val = openidm.read("config/custom/mylookuptable")
val = openidm.read("repo/custom/mylookuptable")
```

For information about constructing an object ID, see "URI Scheme".

You can update entire objects with the `update()` function, as follows:

```
openidm.update("managed/organization/mysampleorg", rev, object)
openidm.update("system/mysystem/account", rev, object)
```

You can apply a partial update to a managed or system object by using the `patch()` function:

```
openidm.patch("managed/organization/mysampleorg", rev, value)
```

The `create()`, `delete()`, and `query()` functions work the same way.

8.2. Accessing Data Objects By Using the REST API

OpenIDM provides RESTful access to data objects via ForgeRock's Common REST API. To access the repository over REST, you can use a client application like Postman, or RESTClient for Firefox. Alternatively you can use the **curl** command-line utility that is included with most operating systems. For more information about **curl**, see https://github.com/bagder/curl.

For a comprehensive overview of the REST API, see "*REST API Reference*".

To obtain a managed object through the REST API, depending on your security settings and authentication configuration, perform an HTTP GET on the corresponding URL, for example `http://localhost:8080/openidm/managed/organization/mysampleorg`.

By default, the HTTP GET returns a JSON representation of the object.

In general, you can map any HTTP request to the corresponding `openidm.method` call. The following example shows how the parameters provided in an `openidm.query` request correspond with the key-value pairs that you would include in a similar HTTP GET request:

Reading an object using the Resource API:

```
openidm.query("managed/user", { "_queryId": "query-all" }, ["userName","sn"])
```

Reading an object using the REST API:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user?_queryId=query-all&_fields=userName,sn"
```

8.3. Defining and Calling Queries

OpenIDM supports an advanced query model that enables you to define queries, and to call them over the REST or Resource API. Three types of queries are supported, on both managed, and system objects:

- Common filter expressions
- Parameterized, or predefined queries
- Native query expressions

Each of these mechanisms is discussed in the following sections.

8.3.1. Common Filter Expressions

The ForgeRock REST API defines common filter expressions that enable you to form arbitrary queries using a number of supported filter operations. This query capability is the standard way to query data if no predefined query exists, and is supported for all managed and system objects.

Common filter expressions are useful in that they do not require knowledge of how the object is stored and do not require additions to the repository configuration.

Common filter expressions are called with the `_queryFilter` keyword. The following example uses a common filter expression to retrieve managed user objects whose user name is Smith:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 'http://localhost:8080/openidm/managed/user?_queryFilter=userName+eq+"smith"'
```

The filter is URL encoded in this example. The corresponding filter using the resource API would be:

```
openidm.query("managed/user", { "_queryFilter" : '/userName eq "smith"' });
```

Note that, this JavaScript invocation is internal and is not subject to the same URL-encoding requirements that a GET request would be. Also, because JavaScript supports the use of single quotes, it is not necessary to escape the double quotes in this example.

For a list of supported filter operations, see "Constructing Queries".

Note that using common filter expressions to retrieve values from arrays is currently not supported. If you need to search within an array, you should set up a predefined (parameterized) in your repository configuration. For more information, see "Parameterized Queries".

8.3.2. Parameterized Queries

Managed objects in the supported OpenIDM repositories can be accessed using a parameterized query mechanism. Parameterized queries on repositories are defined in the repository configuration (`repo.*.json`) and are called by their `_queryId`.

Parameterized queries provide precise control over the query that is executed. Such control might be useful for tuning, or for performing database operations such as aggregation (which is not possible with a common filter expression.)

Parameterized queries provide security and portability for the query call signature, regardless of the backend implementation. Queries that are exposed over the REST interface *must* be parameterized queries to guard against injection attacks and other misuse. Queries on the officially supported repositories have been reviewed and hardened against injection attacks.

For system objects, support for parameterized queries is restricted to `_queryId=query-all-ids`. There is currently no support for user-defined parameterized queries on system objects. Typically, parameterized queries on system objects are not called directly over the REST interface, but are issued from internal calls, such as correlation queries.

A typical query definition is as follows:

```

"query-all-ids" : "select _openidm_id from ${unquoted:_resource}"
```

To call this query, you would reference its ID, as follows:

```
?_queryId=query-all-ids
```

The following example calls query-all-ids over the REST interface:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 "http://localhost:8080/openidm/managed/user?_queryId=query-all-ids"
```

8.3.3. Native Query Expressions

Native query expressions are supported for all managed objects and system objects, and can be called directly, rather than being defined in the repository configuration.

Native queries are intended specifically for internal callers, such as custom scripts, and should be used only in situations where the common filter or parameterized query facilities are insufficient. For example, native queries are useful if the query needs to be generated dynamically.

The query expression is specific to the target resource. For repositories, queries use the native language of the underlying data store. For system objects that are backed by OpenICF connectors, queries use the applicable query language of the system resource.

Native queries on the repository are made using the _queryExpression keyword. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 "http://localhost:8080/openidm/managed/user?_queryExpression=select+from+managed_user"
```

Unless you have specifically enabled native queries over REST, the previous command returns a 403 access denied error message. Native queries are not portable and do not guard against injection attacks. Such query expressions should therefore not be used or made accessible over the REST interface or over HTTP in production environments. They should be used only via the internal Resource API. If you want to enable native queries over REST for development, see "Protect Sensitive REST Interface URLs".

Alternatively, if you really need to expose native queries over HTTP, in a selective manner, you can design a custom endpoint to wrap such access.

8.3.4. Constructing Queries

The openidm.query function enables you to query OpenIDM managed and system objects. The query syntax is openidm.query(id, params), where id specifies the object on which the query should be performed and params provides the parameters that are passed to the query, either _queryFilter or _queryID. For example:

```
var params = {
    '_queryFilter' : 'givenName co "' + sourceCriteria + '" or ' + 'sn co "' + sourceCriteria + '"'
};
var results = openidm.query("system/ScriptedSQL/account", params)
```

Over the REST interface, the query filter is specified as `_queryFilter=filter`, for example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=userName+eq+"Smith"'
```

Note the use of double-quotes around the search term: `Smith`. In `_queryFilter` expressions, string values *must* use double-quotes. Numeric and boolean expressions should not use quotes.

When called over REST, you must URL encode the filter expression. The following examples show the filter expressions using the resource API and the REST API, but do not show the URL encoding, to make them easier to read.

Note that, for generic mappings, any fields that are included in the query filter (for example `userName` in the previous query), must be explicitly defined as *searchable*, if you have set the global `searchableDefault` to false. For more information, see "Improving Search Performance for Generic Mappings".

The *filter* expression is constructed from the building blocks shown in this section. In these expressions the simplest *json-pointer* is a field of the JSON resource, such as `userName` or `id`. A JSON pointer can, however, point to nested elements.

> **Note**
>
> You can also use the negation operator (*!*) to help construct a query. For example, a **_queryFilter=!(userName+eq+"jdoe")** query would return every `userName` except for `jdoe`.

You can set up query filters with one of the following types of expressions.

8.3.4.1. Comparison Expressions

- Equal queries (see "Querying Objects That Equal the Given Value")
- Contains queries (see "Querying Objects That Contain the Given Value")
- Starts with queries (see "Querying Objects That Start With the Given Value")
- Less than queries (see "Querying Objects That Are Less Than the Given Value")
- Less than or equal to queries (see "Querying Objects That Are Less Than or Equal to the Given Value")

- Greater than queries (see "Querying Objects That Are Greater Than the Given Value")
- Greater than or equal to queries (see "Querying Objects That Are Greater Than or Equal to the Given Value")

> **Note**
>
> Certain system endpoints also support `EndsWith` and `ContainsAllValues` queries. However, such queries are *not supported* for managed objects and have not been tested with all supported OpenICF connectors.

8.3.4.1.1. Querying Objects That Equal the Given Value

This is the associated JSON comparison expression: `json-pointer eq json-value`.

Review the following example:

```
"_queryFilter" : '/givenName eq "Dan"'
```

The following REST call returns the user name and given name of all managed users whose first name (`givenName`) is "Dan":

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=givenName+eq+"Dan"&_fields=userName,givenName'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 3,
  "result": [
    {
      "givenName": "Dan",
      "userName": "dlangdon"
    },
    {
      "givenName": "Dan",
      "userName": "dcope"
    },
    {
      "givenName": "Dan",
      "userName": "dlanoway"
    }
}
```

8.3.4.1.2. Querying Objects That Contain the Given Value

This is the associated JSON comparison expression: `json-pointer co json-value`.

Review the following example:

```
"_queryFilter" : '/givenName co "Da"'
```

The following REST call returns the user name and given name of all managed users whose first name (`givenName`) contains "Da":

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=givenName+co+"Da"&_fields=userName,givenName'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 10,
  "result": [
    {
      "givenName": "Dave",
      "userName": "djensen"
    },
    {
      "givenName": "David",
      "userName": "dakers"
    },
    {
      "givenName": "Dan",
      "userName": "dlangdon"
    },
    {
      "givenName": "Dan",
      "userName": "dcope"
    },
    {
      "givenName": "Dan",
      "userName": "dlanoway"
    },
    {
      "givenName": "Daniel",
      "userName": "dsmith"
    }
,
...
}
```

8.3.4.1.3. Querying Objects That Start With the Given Value

This is the associated JSON comparison expression: `json-pointer sw json-value`.

Review the following example:

```
"_queryFilter" : '/sn sw "Jen"'
```

The following REST call returns the user names of all managed users whose last name (`sn`) starts with "Jen":

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=sn+sw+"Jen"&_fields=userName'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 4,
  "result": [
    {
      "userName": "bjensen"
    },
    {
      "userName": "djensen"
    },
    {
      "userName": "cjenkins"
    },
    {
      "userName": "mjennings"
    }
  ]
}
```

8.3.4.1.4. Querying Objects That Are Less Than the Given Value

This is the associated JSON comparison expression: `json-pointer lt json-value`.

Review the following example:

```
"_queryFilter" : '/employeeNumber lt 5000'
```

The following REST call returns the user names of all managed users whose `employeeNumber` is lower than 5000:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=employeeNumber+lt+5000&_fields=userName
,employeeNumber'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 4999,
  "result": [
    {
      "employeeNumber": 4907,
      "userName": "jnorris"
    },
    {
      "employeeNumber": 4905,
      "userName": "afrancis"
    },
    {
      "employeeNumber": 3095,
      "userName": "twhite"
    },
    {
      "employeeNumber": 3921,
      "userName": "abasson"
    },
    {
      "employeeNumber": 2892,
      "userName": "dcarter"
    }
...
  ]
}
```

8.3.4.1.5. Querying Objects That Are Less Than or Equal to the Given Value

This is the associated JSON comparison expression: *json-pointer* le *json-value*.

Review the following example:

```
"_queryFilter" : '/employeeNumber le 5000'
```

The following REST call returns the user names of all managed users whose employeeNumber is 5000 or less:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=employeeNumber+le+5000&_fields=userName
,employeeNumber'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 5000,
  "result": [
    {
      "employeeNumber": 4907,
      "userName": "jnorris"
    },
    {
      "employeeNumber": 4905,
      "userName": "afrancis"
    },
    {
      "employeeNumber": 3095,
      "userName": "twhite"
    },
    {
      "employeeNumber": 3921,
      "userName": "abasson"
    },
    {
      "employeeNumber": 2892,
      "userName": "dcarter"
    }
...
  ]
}
```

8.3.4.1.6. Querying Objects That Are Greater Than the Given Value

This is the associated JSON comparison expression: *json-pointer* gt *json-value*

Review the following example:

```
"_queryFilter" : '/employeeNumber gt 5000'
```

The following REST call returns the user names of all managed users whose employeeNumber is higher than 5000:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=employeeNumber+gt+5000&_fields=userName
,employeeNumber'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 1458,
  "result": [
    {
      "employeeNumber": 5003,
      "userName": "agilder"
    },
    {
      "employeeNumber": 5011,
      "userName": "bsmith"
    },
    {
      "employeeNumber": 5034,
      "userName": "bjensen"
    },
    {
      "employeeNumber": 5027,
      "userName": "cclarke"
    },
    {
      "employeeNumber": 5033,
      "userName": "scarter"
    }
...
  ]
}
```

8.3.4.1.7. Querying Objects That Are Greater Than or Equal to the Given Value

This is the associated JSON comparison expression: *json-pointer* ge *json-value*.

Review the following example:

```
"_queryFilter" : '/employeeNumber ge 5000'
```

The following REST call returns the user names of all managed users whose `employeeNumber` is 5000 or greater:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=employeeNumber+ge+5000&_fields=userName
,employeeNumber'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 1457,
  "result": [
    {
      "employeeNumber": 5000,
      "userName": "agilder"
    },
    {
      "employeeNumber": 5011,
      "userName": "bsmith"
    },
    {
      "employeeNumber": 5034,
      "userName": "bjensen"
    },
    {
      "employeeNumber": 5027,
      "userName": "cclarke"
    },
    {
      "employeeNumber": 5033,
      "userName": "scarter"
    }
...
  ]
}
```

8.3.4.2. Presence Expressions

The following examples show how you can build filters using a presence expression, shown as `pr`. The presence expression is a filter that returns all records with a given attribute.

A presence expression filter evaluates to `true` when a *`json-pointer`* `pr` matches any object in which the *json-pointer* is present, and contains a non-null value. Review the following expression:

```
"_queryFilter" : '/mail pr'
```

The following REST call uses that expression to return the mail addresses for all managed users with a `mail` property:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=mail+pr&_fields=mail'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 2,
  "result": [
    {
      "mail": "jdoe@exampleAD.com"
    },
    {
      "mail": "bjensen@example.com"
    }
  ]
}
```

You can also apply the presence filter on system objects. For example, the following query returns the `uid` of all users in an LDAP system who have the `uid` attribute in their entries:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/system/ldap/account?_queryFilter=uid+pr&_fields=uid'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 2,
  "result": [
    {
      "uid": "jdoe"
    },
    {
      "uid": "bjensen"
    }
  ]
}
```

8.3.4.3. Literal Expressions

A literal expression is a boolean:

- `true` matches any object in the resource.
- `false` matches no object in the resource.

For example, you can list the `_id` of all managed objects as follows:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user?_queryFilter=true&_fields=_id'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 2,
  "result": [
    {
      "_id": "d2e29d5f-0d74-4d04-bcfe-b1daf508ad7c"
    },
    {
      "_id": "709fed03-897b-4ff0-8a59-6faaa34e3af6"
    }
  ]
}
```

8.3.4.4. Complex Expressions

You can combine expressions using the boolean operators `and`, `or`, and `!` (not). The following example queries managed user objects located in London, with last name Jensen:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/managed/user/?_queryFilter=city+eq+"London"+and+sn+eq
+"Jensen"&_fields=userName,givenName,sn'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 3,
  "result": [
    {
      "sn": "Jensen",
      "givenName": "Clive",
      "userName": "cjensen"
    },
    {
      "sn": "Jensen",
      "givenName": "Dave",
      "userName": "djensen"
    },
    {
      "sn": "Jensen",
      "givenName": "Margaret",
      "userName": "mjensen"
    }
  ]
}
```

8.3.5. Paging and Counting Query Results

The common filter query mechanism supports paged query results for managed objects, and for some system objects, depending on the system resource.

Predefined queries must be configured to support paging, in the repository configuration. For example:

```
"query-all-ids" : "select _openidm_id from ${unquoted:_resource} SKIP ${unquoted:_pagedResultsOffset}
        LIMIT ${unquoted:_pageSize}",
```

The query implementation includes a configurable count policy that can be set per query. Currently, counting results is supported only for predefined queries, not for filtered queries.

The count policy can be one of the following:

- `NONE` - to disable counting entirely for that query.
- `EXACT` - to return the precise number of query results. Note that this has a negative impact on query performance.
- `ESTIMATE` - to return a best estimate of the number of query results in the shortest possible time. This number generally correlates with the number of records in the index.

If no count policy is specified, the policy is assumed to be `NONE`. This prevents the overhead of counting results, unless a result count is specifically required.

The following query returns the first three records in the managed user repository:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user?_queryId=query-all-ids&_pageSize=3"
{
  "result": [
    {
      "_id": "scarter",
      "_rev": "1"
    },
    {
      "_id": "bjensen",
      "_rev": "1"
    },
    {
      "_id": "asmith",
      "_rev": "1"
    }
  ],
  "resultCount": 3,
  "pagedResultsCookie": "3",
  "totalPagedResultsPolicy": "NONE",
  "totalPagedResults": -1,
  "remainingPagedResults": -1
}
```

Notice that no counting is done in this query, so the returned value the of `"totalPagedResults"` and `"remainingPagedResults"` fields is `-1`.

To specify that either an `EXACT` or `ESTIMATE` result count be applied, add the `"totalPagedResultsPolicy"` to the query.

The following query is identical to the previous query but includes a count of the total results in the result set.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user?_queryId=query-all-
ids&_pageSize=3&_totalPagedResultsPolicy=EXACT"
{
  "result": [
    {
      "_id": "scarter",
      "_rev": "1"
    },
    {
      "_id": "bjensen",
      "_rev": "1"
    },
    {
      "_id": "asmith",
      "_rev": "1"
    }
  ],
  "resultCount": 3,
  "pagedResultsCookie": "3",
  "totalPagedResultsPolicy": "EXACT",
  "totalPagedResults": 4,
  "remainingPagedResults": -1
}
```

Note that the `totalPagedResultsPolicy` is `EXACT` for this query. To return an exact result count, a corresponding `count` query must be defined in the repository configuration. The following excerpt of the default `repo.orientdb.json` file shows the predefined `query-all-ids` query, and its corresponding `count` query:

```
"query-all-ids" : "select _openidm_id, @version from ${unquoted:_resource}
      SKIP ${unquoted:_pagedResultsOffset} LIMIT ${unquoted:_pageSize}",
"query-all-ids-count" : "select count(_openidm_id) AS total from ${unquoted:_resource}",
```

The following paging parameters are supported:

`_pagedResultsCookie`

Opaque cookie used by the server to keep track of the position in the search results. The format of the cookie is a string value.

The server provides the cookie value on the first request. You should then supply the cookie value in subsequent requests until the server returns a null cookie, meaning that the final page of results has been returned.

Paged results are enabled only if the `_pageSize` is a non-zero integer.

`_pagedResultsOffset`

Specifies the index within the result set of the number of records to be skipped before the first result is returned. The format of the `_pagedResultsOffset` is an integer value. When the value of `_pagedResultsOffset` is greater than or equal to 1, the server returns pages, starting after the specified index.

This request assumes that the `_pageSize` is set, and not equal to zero.

For example, if the result set includes 10 records, the `_pageSize` is 2, and the `_pagedResultsOffset` is 6, the server skips the first 6 records, then returns 2 records, 7 and 8. The `_pagedResultsCookie` value would then be 8 (the index of the last returned record) and the `_remainingPagedResults` value would be 2, the last two records (9 and 10) that have not yet been returned.

If the offset points to a page beyond the last of the search results, the result set returned is empty.

Note that the `totalPagedResults` and `_remainingPagedResults` parameters are not supported for all queries. Where they are not supported, their returned value is always `-1`.

`_pageSize`

An optional parameter indicating that query results should be returned in pages of the specified size. For all paged result requests other than the initial request, a cookie should be provided with the query request.

The default behavior is not to return paged query results. If set, this parameter should be an integer value, greater than zero.

8.3.6. Sorting Query Results

For common filter query expressions, you can sort the results of a query using the `_sortKeys` parameter. This parameter takes a comma-separated list as a value and orders the way in which the JSON result is returned, based on this list.

The `_sortKeys` parameter is not supported for predefined queries.

The following query returns all users with the `givenName` `Dan`, and sorts the results alphabetically, according to surname (`sn`):

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/system/ldap/account?_queryFilter=givenName+eq+"Dan"&_fields=givenName
,sn&_sortKeys=sn'
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 3,
  "result": [
    {
      "sn": "Cope",
      "givenName": "Dan"
    },
    {
      "sn": "Langdon",
      "givenName": "Dan"
    },
    {
      "sn": "Lanoway",
      "givenName": "Dan"
    }
  ]
}
```

Chapter 9
Managing Users, Groups, Roles and Relationships

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

Managed objects and their properties are defined in your project's `conf/managed.json` file. Note that the schema defined in this file is not a comprehensive list of all the properties that can be stored in the managed object repository. If you use a generic object mapping, you can create a managed object with any arbitrary property, and that property will be stored in the repository. For more information about explicit and generic object mappings, see "Using Explicit or Generic Object Mapping With a JDBC Repository".

This chapter describes how to work with the default managed object types and how to create new object types as required by your deployment. For more information about the OpenIDM object model, see "*Data Models and Objects Reference*".

9.1. Creating and Modifying Managed Object Types

If the managed object types provided in the default configuration are not sufficient for your deployment, you can create any number of new managed object types.

The easiest way to create a new managed object type is to use the Admin UI, as follows:

1. Navigate to the Admin UI URL (`https://localhost:8443/admin`) then select Configure > Managed Objects > New Managed Object.

2. Enter a name for the new managed object and, optionally, an icon that will be displayed for that object type in the UI.

   Click Save.

3. Select the Scripts tab and specify any scripts that should be applied on various events associated with that object type, for example, when an object of that type is created, updated or deleted.

4. Specify the schema for the object type, that is, the properties that make up the object, and any policies or restrictions that must be applied to the property values.

You can also create a new managed object type by adding its configuration, in JSON, to your project's `conf/managed.json` file. The following excerpt of the `managed.json` file shows the configuration of a "Phone" object, that was created through the UI.

```
{
    "name": "Phone",
    "schema": {
        "$schema": "http://forgerock.org/json-schema#",
        "type": "object",
        "properties": {
            "brand": {
                "description": "The supplier of the mobile phone",
                "title": "Brand",
                "viewable": true,
                "searchable": true,
                "userEditable": false,
                "policies": [],
                "returnByDefault": false,
                "minLength": "",
                "pattern": "",
                "isVirtual": false,
                "type": [
                    "string",
                    "null"
                ]
            },
            "assetNumber": {
                "description": "The asset tag number of the mobile device",
                "title": "Asset Number",
                "viewable": true,
                "searchable": true,
                "userEditable": false,
                "policies": [],
                "returnByDefault": false,
                "minLength": "",
                "pattern": "",
                "isVirtual": false,
                "type": "string"
            },
            "model": {
                "description": "The model number of the mobile device, such as 6 plus, Galaxy S4",
                "title": "Model",
                "viewable": true,
                "searchable": false,
                "userEditable": false,
                "policies": [],
                "returnByDefault": false,
                "minLength": "",
                "pattern": "",
                "isVirtual": false,
                "type": "string"
            }
        },
        "required": [],
        "order": [
            "brand",
            "assetNumber",
            "model"
        ]
    }
}
```

You can add any arbitrary properties to the schema of a new managed object type. A property definition typically includes the following fields:

`name`

The name of the property.

`title`

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

`description`

A brief description of the property.

`viewable`

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

`searchable`

Specifies whether this property can be searched in the UI. A searchable property is visible within the Managed Object data grid in the Self-Service UI. Note that for a property to be searchable in the UI, it *must be indexed* in the repository configuration. For information on indexing properties in a repository, see "Using Explicit or Generic Object Mapping With a JDBC Repository".

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

`userEditable`

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

`isProtected`

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

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

`minLength`

The minimum number of characters that the value of this property must have.

`pattern`

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

`policies`

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

`required`

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

`type`

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

> **Note**
>
> If a property (such as a `telephoneNumber`) might not exist for a particular user, you must include `null` as one of the property `type`s. You can set a null property type in the Admin UI (Configure > Managed Objects > User > Schema then select the property and set `Nullable` to `true`). You can also set a null property type directly in your `managed.json` file by setting `"type" : '[ "string","null" ]'` for that property (where `string` can be any other valid property type. This information is validated by the `policy.js` script, as described in "Validation of Managed Object Data Types".
>
> If you're configuring a data `type` of `array` through the Admin UI, you're limited to two values.

`isVirtual`

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

`returnByDefault`

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

9.2. Working with Managed Users

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

OpenIDM provides RESTful access to managed users, at the context path /openidm/managed/user. For more information, see "Getting Started With the REST Interface" in the *Installation Guide*.

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

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

The following example retrieves the JSON representation of all managed users in the repository:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
"http://localhost:8080/openidm/managed/user?_queryId=query-all-ids"
```

The following two examples query all managed users for a user named scarter:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
 "http://localhost:8080/openidm/managed/user?_queryFilter=userName+eq+%22scarter%22"
```

In this second example, note the use of single quotes around the URL, to avoid conflicts with the double quotes around the user named smith. Note also that the _queryFilter requires double quotes (or the URL encoded equivalent, %22,) around the search term:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
'http://localhost:8080/openidm/managed/user?_queryFilter=userName+eq+"scarter"'
```

The following example retrieves the JSON representation of a managed user, specified by his ID, scarter:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user/scarter"
```

The following example adds a user with a specific user ID, `bjensen`:

```
$ curl \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "If-None-Match: *" \
 --request PUT \
 --data '{
    "userName":"bjensen",
    "sn":"Jensen",
    "givenName":"Barbara",
    "mail": "bjensen@example.com",
    "telephoneNumber": "082082082",
    "password":"Passw0rd"
  }' \
"http://localhost:8080/openidm/managed/user/bjensen"
```

The following example adds the same user, but allows OpenIDM to generate an ID. Creating objects with system-generated IDs is recommended in production environments:

```
$ curl \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 --data '{
    "userName":"bjensen",
    "sn":"Jensen",
    "givenName":"Barbara",
    "mail": "bjensen@example.com",
    "telephoneNumber": "082082082",
    "password":"Passw0rd"
  }' \
"http://localhost:8080/openidm/managed/user?_action=create"
```

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

```
$ curl \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 --data '[{
  "operation":"replace",
  "field":"/telephoneNumber",
  "value":"1234567"
  }]' \
  "http://localhost:8080/openidm/managed/user?_action=patch&_queryId=for-userName&uid=bjensen"
```

The following example deletes user `bjensen`:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request DELETE \
 "http://localhost:8080/openidm/managed/user/bjensen"
```

9.3. Working With Managed Groups

OpenIDM provides support for a managed `group` object. For a JDBC repository, OpenIDM stores managed groups with all other managed objects, in the `managedobjects` table, and uses the `managedobjectproperties` for indexing. For an OrientDB repository, managed groups are stored in the `managed_group` table.

The managed group object is not provided by default. To use managed groups, add an object similar to the following to your `conf/managed.json` file:

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

With this addition, OpenIDM provides RESTful access to managed groups, at the context path `/openidm/managed/group`.

For an example of a deployment that uses managed groups, see "Sample 2d - Synchronizing LDAP Groups" in the *Samples Guide*.

9.4. Working With Managed Roles

OpenIDM supports two types of roles:

- *Provisioning roles* - used to specify how objects are provisioned to an external system.
- *Authorization roles* - used to specify the authorization rights of a managed object internally, within OpenIDM.

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

Authorization roles can be created either as managed roles (at the context path `openidm/managed/role/role-name`) or as internal roles (at the context path `openidm/repo/internal/role/role-name`). Authorization roles are granted to managed users as values of the user's `authzRoles` property.

Both provisioning roles and authorization roles use the relationships mechanism to link the role to the managed object to which it applies. For more information about relationships between objects, see "Managing Relationships Between Objects".

This section describes how to create and use *managed roles*, either managed provisioning roles, or managed authorization roles. For more information about internal authorization roles, and how OpenIDM controls authorization to its own endpoints, see "Authorization".

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

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

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

```
"roles": [
    {
      "_ref": "managed/role/employee",
      "_refProperties": {
        "_id": "c090818d-57fd-435c-b1b1-bb23f47eaf09",
        "_rev": "1"
      }
    },
    {
      "_ref": "managed/role/supervisor",
      "_refProperties": {
        "_id": "4961912a-e2df-411a-8c0f-8e63b62dbef6",
        "_rev": "1"
      }
    }
  ]
```

Important

The `_ref` property points to the ID of the managed role that has been granted to the user. This particular example uses a client-assigned ID that is the same as the role name, to make the example easier to understand. All other examples in this chapter use system-assigned IDs. In production, you should use system-assigned IDs for role objects.

The following sections describe how to create, read, update, and delete managed roles, and how to grant roles to users. For information about how roles are used to provision users to external systems, see "Working With Role Assignments". For a sample that demonstrates the basic CRUD operations on roles, see "*Demonstrating the Roles Implementation*" in the *Samples Guide*.

9.4.1. Creating a Role

The easiest way to create a new role is by using the Admin UI. Select Manage > Role and click New Role on the Role List page. Enter a name and description for the new role and click Save.

Optionally, select Enable Condition to define a query filter that will allow this role to be granted to members dynamically. For more information, see "Granting Roles Dynamically".

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{
     "name" : "employee",
     "description" : "Role granted to workers on the company payroll"
 }' \
 "http://localhost:8080/openidm/managed/role?_action=create"
{
  "_id": "cedadaed-5774-4d65-b4a2-41d455ed524a",
  "_rev": "1",
  "name": "employee",
  "description": "Role granted to workers on the company payroll"
}
```

At this stage, the `employee` role has no corresponding *assignments*. Assignments are what enables the provisioning logic to the external system. Assignments are created and maintained as separate managed objects, and are referred to within role definitions. For more information about assignments, see "Working With Role Assignments".

9.4.2. Listing Existing Roles

You can display a list of all configured managed roles over REST or by using the Admin UI.

To list the managed roles in the Admin UI, select Manage > Role.

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/role?_queryFilter=true"
{
  "result": [
    {
      "_id": "cedadaed-5774-4d65-b4a2-41d455ed524a",
      "_rev": "1",
      "name": "employee",
      "description": "Role granted to workers on the company payroll"
    }
  ]
,
...
}
```

9.4.3. Granting a Role to a User

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

Roles can be granted manually or dynamically.

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

- Update the value of the user's `roles` property (if the role is a provisioning role) or `authzRoles` property (if the role is an authorization role) to reference the role.
- Update the value of the role's `members` property to reference the user.

Manual role grants are described further in "Granting Roles Manually".

Dynamic role grants use the result of a condition or script to update a user's list of roles. Dynamic role grants are described in detail in "Granting Roles Dynamically".

9.4.3.1. Granting Roles Manually

To grant a role to a user manually, use the Admin UI or the REST interface as follows:

Using the Admin UI

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

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

Over the REST interface

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

- Update the user to refer to the role.

The following sample command grants the `employee` role (with ID `cedadaed-5774-4d65-b4a2-41d455ed524a`) to user scarter:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PATCH \
 --data '[
    {
       "operation": "add",
       "field": "/roles/-",
       "value": {"_ref" : "managed/role/cedadaed-5774-4d65-b4a2-41d455ed524a"}
    }
 ]' \
 "http://localhost:8080/openidm/managed/user/scarter"
{
  "_id": "scarter",
  "_rev": "2",
  "mail": "scarter@example.com",
  "givenName": "Steven",
  "sn": "Carter",
  "description": "Created By XML1",
  "userName": "scarter@example.com",
  "telephoneNumber": "1234567",
  "accountStatus": "active",
  "effectiveRoles": [
    {
      "_ref": "managed/role/cedadaed-5774-4d65-b4a2-41d455ed524a"
    }
  ],
  "effectiveAssignments": []
}
```

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

- Update the role to refer to the user.

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PATCH \
 --data '[
    {
       "operation": "add",
       "field": "/members/-",
       "value": {"_ref" : "managed/user/scarter"}
    }
 ]' \
 "http://localhost:8080/openidm/managed/role/cedadaed-5774-4d65-b4a2-41d455ed524a"
{
  "_id": "cedadaed-5774-4d65-b4a2-41d455ed524a",
  "_rev": "2",
  "name": "employee",
  "description": "Role granted to workers on the company payroll"
}
```

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

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/managed/role/cedadaed-5774-4d65-b4a2-41d455ed524a?_fields=*_ref
,name"
 {
  "_id": "cedadaed-5774-4d65-b4a2-41d455ed524a",
  "_rev": "1",
  "name": "employee",
  "members": [
    {
      "_ref": "managed/user/scarter",
      "_refProperties": {
        "_id": "98d22d75-7090-47f8-9608-01ff92b447a4",
        "_rev": "1"
      }
    }
  ],
  "authzMembers": [],
  "assignments": []
}
```

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

  The following command replaces scarter's entire `roles` entry (that is, overwrites any existing roles) with a single entry, the reference to the `employee` role (ID `cedadaed-5774-4d65-b4a2-41d455ed524a`):

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PATCH \
 --data '[
   {
     "operation": "replace",
     "field":"/roles",
     "value":[
          {"_ref":"managed/role/cedadaed-5774-4d65-b4a2-41d455ed524a"}
     ]
   }
 ]' \
 "http://localhost:8080/openidm/managed/user/scarter"
```

9.4.3.2. Granting Roles Dynamically

The previous section showed how to grant roles to a user manually, by listing a reference to the role as a value of the user's `roles` attribute. OpenIDM also supports the following methods of granting a role *dynamically*:

- Granting a role based on a condition, where that condition is expressed in a query filter in the role definition. If the condition is `true` for a particular member, that member is granted the role.

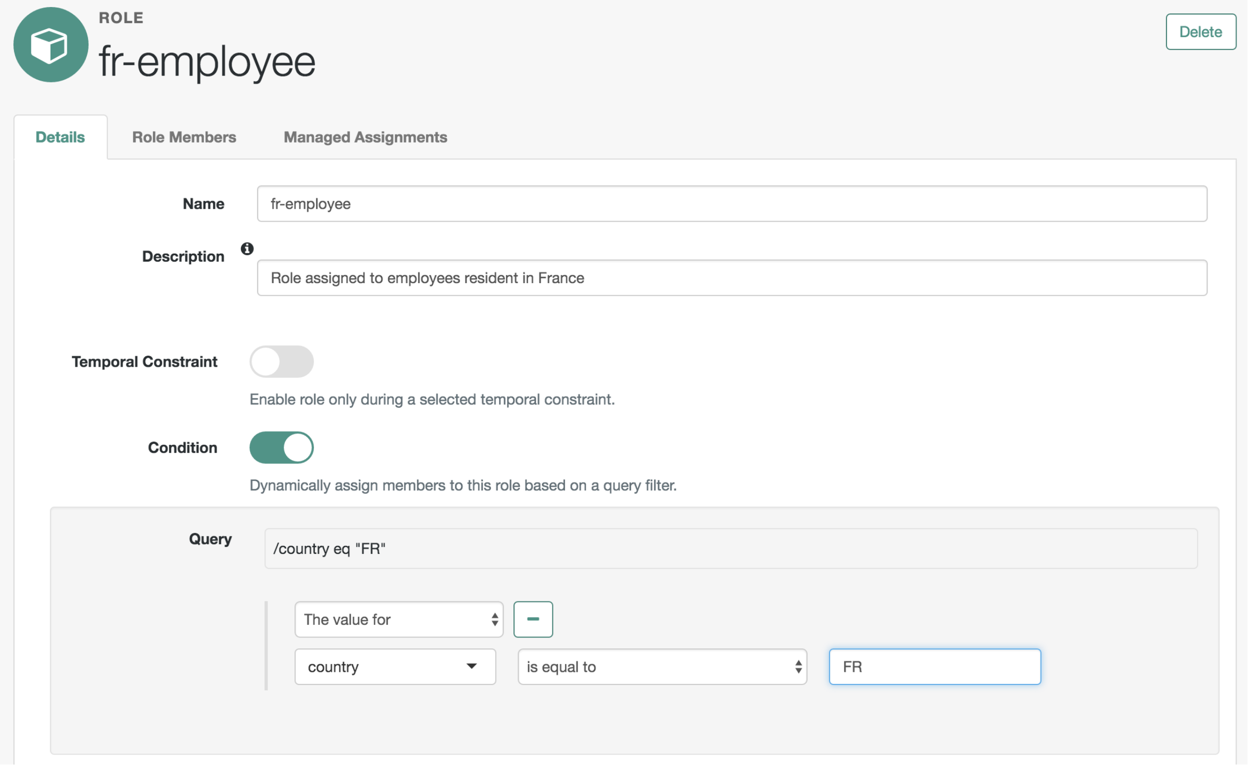
- Using a custom script to define a more complex role granting strategy.

9.4.3.2.1. Granting Roles Based on a Condition

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

To create a conditional role by using the Admin UI, select Condition on the role Details page, then define the query filter that will be used to assess the condition. In the following example, the role `fr-employee` will be granted only to those users who live in France (whose `country` property is set to `FR`):

Granting a Conditional Role, Based On a Query



ROLE
fr-employee

Details | Role Members | Managed Assignments

Name: fr-employee

Description: Role assigned to employees resident in France

Temporal Constraint: off
Enable role only during a selected temporal constraint.

Condition: on
Dynamically assign members to this role based on a query filter.

Query: /country eq "FR"

The value for
country — is equal to — FR

To create a conditional role over REST, include the query filter as a value of the `condition` property in the role definition. The following command creates a role similar to the one created in the previous screen shot:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{
    "name": "fr-employee",
    "description": "Role granted to employees resident in France",
    "condition": "/country eq \"FR\""
 }' \
 "http://localhost:8080/openidm/managed/role?_action=create"
 {
  "_id": "4b0a3e42-e5be-461b-a995-3e66c74551c1",
  "_rev": "1",
  "name": "fr-employee",
  "description": "Role granted to employees resident in France",
  "condition": "/country eq \"FR\""
}
```

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

Caution

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

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

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

9.4.3.2.2. Granting Roles By Using Custom Scripts

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

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

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

The new script will override the default effective roles script.

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

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

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

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

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

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

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

9.4.4. Using Temporal Constraints to Restrict Effective Roles

To restrict the period during which a role is effective, you can set a temporal constraint on the role itself, or on the role grant. A temporal constraint that is set on a role definition applies to all grants of that role. A temporal constraint that is set on a role grant enables you to specify the period that the role is valid *per user*.

For example, you might want a role definition such as `contractors-2016` to apply to all contract employees *only* for the year 2016. Or you might want a `contractors` role to apply to an individual user only for the period of his contract of employment.

The following sections describe how to set temporal constraints on role definitions, and on individual role grants.

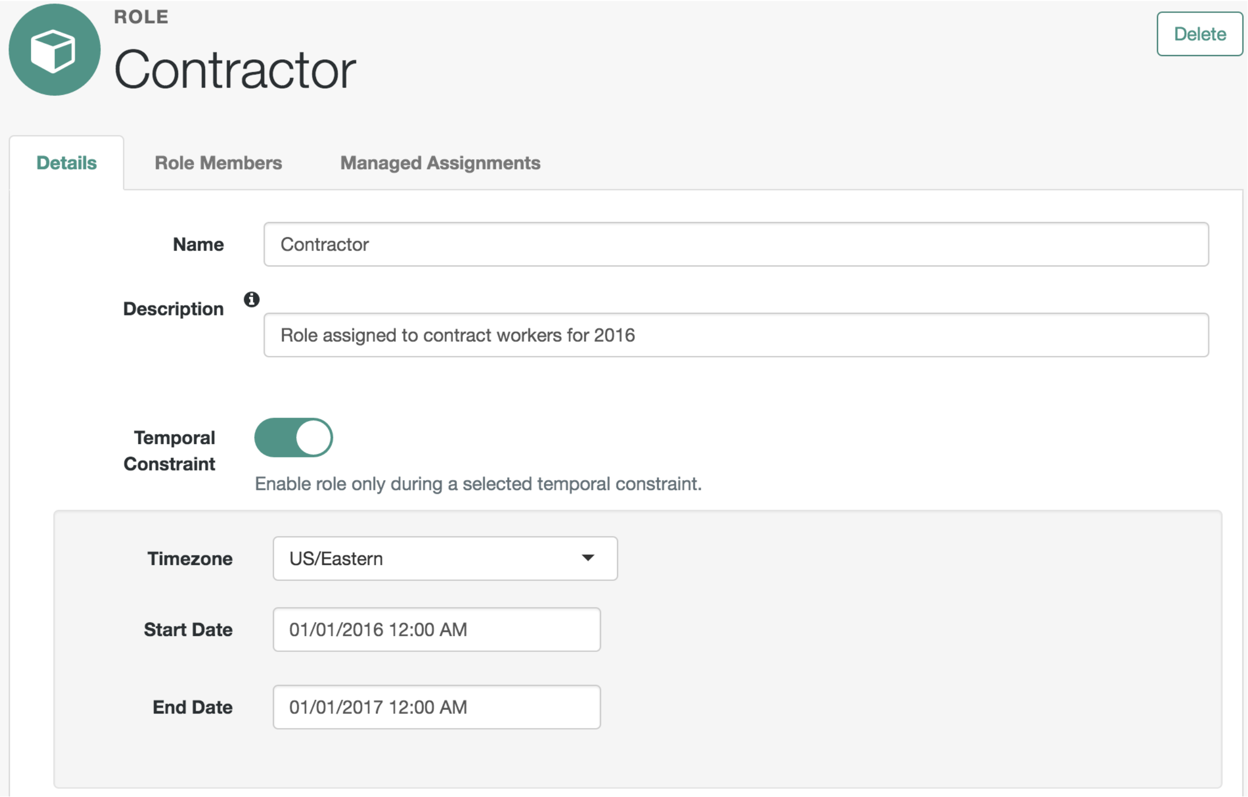
9.4.4.1. Adding a Temporal Constraint to a Role Definition

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

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

In the following example, the `Contractor` role is effective from January 1st, 2016 to January 1st, 2017:

Restricting a Role's Effectiveness to a Specified Time Period



ROLE
Contractor

Name: Contractor

Description: Role assigned to contract workers for 2016

Temporal Constraint: Enable role only during a selected temporal constraint.

Timezone: US/Eastern

Start Date: 01/01/2016 12:00 AM

End Date: 01/01/2017 12:00 AM

The following example adds a similar `contractor` role, over the REST interface:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{
     "name" : "contractor",
     "description" : "Role granted to contract workers for 2016",
     "temporalConstraints" : [
        {
            "duration" :  "2016-01-01T00:00:00.000Z/2017-01-01T00:00:00.000Z"
        }
     ]
 }' \
 "http://localhost:8080/openidm/managed/role?_action=create"
{
  "_id": "071283a8-0237-40a2-a31e-ceaa4d93c93d",
  "_rev": "1",
  "name": "contractor",
  "description": "Role granted to contract workers for 2016",
  "temporalConstraints": [
    {
      "duration": "2016-01-01T00:00:00.000Z/2017-01-01T00:00:00.000Z"
    }
  ]
}
```

The preceding example specifies the time zone as Coordinated Universal Time (UTC) by appending `Z` to the time. If no time zone information is provided, the time zone is assumed to be local time. To specify a different time zone, include an offset (from UTC) in the format `±hh:mm`. For example, an interval of `2016-01-01T00:00:00.000+04:00/2017-01-01T00:00:00.000+04:00` specifies a time zone that is four hours ahead of UTC.

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

The following example assumes that user scarter has been granted a role `contractor-april`. A temporal constraint has been included in the `contractor-april` definition that specifies that the role should be applicable only during the month of April 2016. At the end of this period, a query on scarter's entry shows that his `roles` property still includes the `contractor-april` role (with ID `3eb67be6-205b-483d-b36d-562b43a04ff8`), but his `effectiveRoles` property does not:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user/scarter?_fields=_id,userName,roles,effectiveRoles"
{
  "_id": "scarter",
  "_rev": "1",
  "userName": "scarter@example.com",
  "roles": [
    {
      "_ref": "managed/role/3eb67be6-205b-483d-b36d-562b43a04ff8",
      "_refProperties": {
        "temporalConstraints": [],
        "_grantType": "",
        "_id": "257099f5-56e5-4ce0-8580-f0f4d4b93d93",
        "_rev": "1"
      }
    }
  ],
  "effectiveRoles": []
}
```

In other words, the role is still in place but is no longer effective.

9.4.4.2. Adding a Temporal Constraint to a Role Grant

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

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

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

To apply a temporal constraint to a grant over the REST interface, include the constraint as one of the `_refProperties` of the relationship between the user and the role. The following example assumes a `contractor` role, with ID `9321fd67-30d1-4104-934d-cfd0a22e8182`. The command adds user bjensen as a member of that role, with a temporal constraint that specifies that she be a member of the role only for one year, from January 1st, 2016 to January 1st, 2017:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PATCH \
 --data '[
    {
     "operation": "add",
     "field": "/members/-",
     "value": {
      "_ref" : "managed/user/bjensen",
      "_refProperties": {
       "temporalConstraints": [{"duration": "2016-01-01T00:00:00.000Z/2017-01-01T00:00:00.000Z"}]
      }
     }
    }
 ]' \
 "http://localhost:8080/openidm/managed/role/9321fd67-30d1-4104-934d-cfd0a22e8182"
{
  "_id": "9321fd67-30d1-4104-934d-cfd0a22e8182",
  "_rev": "2",
  "name": "contractor",
  "description": "Role for contract workers"
}
```

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user/bjensen/roles?_queryFilter=true"
{
  "result": [
    {
      "_ref": "managed/role/9321fd67-30d1-4104-934d-cfd0a22e8182",
      "_refProperties": {
        "temporalConstraints": [
          {
            "duration": "2016-01-01T00:00:00.000Z/2017-01-01T00:00:00.000Z"
          }
        ],
        "_id": "84f5342c-cebe-4f0b-96c9-0267bf68a095",
        "_rev": "1"
      }
    }
  ]
,
...
}
```

9.4.5. Querying a User's Manual and Conditional Roles

The easiest way to check what roles have been granted to a user, either manually, or as the result of a condition, is to look at the user's entry in the Admin UI. Select Manage > User, click on the user whose roles you want to see, and select the Provisioning Roles tab.

To obtain a similar list over the REST interface, you can query the user's `roles` property. The following sample query shows that scarter has been granted two roles - an `employee` role (with ID `6bf4701a-7579-43c4-8bb4-7fd6cac552a1`) and an `fr-employee` role (with ID `00561df0-1e7d-4c8a-9c1e-3b1096116903`). specifies :

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user/scarter/roles?_queryFilter=true&_fields=_ref,_refProperties
,name"
{
  "result": [
    {
      "_ref": "managed/role/6bf4701a-7579-43c4-8bb4-7fd6cac552a1",
      "_refProperties": {
        "temporalConstraints": [],
        "_grantType": "",
        "_id": "8417106e-c3ef-4f59-a482-4c92dbf00308",
        "_rev": "2"
      },
      "name": "employee"
    },
    {
      "_ref": "managed/role/00561df0-1e7d-4c8a-9c1e-3b1096116903",
      "_refProperties": {
        "_grantType": "conditional",
        "_id": "e59ce7c3-46ce-492a-ba01-be27af731435",
        "_rev": "1"
      },
      "name": "fr-employee"
    }
  ],
 ...
}
```

Note that the `fr-employee` role has an additional reference property, `_grantType`. This property indicates *how* the role was granted to the user. If there is no `_grantType`, the role was granted manually.

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

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

9.4.6. Deleting a User's Roles

Roles that have been granted manually can be removed from a user's entry in two ways:

- Update the value of the user's `roles` property (if the role is a provisioning role) or `authzRoles` property (if the role is an authorization role) to remove the reference to the role.
- Update the value of the role's `members` property to remove the reference to that user.

Both of these actions can be achieved by using the Admin UI, or over REST.

Using the Admin UI

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

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

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

- Select Manage > Role and click on the role whose members you want to remove.

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

Over the REST interface

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

- Delete the role from the user's `roles` property, including the reference ID (the ID of the relationship between the user and the role) in the delete request:

  The following sample command removes the `employee` role (with ID `6bf4701a-7579-43c4-8bb4-7fd6cac552a1`) from user scarter:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request DELETE \
 "http://localhost:8080/openidm/managed/user/scarter/roles/8417106e-c3ef-4f59-a482-4c92dbf00308"
{
  "_ref": "managed/role/6bf4701a-7579-43c4-8bb4-7fd6cac552a1",
  "_refProperties": {
    "temporalConstraints": [],
    "_grantType": "",
    "_id": "8417106e-c3ef-4f59-a482-4c92dbf00308",
    "_rev": "2"
  }
}
```

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

Caution

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

```
$ curl \
 --header "Content-type: application/json" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request PATCH \
 --data '[
    {
      "operation" : "remove",
      "field" : "/roles",
      "value" :     {
       "_ref": "managed/role/6bf4701a-7579-43c4-8bb4-7fd6cac552a1",
       "_refProperties": {
         "temporalConstraints": [],
         "_grantType": "",
         "_id": "8417106e-c3ef-4f59-a482-4c92dbf00308",
         "_rev": "1"
       }
     }
    }
  ]' \
 "http://localhost:8080/openidm/managed/user/scarter"
{
  "_id": "scarter",
  "_rev": "3",
  "mail": "scarter@example.com",
  "givenName": "Steven",
  "sn": "Carter",
  "description": "Created By XML1",
  "userName": "scarter@example.com",
  "telephoneNumber": "1234567",
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

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

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

```
$ url \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/role/6bf4701a-7579-43c4-8bb4-7fd6cac552a1/members?
_queryFilter=true"
{
  "result": [
    {
      "_ref": "managed/user/bjensen",
      "_refProperties": {
        "temporalConstraints": [],
        "_grantType": "",
        "_id": "3c047f39-a9a3-4030-8d0c-bcd1fadb1d3d",
        "_rev": "3"
      }
    }
  ]
,
...
}
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request DELETE \
 "http://localhost:8080/openidm/managed/role/6bf4701a-7579-43c4-8bb4-7fd6cac552a1/members/3c047f39-a9a3
-4030-8d0c-bcd1fadb1d3d"
{
  "_ref": "managed/user/bjensen",
  "_refProperties": {
    "temporalConstraints": [],
    "_grantType": "",
    "_id": "3c047f39-a9a3-4030-8d0c-bcd1fadb1d3d",
    "_rev": "3"
  }
}
```

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

9.4.7. Deleting a Role Definition

You can delete a managed provisioning or authorization role by using the Admin UI, or over the REST interface.

To delete a role by using the Admin UI, select Manage > Role, select the role you want to remove, and click Delete.

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request DELETE \
 "http://localhost:8080/openidm/managed/role/6bf4701a-7579-43c4-8bb4-7fd6cac552a1"
{
  "_id": "6bf4701a-7579-43c4-8bb4-7fd6cac552a1",
  "_rev": "1",
  "name": "employee",
  "description": "Role granted to workers on the company payroll"
}
```

Note

You cannot delete a role if it is currently granted to one or more users. If you attempt to delete a role that is granted to a user (either over the REST interface, or by using the Admin UI), OpenIDM returns an error. The following command indicates an attempt to remove the `employee` role while it is still granted to user scarter:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request DELETE \
 "http://localhost:8080/openidm/managed/role/6bf4701a-7579-43c4-8bb4-7fd6cac552a1"
{
    "code":409,
    "reason":"Conflict",
    "message":"Cannot delete a role that is currently granted"
 }
```

9.4.8. Working With Role Assignments

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

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

Assignment objects are created, updated and deleted like any other managed object, and are attached to a role by using the relationships mechanism, in much the same way as a role is granted to a user. Assignment are stored in the repository and are accessible at the context path `/openidm/managed/assignment`.

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

9.4.8.1. Creating an Assignment

The easiest way to create an assignment is by using the Admin UI, as follows:

1. Select Manage > Assignment and click New Assignment on the Assignment List page.
2. Enter a name and description for the new assignment, and select the mapping to which the assignment should apply. The mapping indicates the target resource, that is, the resource on which the attributes specified in the assignment will be adjusted.
3. Click Add Assignment.
4. Select the Attributes tab and select the attribute or attributes whose values will be adjusted by this assignment.
    - If a regular text field appears, specify what the value of the attribute should be, when this assignment is applied.
    - If an Item button appears, you can specify a managed object type, such as an object, relationship, or string.
    - If a Properties button appears, you can specify additional information such as an array of role references, as described in "Working With Managed Roles".
5. Select the assignment operation from the dropdown list:
    - `Merge With Target` - the attribute value will be added to any existing values for that attribute. This operation merges the existing value of the target object attribute with the value(s) from the assignment. If duplicate values are found (for attributes that take a list as a value), each value is included only once in the resulting target. This assignment operation is used only with complex attribute values like arrays and objects, and does not work with strings or numbers. (Property: `mergeWithTarget`.)
    - `Replace Target` - the attribute value will overwrite any existing values for that attribute. The value from the assignment becomes the authoritative source for the attribute. (Property: `replaceTarget`.)

    Select the unassignment operation from the dropdown list. You can set the unassignment operation to one of the following:

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

6. Optionally, click the Events tab to specify any scriptable events associated with this assignment.

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

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

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

7. Click the Roles tab to attach this assignment to an existing role definition.

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

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

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{
   "name" : "employee",
   "description": "Assignment for employees.",
   "mapping" : "managedUser_systemLdapAccounts",
   "attributes": [
       {
           "name": "employeeType",
           "value": "Employee",
           "assignmentOperation" : "mergeWithTarget",
           "unassignmentOperation" : "removeFromTarget"
       }
   ]
 }' \
 "http://localhost:8080/openidm/managed/assignment?_action=create"
```

```
{
  "_id": "2fb3aa12-109f-431c-bdb7-e42213747700",
  "_rev": "1",
  "name": "employee",
  "description": "Assignment for employees.",
  "mapping": "managedUser_systemLdapAccounts",
  "attributes": [
    {
      "name": "employeeType",
      "value": "Employee",
      "assignmentOperation": "mergeWithTarget",
      "unassignmentOperation": "removeFromTarget"
    }
  ]
}
```

Note that at this stage, the assignment is not linked to any role, so no user can make use of the assignment. You must add the assignment to a role, as described in the following section.

9.4.8.2. Adding an Assignment to a Role

When you have created a managed role, and a managed assignment, you reference the assignment from the role, in much the same way as a user references a role.

You can update a role definition to include one or more assignments, either by using the Admin UI, or over the REST interface.

Using the Admin UI

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

Over the REST interface

Update the role definition to include a reference to the ID of the assignment in the `assignments` property of the role. The following sample command adds the `employee` assignment (with ID `2fb3aa12-109f-431c-bdb7-e42213747700`) to an existing `employee` role (whose ID is `59a8cc01-bac3-4bae-8012-f639d002ad8c`):

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PATCH \
 --data '[
   {
       "operation" : "add",
       "field" : "/assignments/-",
       "value" : { "_ref": "managed/assignment/2fb3aa12-109f-431c-bdb7-e42213747700" }
   }
 ]' \
 "http://localhost:8080/openidm/managed/role/59a8cc01-bac3-4bae-8012-f639d002ad8c"
{
  "_id": "59a8cc01-bac3-4bae-8012-f639d002ad8c",
  "_rev": "3",
  "name": "employee",
  "description": "Role granted to workers on the company payroll"
}
```

To check that the assignment was added successfully, you can query the `assignments` property of the role:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/role/59a8cc01-bac3-4bae-8012-f639d002ad8c/assignments?
_queryFilter=true&_fields=_ref,_refProperties,name"

{
  "result": [
    {
      "_ref": "managed/assignment/2fb3aa12-109f-431c-bdb7-e42213747700",
      "_refProperties": {
        "_id": "686b328a-e2bd-4e48-be25-4a4e12f3b431",
        "_rev": "4"
      },
      "name": "employee"
    }
  ]
,
...
}
```

Note that the role's `assignments` property now references the assignment that you created in the previous step.

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

9.4.8.3. Deleting an Assignment

You can delete an assignment by using the Admin UI, or over the REST interface.

To delete an assignment by using the Admin UI, select Manage > Assignment, select the assignment you want to remove, and click Delete.

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request DELETE \
 "http://localhost:8080/openidm/managed/assignment/2fb3aa12-109f-431c-bdb7-e42213747700"
     {
  "_id": "2fb3aa12-109f-431c-bdb7-e42213747700",
  "_rev": "1",
  "name": "employee",
  "description": "Assignment for employees.",
  "mapping": "managedUser_systemLdapAccounts",
  "attributes": [
    {
      "name": "employeeType",
      "value": "Employee",
      "assignmentOperation": "mergeWithTarget",
      "unassignmentOperation": "removeFromTarget"
    }
  ]
}
```

Note

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

9.4.9. Understanding Effective Roles and Effective Assignments

Effective roles and *effective assignments* are virtual properties of a user object. Their values are calculated *on the fly* by the `openidm/bin/defaults/script/roles/effectiveRoles.js` and `openidm/bin/defaults/script/roles/effectiveAssignments.js` scripts. These scripts are triggered when a managed user is retrieved.

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

```
"effectiveRoles" : {
    "type" : "array",
    "title" : "Effective Roles",
    "viewable" : false,
    "returnByDefault" : true,
    "isVirtual" : true,
    "onRetrieve" : {
        "type" : "text/javascript",
         "source" : "require('roles/effectiveRoles').calculateEffectiveRoles(object, 'roles');"
    },
    "items" : {
        "type" : "object"
    }
},
"effectiveAssignments" : {
    "type" : "array",
    "title" : "Effective Assignments",
    "viewable" : false,
    "returnByDefault" : true,
    "isVirtual" : true,
    "onRetrieve" : {
        "type" : "text/javascript",
        "file" : "roles/effectiveAssignments.js",
        "effectiveRolesPropName" : "effectiveRoles"
    },
    "items" : {
        "type" : "object"
    }
},
```

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

The `effectiveRoles.js` script uses the `roles` attribute of a user entry to calculate the grants (manual or conditional) that are currently in effect at the time of retrieval, based on temporal constraints or other custom scripted logic.

The `effectiveAssignments.js` script uses the virtual `effectiveRoles` attribute to calculate that user's effective assignments. The synchronization engine reads the calculated value of the `effectiveAssignments` attribute when it processes the user. The target system is updated according to the configured `assignmentOperation` for each assignment.

Do not change the default `effectiveRoles.js` and `effectiveAssignments.js` scripts. If you need to change the logic that calculates `effectiveRoles` and `effectiveAssignments`, create your own custom script and include a reference to it in your project's `conf/managed.json` file. For more information about using custom scripts, see "*Scripting Reference*".

When a user entry is retrieved, OpenIDM calculates the `effectiveRoles` and `effectiveAssignments` for that user based on the current value of the user's `roles` property, and on any roles that might be granted dynamically through a custom script. The previous set of examples showed the creation of a role `employee` that referenced an assignment `employee` and was granted to user bjensen. Querying that user entry would show the following effective roles and effective assignments:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin"  \
 --request GET \
 "http://localhost:8080/openidm/managed/user/bjensen?_fields=userName,roles,effectiveRoles
,effectiveAssignments"
{
  "_id": "bjensen",
  "_rev": "2",
  "userName": "bjensen@example.com",
  "roles": [
    {
      "_ref": "managed/role/59a8cc01-bac3-4bae-8012-f639d002ad8c",
      "_refProperties": {
        "temporalConstraints": [],
        "_grantType": "",
        "_id": "881f0b96-06e9-4af4-b86b-aba4ee15e4ef",
        "_rev": "2"
      }
    }
  ],
  "effectiveRoles": [
    {
      "_ref": "managed/role/59a8cc01-bac3-4bae-8012-f639d002ad8c"
    }
  ],
  "effectiveAssignments": [
    {
      "name": "employee",
      "description": "Assignment for employees.",
      "mapping": "managedUser_systemLdapAccounts",
      "attributes": [
        {
          "name": "employeeType",
          "value": "Employee",
          "assignmentOperation": "mergeWithTarget",
          "unassignmentOperation": "removeFromTarget"
        }
      ],
      "_id": "4606245c-9412-4f1f-af0c-2b06852dedb8",
      "_rev": "2"
    }
  ]
}
```

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

9.4.10. Managed Role Script Hooks

Like any other managed object, a role has script hooks that enable you to configure role behavior. The default role definition in `conf/managed.json` includes the following script hooks:

```
{
    "name" : "role",
    "onDelete" : {
        "type" : "text/javascript",
        "file" : "roles/onDelete-roles.js"
    },
    "onSync" : {
        "type" : "text/javascript",
        "source" : "require('roles/onSync-roles').syncUsersOfRoles(resourceName, oldObject, newObject,
 ['members']);"
    },
    "onCreate" : {
        "type" : "text/javascript",
        "source" : "require('roles/conditionalRoles').roleCreate(object);"
    },
    "onUpdate" : {
        "type" : "text/javascript",
        "source" : "require('roles/conditionalRoles').roleUpdate(oldObject, object);"
    },
    "postCreate" : {
        "type" : "text/javascript",
        "file" : "roles/postOperation-roles.js"
    },
    "postUpdate" : {
        "type" : "text/javascript",
        "file" : "roles/postOperation-roles.js"
    },
    "postDelete" : {
        "type" : "text/javascript",
        "file" : "roles/postOperation-roles.js"
    },
...
```

When a role is deleted, the `onDelete` script hook calls the `bin/default/script/roles/onDelete-roles.js` script.

When a role is synchronized, the `onSync` hook causes a synchronization operation on all managed objects that reference the role.

When a *conditional role* is created or updated, the `onCreate` and `onUpdate` script hooks force an update on all managed users affected by the conditional role.

Directly after a role is created, updated or deleted, the `postCreate`, `postUpdate`, and `postDelete` hooks call the `bin/default/script/roles/postOperation-roles.js` script. Depending on when this script is called, it either creates or removes the scheduled jobs required to manage temporal constraints on roles.

9.5. Managing Relationships Between Objects

OpenIDM enables you to define *relationships* between two managed objects. Managed roles are implemented using relationship objects, but you can create a variety of relationship objects, as required by your deployment.

9.5.1. Defining a Relationship Type

Relationships are defined in your project's managed object configuration file (`conf/managed.json`). By default, OpenIDM provides a relationship named `manager`, that enables you to configure a management relationship between two managed users. The `manager` relationship is a good example from which to understand how relationships work.

The default `manager` relationship is configured as follows:

```
"manager" : {
    "type" : "relationship",
    "returnByDefault" : false,
    "description" : "",
    "title" : "Manager",
    "viewable" : true,
    "searchable" : false,
    "properties" : {
        "_ref" : { "type" : "string" },
        "_refProperties": {
            "type": "object",
            "properties": {
                "_id": { "type": "string" }
            }
    }
},
```

All relationships have the following configurable properties:

`type` **(string)**

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

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

Specifies whether the relationship should be returned in the result of a read or search query on the managed object that has the relationship. If included in an array, always set this property to `false`. By default, relationships are not returned, unless explicitly requested.

`description` **(string, optional)**

An optional string that provides additional information about the relationship object.

`title` **(string)**

Used by the UI to refer to the relationship.

`viewable` **(boolean, `true, false`)**

Specifies whether the relationship is visible as a field in the UI. The default value is `true`.

`searchable` **(boolean, `true, false`)**

Specifies whether values of the relationship can be searched, in the UI. For example, if you set this property to `true` for the `manager` relationship, a user will be able to search for managed user entries using the `manager` field as a filter.

_ref (JSON object)

Specifies how the relationship between two managed objects is referenced.

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

_refProperties (JSON object)

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

9.5.2. Establishing a Relationship Between Two Objects

When you have defined a relationship *type*, (such as the `manager` relationship, described in the previous section), you can reference that relationship from a managed user, using the `_ref` property.

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "If-None-Match: *" \
 --header "Content-Type: application/json" \
 --request PUT \
 --data '{
    "sn":"Smith",
    "userName":"psmith",
    "givenName":"Patricia",
    "displayName":"Patti Smith",
    "description" : "psmith - new user",
    "mail" : "psmith@example.com",
    "phoneNumber" : "0831245986",
    "password" : "Passw0rd",
    "manager" : {"_ref" : "managed/user/bjensen"}
  }' \
"http://localhost:8080/openidm/managed/user/psmith"
{
  "_id": "psmith",
  "_rev": "1",
  "sn": "Smith",
  "userName": "psmith",
  "givenName": "Patricia",
  "displayName": "Patti Smith",
  "description": "psmith - new user",
  "mail": "psmith@example.com",
  "phoneNumber": "0831245986",
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

Note that the relationship information is not returned by default in the command-line output.

Any change to a relationship triggers a synchronization operation on any other managed objects that are referenced by the relationship. For example, OpenIDM maintains referential integrity by deleting the relationship reference, if the object referred to by that relationship is deleted. In our example, if bjensen's user entry is deleted, the corresponding reference in psmith's `manager` property is removed.

9.5.3. Validating Relationships Between Objects

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

When you create a new relationship type, validation is disabled by default as it entails a query to the relationship that can be expensive, if it is not required. To configure validation of a referenced relationship, set `"validate": true` in the object configuration (in `managed.json`). The `managed.json` files provided with OpenIDM enable validation for the following relationships:

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

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

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

9.5.4. Working With Bi-Directional Relationships

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

A reverse relationship is declared in the managed object configuration (`conf/managed.json`). Consider the following sample excerpt of the default managed object configuration:

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

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

In other words, you can list all users whose `manager` property is set to the currently queried user.

That reverse relationship uses a `resourceCollection` of managed users, as shown here:

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

In this case, users are listed with the noted fields. You can configure these relationships from the Admin UI. For an example of the process, see "Configure a Relationship From the User Managed Object".

9.5.5. Viewing Relationships Over REST

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

```
$ curl
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user/psmith?_fields=manager"
{
  "_id": "psmith",
  "_rev": "1",
  "manager": {
    "_ref": "managed/user/bjensen",
    "_refProperties": {
      "_id": "e15779ad-be54-4a1c-b643-133dd9bb2e99",
      "_rev": "1"
    }
  }
}
```

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

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

```
$ curl
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user/psmith?_fields=manager/mail,manager/phoneNumber"
{
  "_id": "psmith",
  "_rev": "1",
  "phoneNumber": "1234567",
  "manager": {
    "_ref": "managed/user/bjensen",
    "_refProperties": {
      "_id": "e15779ad-be54-4a1c-b643-133dd9bb2e99",
      "_rev": "1"
    },
    "mail": "bjensen@example.com",
    "phoneNumber": "1234567"
  }
}
```

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/managed/user/psmith?_fields=*_ref"
{
  "_id": "psmith",
  "_rev": "1",
  "roles": [],
  "authzRoles": [
    {
      "_ref": "repo/internal/role/openidm-authorized",
      "_refProperties": {
        "_id": "8e7b2c97-dfa8-4eec-a95b-b40b710d443d",
        "_rev": "1"
      }
    }
  ],
  "manager": {
    "_ref": "managed/user/bjensen",
    "_refProperties": {
      "_id": "3a246327-a972-4576-b6a6-7126df780029",
      "_rev": "1"
    }
  }
}
```

9.5.6. Viewing Relationships in Graph Form

OpenIDM provides a relationship graph widget that gives a visual display of the relationships between objects.

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

1. Log into the Admin UI.

2. Select Dashboards, and choose the dashboard to which you want to add the widget.

   For more information about managing dashboards in the UI, see "Creating and Modifying Dashboards".

3. Select Add Widgets. In the Add Widgets window, scroll to the Identity Relationships widget, and click Add.

4. Select Close to exit the Add Widgets window.

5. On the dashboard, scroll down to the Identity Relationships widget. Select the vertical ellipses > Settings to configure the widget.

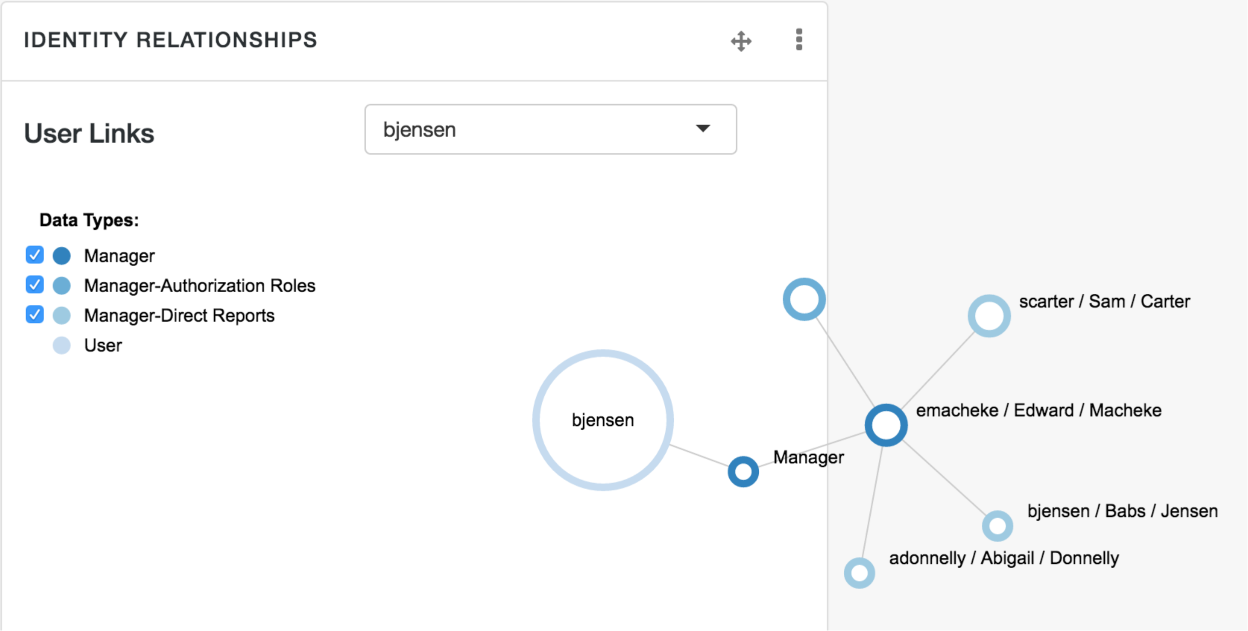
6. Choose the Widget Size, then enter the object for which you want to display relationships such as `user` and the search property for that object, such as `userName`.

If you want to include an additional level of relationships in the graph, select Display sub-relationships. In a traditional organization, this option will display a user's manager, along with all users with that same manager.

7. Click Save.

When you have configured the Identity Relationships widget, enter the user whose relationships you want to search.

The following graph shows all of bjensen's relationships. The graph shows bjensen's manager (emacheke) and all other users who are direct reports of emacheke.



IDENTITY RELATIONSHIPS

User Links: bjensen

Data Types:

☑ ● Manager
☑ ● Manager-Authorization Roles
☑ ● Manager-Direct Reports
● User

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

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

9.6. Running Scripts on Managed Objects

OpenIDM provides a number of *hooks* that enable you to manipulate managed objects using scripts. These scripts can be triggered during various stages of the lifecycle of the managed object, and are defined in the managed objects configuration file (`managed.json`).

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

In addition, OpenIDM supports the use of post-action scripts for managed objects, including after the creation of an object is complete (postCreate), after the update of an object is complete (postUpdate), and after the deletion of an object (postDelete).

The following sample extract of a `managed.json` file runs a script to calculate the effective assignments of a managed object, whenever that object is retrieved from the repository:

```
"effectiveAssignments" : {
    "type" : "array",
    "title" : "Effective Assignments",
    "viewable" : false,
    "returnByDefault" : true,
    "isVirtual" : true,
    "onRetrieve" : {
        "type" : "text/javascript",
        "file" : "roles/effectiveAssignments.js",
        "effectiveRolesPropName" : "effectiveRoles"
    },
    "items" : {
        "type" : "object"
    }
},
```

9.7. Encoding Attribute Values

OpenIDM supports two methods of encoding attribute values for managed objects - reversible encryption and the use of salted hashing algorithms. Attribute values that might be encoded include passwords, authentication questions, credit card numbers, and social security numbers. If passwords are already encoded on the external resource, they are generally excluded from the synchronization process. For more information, see "*Managing Passwords*".

You configure attribute value encoding, per schema property, in the managed object configuration (in your project's `conf/managed.json` file). The following sections show how to use reversible encryption and salted hash algorithms to encode attribute values.

9.7.1. Encoding Attribute Values With Reversible Encryption

The following excerpt of a `managed.json` file shows a managed object configuration that encrypts and decrypts the `password` attribute using the default symmetric key:

```
{
    "objects" : [
        {
            "name" : "user",
            ...
            "schema" : {
                ...
                "properties" : {
                    ...
                    "password" : {
                        "title" : "Password",
                        ...
                        "encryption" : {
                            "key" : "openidm-sym-default"
                        },
                        "scope" : "private",
         ...
        }
    ]
}
```

Tip

To configure encryption of properties by using the Admin UI:

1. Select Configure > Managed Objects, and click on the object type whose property values you want to encrypt (for example User).
2. On the Properties tab, select the property whose value should be encrypted and select the Encrypt checkbox.

For information about encrypting attribute values from the command-line, see "Using the **encrypt** Subcommand".

Important

Hashing is a one way operation - property values that are hashed can not be "unhashed" in the way that they can be decrypted. Therefore, if you hash the value of any property, you cannot synchronize that property value to an external resource. For managed object properties with hashed values, you must either exclude those properties from the mapping or set a random default value if the external resource requires the property.

9.7.2. Encoding Attribute Values by Using Salted Hash Algorithms

To encode attribute values with salted hash algorithms, add the `secureHash` property to the attribute definition, and specify the algorithm that should be used to hash the value. OpenIDM supports the following hash algorithms:

`MD5`
`SHA-1`
`SHA-256`
`SHA-384`

SHA-512

The following excerpt of a managed.json file shows a managed object configuration that hashes the values of the password attribute using the SHA-1 algorithm:

```
{
    "objects" : [
        {
            "name" : "user",
            ...
            "schema" : {
                ...
                "properties" : {
                    ...
                    "password" : {
                        "title" : "Password",
                        ...
                        "secureHash" : {
                            "algorithm" : "SHA-1"
                        },
                        "scope" : "private",
         ...
        }
    ]
}
```

Tip

To configure hashing of properties by using the Admin UI:

1. Select Configure > Managed Objects, and click on the object type whose property values you want to hash (for example User).
2. On the Properties tab, select the property whose value must be hashed and select the Hash checkbox.
3. Select the algorithm that should be used to hash the property value.

   OpenIDM supports the following hash algorithms:

   MD5
   SHA-1
   SHA-256
   SHA-384
   SHA-512

For information about hashing attribute values from the command-line, see "Using the **secureHash** Subcommand".

9.8. Restricting HTTP Access to Sensitive Data

You can protect specific sensitive managed data by marking the corresponding properties as private. Private data, whether it is encrypted or not, is not accessible over the REST interface. Properties that are marked as private are removed from an object when that object is retrieved over REST.

To mark a property as private, set its `scope` to `private` in the `conf/managed.json` file.

The following extract of the `managed.json` file shows how HTTP access is prevented on the `password` and `securityAnswer` properties:

```
{
    "objects": [
        {
            "name": "user",
            "schema": {
                "id" : "http://jsonschema.net",
                "title" : "User",
                ...
                "properties": {
                ...
                    {
                        "name": "securityAnswer",
                        "encryption": {
                            "key": "openidm-sym-default"
                        },
                        "scope" : "private"
                    },
                    {
                        "name": "password",
                        "encryption": {
                            "key": "openidm-sym-default"
                        }'
                        "scope" : "private"
                    }
         },
         ...
      }
   ]
}
```

Tip

To configure private properties by using the Admin UI:

1. Select Configure > Managed Objects, and click on the object type whose property values you want to make private (for example User).

2. On the Properties tab, select the property that must be private and select the Private checkbox.

A potential caveat with using private properties is that private properties are *removed* if an object is updated by using an HTTP `PUT` request. A `PUT` request replaces the entire object in the repository. Because properties that are marked as private are ignored in HTTP requests, these properties are effectively removed from the object when the update is done. To work around this limitation, do not use `PUT` requests if you have configured private properties. Instead, use a `PATCH` request to update only those properties that need to be changed.

For example, to update the `givenName` of user jdoe, you could run the following command:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--header "Content-Type: application/json"
 \
--request POST
 \
--data '[
   {
   "operation":"replace",
   "field":"/givenName",
   "value":"Jon"
   }
]' \
"http://localhost:8080/openidm/managed/user?_action=patch&_queryId=for-userName&uid=jdoe"
```

Note

The filtering of private data applies only to direct HTTP read and query calls on managed objects. No automatic filtering is done for internal callers, and the data that these callers choose to expose.

Chapter 10
Configuring Social ID Providers

OpenIDM provides a standards-based solution for social authentication requirements, based on the OAuth 2.0 and OpenID Connect 1.0 standards. They are similar, as OpenID Connect 1.0 is an authentication layer built on OAuth 2.0.

This chapter describes how to configure OpenIDM to register and authenticate users with multiple social identity providers.

To configure different social identity providers, you'll take the same general steps:

- Set up the provider. You'll need information such as a `Client ID` and `Client Secret` to set up an interface with OpenIDM.
- Configure the provider on OpenIDM.
- Set up User Registration. Activate `Social Registration` in the applicable Admin UI screen or configuration file.
- Set up an authentication module. OpenIDM includes a `SOCIAL_PROVIDERS` module for this purpose. You'll configure that module in the same way for all supported providers, as described in "Configuring the Social Providers Authentication Module".
- After configuration is complete, test the result. For a common basic procedure, see "Testing the Social ID Provider".

To understand how data is transmitted between OpenIDM and a social identity provider, read "OpenID Connect Authorization Code Flow".

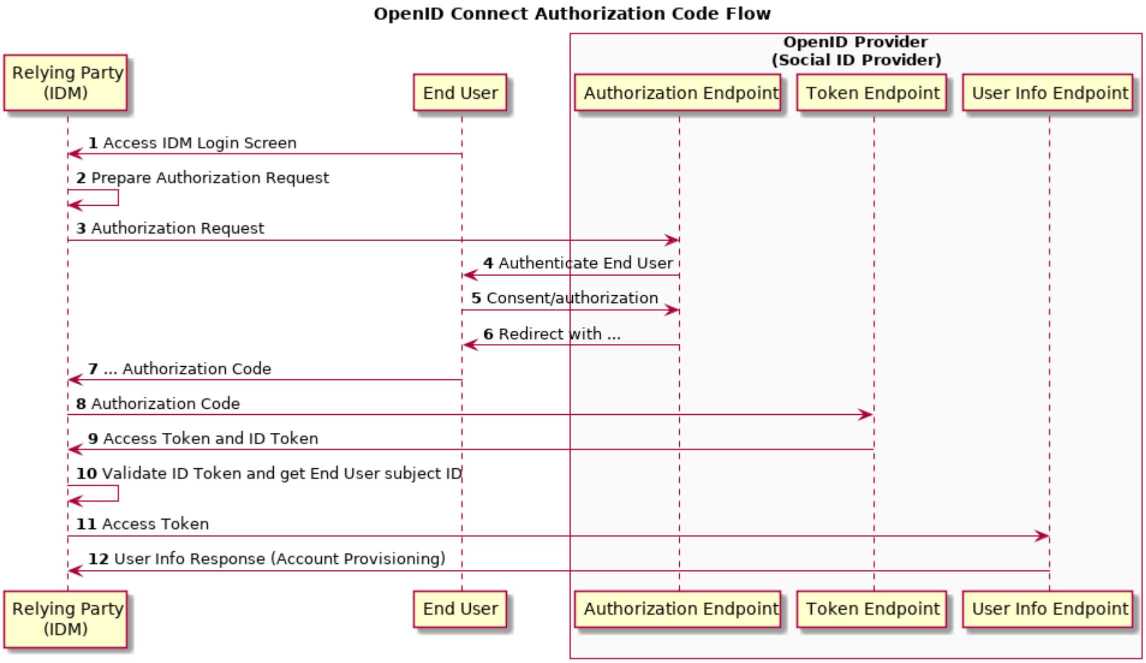
Note

For all social identity providers, set up a FQDN for OpenIDM, along with information in a DNS server, or system `hosts` files. For test purposes, FQDNs that comply with RFC 2606, such as `localhost` and `openidm.example.com`, are acceptable.

10.1. OpenID Connect Authorization Code Flow

The OpenID Connect Authorization Code Flow specifies how OpenIDM (Relying Party) interacts with the OpenID Provider (Social ID Provider), based on use of the OAuth 2.0 authorization grant. The following sequence diagram illustrates successful processing from the authorization request, through grant of the authorization code, access token, ID token, and provisioning from the social ID provider to OpenIDM.

OpenID Connect Authorization Code Flow for Social ID Providers



The following list describes details of each item in the authorization flow:

1. A user navigates to the OpenIDM Self-Service UI, and selects the `Sign In` link for the desired social identity provider.

2. OpenIDM prepares an authorization request.

3. OpenIDM sends the request to the Authorization Endpoint that you configured for the social identity provider, with a Client ID.

4. The social identity provider requests end user authentication and consent.

5. The end user transmits authentication and consent.

6. The social identity provider sends a redirect message, with an authorization code, to the end user's browser.

7. The browser transmits the redirect message, with the authorization code, to OpenIDM.

8. OpenIDM records the authorization code, and sends it to the social identity provider Token Endpoint.

9. The social identity provider token endpoint returns access and ID tokens.

10. OpenIDM validates the token, and sends it to the social identity provider User Info Endpoint.

11. The social identity provider responds with information on the user's account, that OpenIDM can provision as a new Managed User.

You'll configure these credentials and endpoints, in some form, for each social identity provider.

10.2. Many Social ID Providers, One Schema

Most social ID providers include common properties, such as name, email address, and location.

OpenIDM includes two sets of property maps that translate information from a social ID provider to your managed user objects. These property maps are as follows:

- The `identityProviders.json` file includes a `propertyMap` code block for each supported provider. This file maps properties from the provider to a generic managed user object. You should not customize this file.

- The `selfservice.propertymap.json` file translates the generic managed user properties to the managed user schema that you have defined in `managed.json`. If you have customized the managed user schema, this is the file that you must change, to indicate how your custom schema maps to the generic managed user schema.

Examine the `identityProviders.json` file in the `conf/` subdirectory for your project. The following excerpt represents the Facebook `propertyMap` code block from that file:

```
"propertyMap" : [
   {
      "source" : "id",
      "target" : "id"
   },
   {
      "source" : "name",
      "target" : "displayName"
   },
   {
      "source" : "first_name",
      "target" : "givenName"
   },
   {
      "source" : "last_name",
      "target" : "familyName"
   },
   {
      "source" : "email",
      "target" : "email"
   },
   {
      "source" : "email",
      "target" : "username"
   },
   {
      "source" : "locale",
      "target" : "locale"
   }
]
```

The source lists the Facebook property, the target lists the corresponding property for a generic managed user.

OpenIDM then processes that information through the `selfservice.propertymap.json` file, where the source corresponds to the generic managed user and the target corresponds to your customized managed user schema (defined in your project's `managed.json` file).

```
{
   "properties" : [
      {
         "source" : "givenName",
         "target" : "givenName"
      },
      {
         "source" : "familyName",
         "target" : "sn"
      },
      {
         "source" : "email",
         "target" : "mail"
      },
      {
         "source" : "postalAddress",
         "target" : "postalAddress",
         "condition" : "/object/postalAddress  pr"
      },
      {
         "source" : "addressLocality",
         "target" : "city",
         "condition" : "/object/addressLocality  pr"
      },
      {
         "source" : "addressRegion",
         "target" : "stateProvince",
         "condition" : "/object/addressRegion  pr"
      },
      {
         "source" : "postalCode",
         "target" : "postalCode",
         "condition" : "/object/postalCode  pr"
      },
      {
         "source" : "country",
         "target" : "country",
         "condition" : "/object/country  pr"
      },
      {
         "source" : "phone",
         "target" : "telephoneNumber",
         "condition" : "/object/phone  pr"
      },
      {
         "source" : "username",
         "target" : "userName"
      }
   ]
}
```

Tip

To take additional information from a social ID provider, make sure the property is mapped through the `identityProviders.json` and `selfservice.propertymap.json` files.

Several of the property mappings include a `pr` presence expression which is a filter that returns all records with the given attribute. For more information, see "Presence Expressions".

10.3. Setting Up Google as a Social Identity Provider

As suggested in the introduction to this chapter, you'll need to take four basic steps to configure Google as a social identity provider for OpenIDM:

- "Setting Up Google".
- "Configuring a Google Social ID Provider".
- "Configuring User Registration to Link to Google".
- "Configuring the Social Providers Authentication Module". (This section is common for all providers.)

10.3.1. Setting Up Google

To set up Google as a social identity provider for OpenIDM, navigate to the *Google API Manager*. You'll need a Google account. If you have GMail, you already have a Google account. While you could use a personal Google account, it is best to use an organizational account to avoid problems if specific individuals leave your organization. When you set up a Google social identity provider, you'll need to perform the following tasks:

Plan ahead. It may take some time before the `Google+` API that you configure for OpenIDM is ready for use.

- In the Google API Manager, select and enable the `Google+` API. It is one of the Google "social" APIs.
- Create a project for OpenIDM.
- Create OAuth client ID credentials. You'll need to configure an `OAuth consent screen` with at least a product name and email address.
- When you set up a Web application for the client ID, you'll need to set up a web client with:
  - `Authorized JavaScript origins`

    The origin URL for OpenIDM, typically a URL such as `https://openidm.example.com:8443`
  - `Authorized redirect URIs`

    The redirect URI after users are authenticated, typically: `https://openidm.example.com:8443/oauthReturn.html` and `https://openidm.example.com:8443/admin/oauthReturn.html`

    Note

    The `oauthReturn.html` file is needed as an intermediate step, as social identity providers do not allow redirect URIs with hash fragments.
- In the list of credentials, you'll see a unique `Client ID` and `Client secret`. You'll need this information when you configure the Google social ID provider in OpenIDM in the following procedure: "Configuring a Google Social ID Provider".

For Google's procedure, see the Google Identity Platform documentation on *Setting Up OAuth 2.0*.

10.3.2. Configuring a Google Social ID Provider

1. To configure a Google social ID provider, log into the Admin UI and navigate to Configure > Social ID Providers.
2. Enable the Google social ID provider, and select the edit icon.
3. Include the Google values for `Client ID` and `Client Secret` for your project, as described earlier in this section.

4. Under regular and `Advanced Options`, include the options shown in the following appendix: "Google Social ID Provider Configuration Details". The defaults are based on Google's documentation on *OpenID Connect*.

The default installation of OpenIDM does not include configuration files specific to social ID providers. When you enable a Google social ID provider in the Admin UI, OpenIDM generates the `identityProvider-google.json` file in your project's `conf/` subdirectory.

When you review that file, you should see information beyond what you see in the Admin UI. One part of the file includes the authentication protocol, `OPENID_CONNECT`, the sign-in button, *Authorization scopes*, and Google's authentication identifier for `authenticationId`.

```
{
    "name" : "google",
    "type" : "OPENID_CONNECT",
    "icon" : "<button class=\"btn btn-lg btn-default btn-block btn-social-provider btn-google\"><img
src=\"images/g-logo.png\"> {{action}} with Google</button>",
    "scope" : [
        "openid",
        "profile",
        "email"
    ],
    "authenticationId" : "id",
```

Another part of the file includes a `propertyMap`, which maps user information entries between the `source` (Google) and the `target` (OpenIDM).

The next part of the file includes `schema` information, which includes properties for each social ID account, as collected by OpenIDM, as well as the order in which it appears in the Admin UI. When you've registered a user with a Google social ID, you can verify this by selecting Manage > Google, and then selecting a user.

Finally, there's the part of the file that you may have configured through the Admin UI:

```
    "enabled" : true,
    "client_id" : "<someUUID>.apps.googleusercontent.com",
    "client_secret" : {
        "$crypto" : {
            "type" : "x-simple-encryption",
            "value" : {
                "cipher" : "AES/CBC/PKCS5Padding",
                "salt" : "<hashValue>",
                "data" : "<encryptedValue>",
                "iv" : "<encryptedValue>",
                "key" : "openidm-sym-default",
                "mac" : "<hashValue>"
            }
        }
    },
    "authorization_endpoint" : "https://accounts.google.com/o/oauth2/v2/auth",
    "token_endpoint" : "https://www.googleapis.com/oauth2/v4/token",
    "userinfo_endpoint" : "https://www.googleapis.com/oauth2/v3/userinfo",
    "well-known" : "https://accounts.google.com/.well-known/openid-configuration"
}
```

If you need more information about the properties in this file, refer to the following appendix: "Google Social ID Provider Configuration Details".

10.3.3. Configuring User Registration to Link to Google

Once you've configured the Google social ID provider, you can activate it through User Registration. To do so in the Admin UI, select Configure > User Registration, and enable the option associated with Social Registration. For more information on OpenIDM user self-service features, see "*Configuring User Self-Service*".

When you enable one or more social ID providers, OpenIDM changes the `selfservice-registration.json` file in the `conf/` subdirectory for your project, by replacing `userDetails` with `socialUserDetails` in the `stageConfigs` code block:

```
"name" : "socialUserDetails"
```

When you enable social ID providers in User Registration, you're allowing users to register through all active social identity providers.

10.4. Setting Up LinkedIn as a Social Identity Provider

As suggested in the introduction to this chapter, you'll need to take four basic steps to configure LinkedIn as a social identity provider for OpenIDM:

- "Setting Up LinkedIn".
- "Configuring a LinkedIn Social ID Provider".
- "Configuring User Registration to Link to LinkedIn".
- "Configuring the Social Providers Authentication Module". (This section is common to all social providers.)

10.4.1. Setting Up LinkedIn

To set up LinkedIn as a social identity provider for OpenIDM, navigate to the *LinkedIn Developers* page for `My Applications`. You'll need a LinkedIn account. While you could use a personal LinkedIn account, it is best to use an organizational account to avoid problems if specific individuals leave your organization. When you set up a LinkedIn social identity provider, you'll need to perform the following tasks:

- In the LinkedIn Developers page for My Applications, select Create Application.
- You'll need to include the following information when creating an application:
    - Company Name
    - Application Name

  - Description
  - Application Logo
  - Application Use
  - Website URL
  - Business Email
  - Business Phone
- When you see Authentication Keys for your LinkedIn application, save the `Client ID` and `Client Secret`.
- Enable the following default application permissions:
  - `r_basicprofile`
  - `r_emailaddress`
- When you set up a Web application for the client ID, you'll need to set up a web client with OAuth 2.0 Authorized Redirect URLs. For example, if your OpenIDM FQDN is `openidm.example.com`, add the following URLs:
  - https://openidm.example.com:8443/oauthReturn.html
  - https://openidm.example.com:8443/admin/oauthReturn.html

  You can ignore any LinkedIn URL boxes related to OAuth 1.0a.

For LinkedIn's procedure, see their documentation on *Authenticating with OAuth 2.0*.

10.4.2. Configuring a LinkedIn Social ID Provider

1. To configure a LinkedIn social ID provider, log into the Admin UI and navigate to Configure > Social ID Providers.
2. Enable the LinkedIn social ID provider.
3. Include the values that LinkedIn created for `Client ID` and `Client Secret`, as described in "Setting Up LinkedIn".
4. Under regular and `Advanced Options`, include the options shown in the following appendix: "LinkedIn Social ID Provider Configuration Details".

The default installation of OpenIDM includes configuration details specific to social ID providers. When you enable a Google social ID provider, OpenIDM generates the `identityProvider-linkedIn.json` file in your project's `conf/` subdirectory.

When you review that file, you should see information beyond what you see in the Admin UI. One part of the file includes the authentication protocol, `OAUTH`, the sign-in button, and LinkedIn's authentication identifier for `authenticationId`.

```
{
    "name" : "linkedIn",
    "type" : "OAUTH",
    "icon" : "<button class=\"btn btn-lg btn-default btn-block btn-social-provider\">
    {{action}} with LinkedIn</button>",
    "scope" : [
        "r_basicprofile",
        "r_emailaddress"
    ],
    "authenticationId" : "id",

```

Another part of the file includes a `propertyMap`, which maps user information entries between the `source` (LinkedIn) and the `target` (OpenIDM).

The next part of the file includes `schema` information, which includes properties for each social ID account, as collected by OpenIDM, as well as the order in which it appears in the Admin UI. When you've registered a user with a LinkedIn social ID, you can verify this by selecting Manage > LinkedIn, and then selecting a user.

Finally, there's the part of the file that you may have configured through the Admin UI:

```
    "enabled" : true,
    "client_id" : "<someUUID>",
    "client_secret" : {
        "$crypto" : {
            "type" : "x-simple-encryption",
            "value" : {
                "cipher" : "AES/CBC/PKCS5Padding",
                "salt" : "<hashValue>",
                "data" : "<encryptedValue>",
                "iv" : "<encryptedValue>",
                "key" : "openidm-sym-default",
                "mac" : "<hashValue>"
            }
        }
    },
    "authorization_endpoint" : "https://www.linkedin.com/oauth/v2/authorization",
    "token_endpoint" : "https://www.linkedin.com/oauth/v2/accessToken",
    "userinfo_endpoint" : "https://api.linkedin.com/v1/people/~:(id,formatted-name,first-name,last-
name,email-address,location)?format=json"
}
```

If you need more information about the properties in this file, refer to the following appendix: "LinkedIn Social ID Provider Configuration Details".

10.4.3. Configuring User Registration to Link to LinkedIn

Once you've configured the LinkedIn social ID provider, you can activate it through User Registration. To do so in the Admin UI, select Configure > User Registration, and enable the option associated with Social Registration. For more information on OpenIDM user self-service features, see "*Configuring User Self-Service*".

When you enable social ID providers, OpenIDM changes the `selfservice-registration.json` file in the `conf/` subdirectory for your project, by adding the following entry to the `stageConfigs` code block:

```
"name" : "socialUserDetails"
```

When you enable social ID providers in User Registration, you're allowing users to register through all active social identity providers.

10.5. Setting Up Facebook as a Social Identity Provider

As suggested in the introduction to this chapter, you'll need to take four basic steps to configure Facebook as a social identity provider for OpenIDM:

- "Setting Up Facebook"
- "Configuring a Facebook Social ID Provider"
- "Configuring User Registration to Link to Facebook"
- "Configuring the Social Providers Authentication Module". (This section is common to all social providers.)

10.5.1. Setting Up Facebook

To set up Facebook as a social identity provider for OpenIDM, navigate to the *Facebook for Developers* page. You'll need a Facebook account. While you could use a personal Facebook account, it is best to use an organizational account to avoid problems if specific individuals leave your organization. When you set up a Facebook social identity provider, you'll need to perform the following tasks:

> **Note**
>
> This procedure was tested with Facebook API version v2.7.

- In the Facebook for Developers page, select My Apps and Add a New App. For OpenIDM, you'll create a `Website` application.
- You'll need to include the following information when creating a Facebook website application:
  - Display Name
  - Contact Email
  - OpenIDM URL

- When complete, you should see your App and a link to a Dashboard. Navigate to the Dashboard for your App.
- Make a copy of the `App ID` and `App Secret` for when you configure the Facebook social ID provider in OpenIDM.
- In the settings for your App, you should see an entry for `App Domains`, such as `example.com`.

For Facebook's documentation on the subject, see *Facebook Login for the Web with the JavaScript SDK*.

10.5.2. Configuring a Facebook Social ID Provider

1. To configure a Facebook social ID provider, log into the Admin UI and navigate to Configure > Social ID Providers.
2. Enable the Facebook social ID provider.
3. Include the values that Facebook created for `App ID` and `App Secret`, as described in "Setting Up LinkedIn".
4. Under regular and `Advanced Options`, include the options shown in the following appendix: "Facebook Social ID Provider Configuration Details".

The default installation of OpenIDM includes configuration details specific to social ID providers. When you enable a Facebook social ID provider, in the Admin UI, OpenIDM generates the `identityProvider-facebook.json` file in your project's `conf/` subdirectory.

When you review that file, you should see information beyond what you see in the Admin UI. One part of the file includes the authentication protocol, `OAUTH`, the sign-in button, and Facebook's authentication identifier for `authenticationId`.

```
{
    "name" : "facebook",
    "type" : "OAUTH",
    "icon" : "<button class=\"btn btn-lg btn-default btn-block btn-social-provider\">
    {{action}} with Facebook</button>",
    "scope" : [
        "email",
        "user_birthday"
    ],
    "authenticationId" : "id",
```

Another part of the file includes a `propertyMap`, which maps user information entries between the `source` (Facebook) and the `target` (OpenIDM).

The next part of the file includes `schema` information, which includes properties for each social ID account, as collected by OpenIDM, as well as the order in which it appears in the Admin UI. When you've registered a user with a Facebook social ID, you can verify this by selecting Manage > Facebook, and then selecting a user.

Finally, there's the part of the file that you may have configured through the Admin UI:

```
    "enabled" : true,
    "client_id" : "<someUUID>",
    "client_secret" : {
        "$crypto" : {
            "type" : "x-simple-encryption",
            "value" : {
                "cipher" : "AES/CBC/PKCS5Padding",
                "salt" : "<hashValue>",
                "data" : "<encryptedValue>",
                "iv" : "<encryptedValue>",
                "key" : "openidm-sym-default",
                "mac" : "<hashValue>"
            }
        }
    },
    "authorization_endpoint" : "https://www.facebook.com/dialog/oauth",
    "token_endpoint" : "https://graph.facebook.com/v2.7/oauth/access_token",
    "userinfo_endpoint" : "https://graph.facebook.com/me?
fields=id,name,picture,email,first_name,last_name,locale"
}
```

If you need more information about the properties in this file, refer to the following appendix: "Facebook Social ID Provider Configuration Details".

10.5.3. Configuring User Registration to Link to Facebook

Once you've configured the Facebook social ID provider, you can activate it through User Registration. To do so in the Admin UI, select Configure > User Registration, and enable the option associated with Social Registration. For more information on OpenIDM user self-service features, see "*Configuring User Self-Service*".

When you enable social ID providers, OpenIDM changes the `selfservice-registration.json` file in the `conf/` subdirectory for your project, by adding the following entry to the `stageConfigs` code block:

```
"name" : "socialUserDetails"
```

When you enable social ID providers in User Registration, you're allowing users to register through all active social identity providers.

10.6. Setting Up a Custom Social Identity Provider

As suggested in the introduction to this chapter, you'll need to take five basic steps to configure a custom social identity provider for OpenIDM:

- "Preparing OpenIDM For a Custom Social ID Provider"
- "Setting Up a Custom Social ID Provider"
- "Configuring a Custom Social ID Provider"
- "Configuring User Registration to Link to a Custom Provider"

- "Configuring the Social Providers Authentication Module". (This section is common to all social providers.)

> **Note**
>
> These instructions require the social identity provider to be *fully* compliant with *The OAuth 2.0 Authorization Framework* or the *OpenID Connect* standards.

10.6.1. Preparing OpenIDM For a Custom Social ID Provider

While OpenIDM includes provisions to work with OpenID Connect 1.0 and OAuth 2.0 social identity providers, OpenIDM does not support connections to those providers, other than those listed in this chapter.

To set up another social provider, first add a code block to the `identityProviders.json` file, such as:

```
{
   "name" : "custom",
   "type" : "OAUTH",
   "icon" : "<button class=\"btn btn-lg btn-default btn-block btn-social-provider
      btn-custom\"><img src=\"images/custom-logo.png\">{{action}} with Custom Social ID</button>",
   "authorization_endpoint" : "",
   "token_endpoint" : "",
   "userinfo_endpoint" : "",
   "client_id" : "",
   "client_secret" : "",
   "scope" : [ ],
   "authenticationId" : "id",
   "schema" : {
      "id" : "http://jsonschema.net",
      "viewable" : true,
      "type" : "object",
      "$schema" : "http://json-schema.org/draft-03/schema",
      "properties" : {
         "id" : {
            "title" : "ID",
            "viewable" : true,
            "type" : "string",
            "searchable" : true
         },
         "name" : {
            "title" : "Name",
            "viewable" : true,
            "type" : "string",
            "searchable" : true
         },
         "first_name" : {
            "title" : "First Name",
            "viewable" : true,
            "type" : "string",
            "searchable" : true
         },
         "last_name" : {
            "title" : "Last Name",
            "viewable" : true,
            "type" : "string",
            "searchable" : true
         },
         "email" : {
            "title" : "Email Address",
            "viewable" : true,
            "type" : "string",
            "searchable" : true
         },
         "locale" : {
            "title" : "Locale Code",
            "viewable" : true,
            "type" : "string",
            "searchable" : true
         }
      },
      "order" : [
         "id",
         "name",
         "first_name",
         "last_name",
         "email",
         "locale"
      ],
      "required" : [ ]
   },
   "propertyMap" : [
      {
         "source" : "id",
         "target" : "id"
      },
      {
         "source" : "name",
         "target" : "displayName"
      },
      {
         "source" : "first_name",
         "target" : "givenName"
      },
      {
         "source" : "last_name",
         "target" : "familyName"
      },
      {
         "source" : "email",
         "target" : "email"
      },
      {
         "source" : "email",
         "target" : "username"
      },
      {
         "source" : "locale",
         "target" : "locale"
      }
   ]
},
```

Modify this code block for your selected social provider. Some of these properties may appear under other names. For example, some providers specify an `App ID` that you'd include as a `client_id`.

In the `propertyMap` code block, you should substitute the properties from the selected social ID provider for various values of `source`. Make sure to trace the property mapping through `selfservice.propertymap.json` to the Managed User property shown in `managed.json`. For more information on this multi-step mapping, see "Many Social ID Providers, One Schema".

As shown in "OpenID Connect Authorization Code Flow", user provisioning information goes through the User Info Endpoint. Some providers, such as Linkedin and Facebook, may require a list of properties with the endpoint. Consult the documentation for your provider for details.

With the `icon` property, note the `{{action}}` tag. It is a placeholder; OpenIDM substitutes *Sign in* or *Register* for the tag, depending on the functionality of the UI login page.

You can configure some of this code block through the Admin UI. Based on the `"name" : "custom"` line in the code block, select Configure > Social ID Providers. You'll see the entry as `Custom`, and you can configure the provider in the same way as others.

Alternatively, you can copy that code block directly to a new file. Based on `"name" : "custom"` you'd create the following file: `identityProvider-custom.json`.

Both files, `identityProviders.json` and `identityProvider-custom.json`, should include the same information for the new `custom` identity provider. For property details, see "Custom Social ID Provider Configuration Details".

Once you've included information from your selected social ID provider, proceed with the configuration process. You'll use the same basic steps described for other specified social providers.

10.6.2. Setting Up a Custom Social ID Provider

Every social identity provider should be able to provide the information you need to specify properties in the code block shown in "Preparing OpenIDM For a Custom Social ID Provider".

In general, you'll need an `authorization_endpoint`, a `token_endpoint` and a `userinfo_endpoint`. To link to the custom provider, you'll also have to copy the `client_id` and `client_secret` that you created with that provider. In some cases, you'll get this information in a slightly different format, such as an `App ID` and `App Secret`.

For the `propertyMap`, check the `source` properties. You may need to revise these properties to match those available from your custom provider.

For examples, refer to the specific social ID providers documented in this chapter.

10.6.3. Configuring a Custom Social ID Provider

1. To configure a custom social ID provider, log into the Admin UI and navigate to Configure > Social ID Providers.

2. Enable the custom social ID provider. The name you see is based on the `name` property in the relevant code block in the `identityProviders.json` file.

3. If you haven't already done so, include the values provided by your social ID provider for the properties shown. For more information, see the following appendix: "Custom Social ID Provider Configuration Details".

10.6.4. Configuring User Registration to Link to a Custom Provider

Once you've configured a custom social ID provider, you can activate it through User Registration. To do so in the Admin UI, select Configure > User Registration, and enable the option associated with Social Registration. For more information on OpenIDM user self-service features, see "*Configuring User Self-Service*".

When you enable social ID providers, OpenIDM changes the `selfservice-registration.json` file in the `conf/` subdirectory for your project, by adding the following entry to the `stageConfigs` code block:

```
"name" : "socialUserDetails"
```

When you enable social ID providers in User Registration, you're allowing users to register through all active social identity providers.

10.7. Configuring the Social Providers Authentication Module

OpenIDM includes a `SOCIAL_PROVIDERS` authentication module, which incorporates the requirements from social ID providers who rely on either the OAuth2 or the OpenID Connect standards. To configure this module in the Admin UI, select Configure > Authentication, choose the Modules tab. In the Select a Module text box, select and enable the Social Providers authentication module.

When configured, OpenIDM adds the following code block to the `authentication.json` file for your project:

```
{
    "enabled" : true,
    "properties" : {
        "queryOnResource" : "managed/user",
        "defaultUserRoles" : [
            "openidm-authorized"
        ],
        "propertyMapping" : {
            "userRoles" : "authzRoles"
        }
    },
    "name" : "SOCIAL_PROVIDERS"
}
```

For more information on these options, see "Common Module Properties".

10.8. Managing the Social ID Provider Over REST

You can identify the current status of configured social ID providers with the following REST call:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
http://localhost:8080/openidm/authentication
```

The output that you see includes JSON information from each configured social ID provider, as described in the `identityProvider-`*`provider`* file in your project's `conf/` subdirectory.

One key line from this output specifies whether the social ID provider is enabled:

```
"enabled" : true
```

If the `SOCIAL_PROVIDERS` authentication module is disabled, you'll see the following output from that REST call:

```
{
   "providers" : [ ]
}
```

For more information, see "Configuring the Social Providers Authentication Module".

If the `SOCIAL_PROVIDERS` module is disabled, you can still review the standard configuration of each social provider (enabled or not) by running the same REST call on a different endpoint (do not forget the `s` at the end of `identityProviders`):

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
http://localhost:8080/openidm/identityProviders
```

> **Note**
>
> If you have not configured a social ID provider, you'll see the following output from the REST call on the `openidm/identityProviders` endpoint:
>
> ```
> {
>     "providers" : [ ]
> }
> ```

You can still get information about the available configuration for social ID providers on a slightly different endpoint:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
http://localhost:8080/openidm/config/identityProviders
```

The `config` in the endpoint refers to the configuration, starting with the `identityProviders.json` configuration file. Note how it matches the corresponding term in the endpoint.

You can review information for a specific provider by including the name with the endpoint. For example, if you've configured LinkedIn as described in "Setting Up LinkedIn as a Social Identity Provider", run the following command:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
http://localhost:8080/openidm/config/identityProvider/linkedIn
```

The above command differs in subtle ways. The `config` in the endpoint points to configuration data. The `identityProvider` at the end of the endpoint is singular, which matches the corresponding configuration file, `identityProvider-linkedIn.json`. And `linkedIn` includes a capital `I` in the middle of the word.

In a similar fashion, you can delete a specific provider:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request DELETE \
http://localhost:8080/openidm/config/identityProvider/linkedIn
```

If you have the information needed to set up a provider, such as the output from the previous two REST calls, you can use the following command to add a provider:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--header "Content-type: application/json"
 \
--request PUT
 \
--data '{
   "_id" : "identityProvider/linkedIn",
   "name" : "linkedIn",
   "type" : "OAUTH",
   "icon" : "<button class=\"btn btn-lg btn-default btn-block btn-social-provider btn-linkedin\"><img src=
\"images/ln-logo.png\">{{action}} LinkedIn</button>",
   "scope" : [ "r_basicprofile", "r_emailaddress" ],
   "authenticationId" : "id",
   "propertyMap" : [ {
      "source" : "id",
      "target" : "id"
   }, {
      "source" : "formattedName",
      "target" : "displayName"
   }, {
      "source" : "firstName",
      "target" : "givenName"
   }, {
      "source" : "lastName",
      "target" : "familyName"
   }, {
      "source" : "emailAddress",
      "target" : "email"
   }, {
      "source" : "emailAddress",
      "target" : "username"
   }, {
      "source" : "location",
      "target" : "locale",
      "transform" : {
         "type" : "text/javascript",
         "source" : "source.country.code",
         "file" : null
      }
   } ],
   "enabled" : true,
   "client_id" : "<some UUID>",
   "client_secret" : "<some client secret>",
   "authorization_endpoint" : "https://www.linkedin.com/oauth/v2/authorization",
   "token_endpoint" : "https://www.linkedin.com/oauth/v2/accessToken",
   "userinfo_endpoint" : "https://api.linkedin.com/v1/people/~:(id,formatted-name,first-name,last-name
,email-address,location)?format=json"
}' \
http://localhost:8080/openidm/config/identityProvider/linkedIn
```

You can even disable a social ID provider with a `PATCH` REST call, as shown:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--header "Content-type: application/json"
 \
--request PATCH
 \
--data '[
   {
      "operation":"replace",
      "field" : "enabled",
      "value" : false
   }
]' \
http://localhost:8080/openidm/config/identityProvider/linkedIn
```

You can reverse the process by substituting `true` for `false` in the previous `PATCH` REST call.

You can manage the social ID providers associated with individual users over REST, as described in "Managing Links Between End User Accounts and Social ID Providers".

10.9. Testing the Social ID Provider

In all cases, once configuration is complete, you should test the Social ID Provider. To do so, go through the steps in the following procedure:

1. Navigate to the login screen for the self-service UI, `https://openidm.example.com:8443`.
2. Select the `Register` link (after the "Don't have an account?" question) on the login page.
3. You should see a link to sign in with your selected social ID provider. Select that link.

   > **Note**
   >
   > If you do not see a link to sign in with any social ID provider, you probably did not enable the option associated with Social Registration. To make sure, access the Admin UI, and select Configure > User Registration.

   > **Warning**
   >
   > If you see a redirect URI error from a social ID provider, check the configuration for your web application in the social ID provider developer console. There may be a mistake in the redirect URI or redirect URL.

4. Follow the prompts from your social ID provider to log into your account.
5. You should next see the OpenIDM Register Your Account screen, pre-populated with a username, first name, last name, and email address (if available). You should be able to modify these entries. If the username already exists in the OpenIDM managed user datastore, you'll have to change that username before you can save and register the account.
6. As Knowledge-based Authentication (KBA) is enabled by default, you'll need to add at least one security question and answer to proceed. For more information, see "Configuring Self-Service Questions".

   When the Social ID registration process is complete, OpenIDM takes you to the self-service login URL at `https://openidm.example.com:8443`.

7. At the self-service login URL, you should now be able to use the sign in link for your social ID provider to log into OpenIDM.

10.10. Managing Links Between End User Accounts and Social ID Providers

If your users have one or more social ID providers, they can link them to the same OpenIDM user account. This section assumes that you have configured more than one of the social ID providers described in this chapter.

Conversely, you should not be able to configure more than one OpenIDM account with a single social ID provider account. When social accounts are associated with an account, a related managed record is created for the user. This related record uses the social ID provider name as the managed object type, and the subject is used as the `_id`. This combination has a unique constraint; if you try to associate a second OpenIDM account with the same social account, OpenIDM detects a conflict, which prevents the association.

10.10.1. The Process for End Users

When your users register with a social ID provider, as defined in "Testing the Social ID Provider", they create an account in your OpenIDM managed user datastore. They can link additional social ID providers to that datastore, using the following steps:

1. Navigate to the self-service UI, at an URL such as `https://openidm.example.com:8443`.
2. Log into the account, either as an OpenIDM user, or with the social ID provider.
3. Navigate to Profile > Social Identities.
4. Enable a second social ID provider. Unless you've previously authenticated with that social provider, you should be prompted to log into that provider.
5. To test the result, log out and log back in, using the link for the second social ID provider.

10.10.2. Reviewing Linked Accounts as an Administrator

You can review social ID accounts linked to an OpenIDM account, from the Admin UI and from the command line. You can disable or delete social ID provider information for a specific user from the command line, as described in "Reviewing Linked Accounts Over REST".

> **Note**
>
> An end-user can unbind social providers through their managed user accounts. However, an administrative user *cannot* delete social provider accounts through the Admin UI.

When you activate a social ID provider, OpenIDM creates a new managed object for that provider. You can review that managed object in the `managed.json` file, as well as in the Admin UI, by selecting Configure > Managed Objects.

The information shown is reflected in the schema in the `identityProvider-providername.json` file for the selected provider.

> **Note**
>
> Best practice: do not edit social ID provider profile information via OpenIDM. Any changes that you make won't be synchronized with that provider.

10.10.2.1. Reviewing Linked Accounts Over REST

You can also review the social ID accounts linked to specific users with REST calls. Start by finding the `_id` for your user with the following command:

```
$ curl \
--header "X-OpenIDM-Username:openidm-admin"
 \
--header "X-OpenIDM-Password:openidm-admin"
 \
--request GET \
"http://localhost:8080/openidm/managed/user/?_queryId=query-all-ids"
```

The following REST call finds all data from a specified user.

```
$ curl \
--header "X-OpenIDM-Username:openidm-admin"
 \
--header "X-OpenIDM-Password:openidm-admin"
 \
--request GET \
"http://localhost:8080/openidm/managed/user/10aa857f-b2cc-47a4-a295-f842df96e5e8"
```

From the following output, you can see how Jane Doe's `idpData` includes Linkedin information in a code block similar to her Google information. The order makes no functional difference in whether users can log in via their Google or Linkedin accounts.

```
{
  "_id" : "10aa857f-b2cc-47a4-a295-f842df96e5e8",
  "_rev" : "2",
  "givenName" : "Jane",
  "sn" : "Doe",
  "mail" : "Jane.Doe@example.com",
  "userName" : "Jane.Doe@example.com",
  "idpData" : {
    "google" : {
      "subject" : "105533855303935356522",
      "enabled" : true,
      "dateCollected" : "2016-09-16T17:25Z",
      "rawProfile" : {
        "sub" : "105533855303935356522",
        "name" : "Jane",
        "given_name" : "Jane",
        "family_name" : "Doe",
        "profile" : "https://plus.google.com/<some number>",
        "picture" : "https://lh4.googleusercontent.com/<some path>/photo.jpg",
        "email" : "Jane.Doe@example.com",
        "email_verified" : true,
        "gender" : "female",
        "locale" : "en",
        "hd" : "example.com"
      }
    },
    "linkedIn" : {
      "rawProfile" : {
        "emailAddress" : "Jane.Doe@example.net",
        "firstName" : "Jane",
        "formattedName" : "Jane Doe",
        "id" : "MW9FE_KyQH",
        "lastName" : "Doe",
        "location" : {
          "country" : {
            "code" : "us"
          },
          "name" : "Portland, Oregon Area"
        }
      },
      "enabled" : true,
      "subject" : "MW9FE_KyQH"
    }
  },
  "kbaInfo" : [ {
    "answer" : {
      "$crypto" : {
        "value" : {
          "algorithm" : "SHA-256",
          "data" : "<some hashed value>"
        },
        "type" : "salted-hash"
      }
    },
    "questionId" : "1"
  } ],
  "accountStatus" : "active",
  "effectiveRoles" : [ ],
  "effectiveAssignments" : [ ]
```

When a user disables logins via one specific social ID provider in the self- service UI, that sets `"enabled" : false` in the data for that provider. However, that user's social ID information is preserved.

Alternatively, you can use a REST call to disable logins to a specific social ID provider. The following REST call disables logins for the same user via Google:

```
$ curl  \
--header "-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--header "Content-type: application/json"
 \
--request POST \
"http://localhost:8080/openidm/managed/user/10aa857f-b2cc-47a4-a295-f842df96e5e8?
_action=unbind&provider=google"
```

For privacy purposes, you can also set up deletion of a disabled social ID provider. To do so, you need to make one change to the `unBindBehavior.js` file in the following `/path/to/openidm` subdirectory: `bin/defaults/script/ui/`.

```
// uncomment below line to delete social provider data -
     // delete object.idpData[request.additionalParameters.provider];
```

As suggested by the file, when you uncomment the noted line, disabling one social ID provider (in the UI or via REST) removes data for that provider from that user's information in the OpenIDM repository.

10.10.2.2. Reviewing Linked Accounts From the Admin UI

When you configure a social ID provider, OpenIDM includes two features in the Admin UI.

- The ability to review the social ID accounts linked to specific users. To see how this works, log into the Admin UI, and select Manage > User, and select a user. Under the Identity Providers tab, you can review the social ID providers associated with a specific account.
- A managed object for each provider. For example, if you've enabled Google as a social ID provider, select Manage > Google. In the screen that appears, you can select the ID for any Google social ID account that has been used or linked to an existing OpenIDM account, and review the profile information shared from that provider.

Chapter 11
Using Policies to Validate Data

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

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

The policy service enables you to accomplish the following tasks:

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

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

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

11.1. Configuring the Default Policy for Managed Objects

Policies applied to managed objects are configured in two files:

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

Note

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

11.1.1. Understanding the Policy Script File

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

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

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

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

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

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

  return [];
}
...
```

To enforce user passwords that contain at least one capital letter, the `policyId` from the preceding example is applied to the appropriate resource (`managed/user/*`). The required number of capital letters is defined in the policy configuration element of the managed object configuration file (see "Understanding the Policy Configuration Element".

11.1.1.1. Policy Configuration Objects

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

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

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

11.1.1.2. Policy Implementation Functions

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

```

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

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

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

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

11.1.2. Understanding the Policy Configuration Element

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

```
{
    "objects" : [
        {
            "name" : "user",
            ...
            "schema" : {
                "id" : "http://jsonschema.net",
                ...
                "properties" : {
                    "_id" : {
                        "type" : "string",
                        "viewable" : false,
                        "searchable" : false,
                        "userEditable" : false,
                        "policies" : [
                            {
                                "policyId" : "cannot-contain-characters",
                                "params" : {
                                    "forbiddenChars" : ["/"]
                                }
                            }
                        ]
                    },
                    "password" : {
                        "type" : "string",
                        "viewable" : false,
                        "searchable" : false,
                        "minLength" : 8,
                        "userEditable" : true,
                        "policies" : [
                            {
                                "policyId" : "at-least-X-capitals",
                                "params" : {
                                    "numCaps" : 1
                                }
                            },
                            {
                                "policyId" : "at-least-X-numbers",
                                "params" : {
                                    "numNums" : 1
                                }
                            },
                            {
                                "policyId" : "cannot-contain-others",
                                "params" : {
                                    "disallowedFields" : [
                                        "userName",
                                        "givenName",
                                        "sn"
                                    ]
                                }
                            }
                        ]
                    },
```

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

11.1.3. Validation of Managed Object Data Types

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

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

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

```
"telephoneNumber" : {
    "description" : "",
    "title" : "Mobile Phone",
    "viewable" : true,
    "searchable" : false,
    "userEditable" : true,
    "policies" : [ ],
    "returnByDefault" : false,
    "minLength" : null,
    "pattern" : "^\\+?([0-9\\- \\(\\)])*$",
    "type" : [
        "string",
        "null"
    ]
},
```

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

11.1.4. Configuring Policy Validation in the UI

The Admin UI provides rudimentary support for applying policy validation to managed object properties. To configure policy validation for a managed object type update the configuration of the object type in the UI. For example, to specify validation policies for specific properties of managed user objects, select Configure > Managed Objects then click on the User object. Scroll down to the bottom of the Managed Object configuration, then update, or add, a validation policy. The `Policy` field here refers to a function that has been defined in the policy script file. For more information, see "Understanding the Policy Script File". You cannot define additional policy functions by using the UI.

> **Note**
>
> Take care with Validation Policies. If it relates to an array of relationships, such as between a user and multiple devices, "Return by Default" should always be set to false. You can verify this in the `managed.json` file for your project, with the `"returnByDefault" : false` entry for the applicable managed object, whenever there are `items` of `"type" : "relationship"`.

11.2. Extending the Policy Service

You can extend the policy service by adding custom scripted policies, and by adding policies that are applied only under certain conditions.

11.2.1. Adding Custom Scripted Policies

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

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

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

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

addPolicy(policy);

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

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

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

11.2.2. Adding Conditional Policy Definitions

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

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

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

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

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

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

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

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

Note

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

11.3. Disabling Policy Enforcement

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

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

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

11.4. Managing Policies Over REST

You can manage the policy service over the REST interface, by calling the REST endpoint https://localhost:8443/openidm/policy, as shown in the following examples.

11.4.1. Listing the Defined Policies

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

```
$ curl \
 --cacert self-signed.crt \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "https://localhost:8443/openidm/policy"
{
  "_id": "",
  "resources": [
    {
      "resource": "repo/internal/user/*",
      "properties": [
        {
          "name": "_id",
          "policies": [
            {
              "policyId": "cannot-contain-characters",
              "params": {
                "forbiddenChars": [
                  "/"
                ]
              },
              "policyFunction": "\nfunction (fullObject, value, params,
 property)
...
```

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

```
$ curl \
 --cacert self-signed.crt \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "https://localhost:8443/openidm/policy/managed/user/*"
{
  "_id": "*",
  "resource": "managed/user/*",
  "properties": [
    {
      "name": "_id",
      "conditionalPolicies": null,
      "fallbackPolicies": null,
      "policyRequirements": [
        "CANNOT_CONTAIN_CHARACTERS"
      ],
      "policies": [
        {
          "policyId": "cannot-contain-characters",
          "params": {
            "forbiddenChars": [
              "/"
            ]
...
```

11.4.2. Validating Objects and Properties Over REST

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

The following example verifies that a new managed user object is acceptable, in terms of the policy requirements:

```
$ curl \
 --cacert self-signed.crt \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{
  "sn":"Jones",
  "givenName":"Bob",
  "_id":"bjones",
  "telephoneNumber":"0827878921",
  "passPhrase":null,
  "mail":"bjones@example.com",
  "accountStatus":"active",
  "userName":"bjones@example.com",
  "password":"123"
 }' \
 "https://localhost:8443/openidm/policy/managed/user/bjones?_action=validateObject"
{
  "result": false,
  "failedPolicyRequirements": [
    {
      "policyRequirements": [
        {
          "policyRequirement": "MIN_LENGTH",
          "params": {
            "minLength": 8
          }
        }
      ],
      "property": "password"
    },
    {
      "policyRequirements": [
        {
          "policyRequirement": "AT_LEAST_X_CAPITAL_LETTERS",
          "params": {
            "numCaps": 1
          }
        }
      ],
      "property": "password"
    }
  ]
}
```

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

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

The following example checks whether Barbara Jensen's new password (12345) is acceptable:

```
$ curl \
 --cacert self-signed.crt \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{ "password" : "12345" }' \
 "https://localhost:8443/openidm/policy/managed/user/bjensen?_action=validateProperty"
{
  "result": false,
  "failedPolicyRequirements": [
    {
      "policyRequirements": [
        {
          "policyRequirement": "MIN_LENGTH",
          "params": {
            "minLength": 8
          }
        }
      ],
      "property": "password"
    },
    {
      "policyRequirements": [
        {
          "policyRequirement": "AT_LEAST_X_CAPITAL_LETTERS",
          "params": {
            "numCaps": 1
          }
        }
      ],
      "property": "password"
    }
  ]
}
```

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

Validating a property that does fulfil the policy requirements returns a true result, for example:

```
$ curl \
 --cacert self-signed.crt \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{ "password" : "1NewPassword" }' \
 "https://localhost:8443/openidm/policy/managed/user/bjensen?_action=validateProperty"
{
  "result": true,
  "failedPolicyRequirements": []
}
```

Chapter 12
Configuring Server Logs

In this chapter, you will learn about server logging, that is, the messages that OpenIDM logs related to server activity.

Server logging is separate from *auditing*. Auditing logs activity on the OpenIDM system, such as access and synchronization. For information about audit logging, see "*Logging Audit Information*". To configure server logging, edit the logging.properties file in your *project-dir*/conf directory.

Important

When you change the logging settings you must restart the server for those changes to take effect. Alternatively, you can use JMX via jconsole to change the logging settings, in which case changes take effect without restarting the server.

12.1. Log Message Files

The default configuration writes log messages in simple format to openidm/logs/openidm*.log files, rotating files when the size reaches 5 MB, and retaining up to 5 files. Also by default, OpenIDM writes all system and custom log messages to the files.

You can modify these limits in the following properties in the logging.properties file for your project:

```
# Limiting size of output file in bytes:
java.util.logging.FileHandler.limit = 5242880

# Number of output files to cycle through, by appending an
# integer to the base file name:
java.util.logging.FileHandler.count = 5
```

12.2. Specifying the Logging Level

By default, IDM logs messages at the INFO level. This logging level is specified with the following global property in conf/logging.properties:

```
.level=INFO
```

You can specify different separate logging levels for individual server features which override the global logging level. Set the log level, per package to one of the following:

```
SEVERE (highest value)
WARNING
INFO
CONFIG
FINE
FINER
FINEST (lowest value)
```

For example, the following setting decreases the messages logged by the embedded PostgreSQL database:

```
# reduce the logging of embedded postgres since it is very verbose
ru.yandex.qatools.embed.postgresql.level = SEVERE
```

Set the log level to `OFF` to disable logging completely (see in "Disabling Logs"), or to `ALL` to capture all possible log messages.

If you use `logger` functions in your JavaScript scripts, set the log level for the scripts as follows:

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

You can override the log level settings, per script, with the following setting:

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

For more information about using `logger` functions in scripts, see "Logging Functions".

Important

It is strongly recommended that you do *not* log messages at the `FINE` or `FINEST` levels in a production environment. Although these levels are useful for debugging issues in a test environment, they can result in accidental exposure of sensitive data. For example, a password change patch request can expose the updated password in the Jetty logs.

12.3. Disabling Logs

You can also disable logs if desired. For example, before starting OpenIDM, you can disable `ConsoleHandler` logging in your project's `conf/logging.properties` file.

Just set `java.util.logging.ConsoleHandler.level = OFF`, and comment out other references to `ConsoleHandler`, as shown in the following excerpt:

```
   # ConsoleHandler: A simple handler for writing formatted records to System.err
   #handlers=java.util.logging.FileHandler, java.util.logging.ConsoleHandler
   handlers=java.util.logging.FileHandler
   ...
   # --- ConsoleHandler ---
   # Default: java.util.logging.ConsoleHandler.level = INFO
   java.util.logging.ConsoleHandler.level = OFF
   #java.util.logging.ConsoleHandler.formatter = ...
   #java.util.logging.ConsoleHandler.filter=...
```

Chapter 13
Connecting to External Resources

This chapter describes how to connect to external resources such as LDAP, Active Directory, flat files, and others. Configurations shown here are simplified to show essential aspects. Not all resources support all OpenIDM operations; however, the resources shown here support most of the CRUD operations, and also reconciliation and liveSync.

In OpenIDM, *resources* are external systems, databases, directory servers, and other sources of identity data that are managed and audited by the identity management system. To connect to resources, OpenIDM loads the Identity Connector Framework, OpenICF. OpenICF aims to avoid the need to install agents to access resources, instead using the resources' native protocols. For example, OpenICF connects to database resources using the database's Java connection libraries or JDBC driver. It connects to directory servers over LDAP. It connects to UNIX systems by using **ssh**.

13.1. The Open Identity Connector Framework (OpenICF)

OpenICF provides a common interface to allow identity services access to the resources that contain user information. OpenIDM loads the OpenICF API as one of its OSGi modules. OpenICF uses *connectors* to separate the OpenIDM implementation from the dependencies of the resource to which OpenIDM is connecting. A specific connector is required for each remote resource. Connectors can run either locally or remotely.

Local connectors are loaded by OpenICF as regular bundles in the OSGi container. Most connectors can be run locally. Remote connectors must be executed on a remote *connector server*. If a resource requires access libraries that cannot be included as part of the OpenIDM process, you must use a connector server. For example, OpenICF OpenICF connects to Microsoft Active Directory through a remote connector server that is implemented as a .NET service.

Connections to remote connector servers are configured in a single *connector info provider* configuration file, located in your project's `conf/` directory.

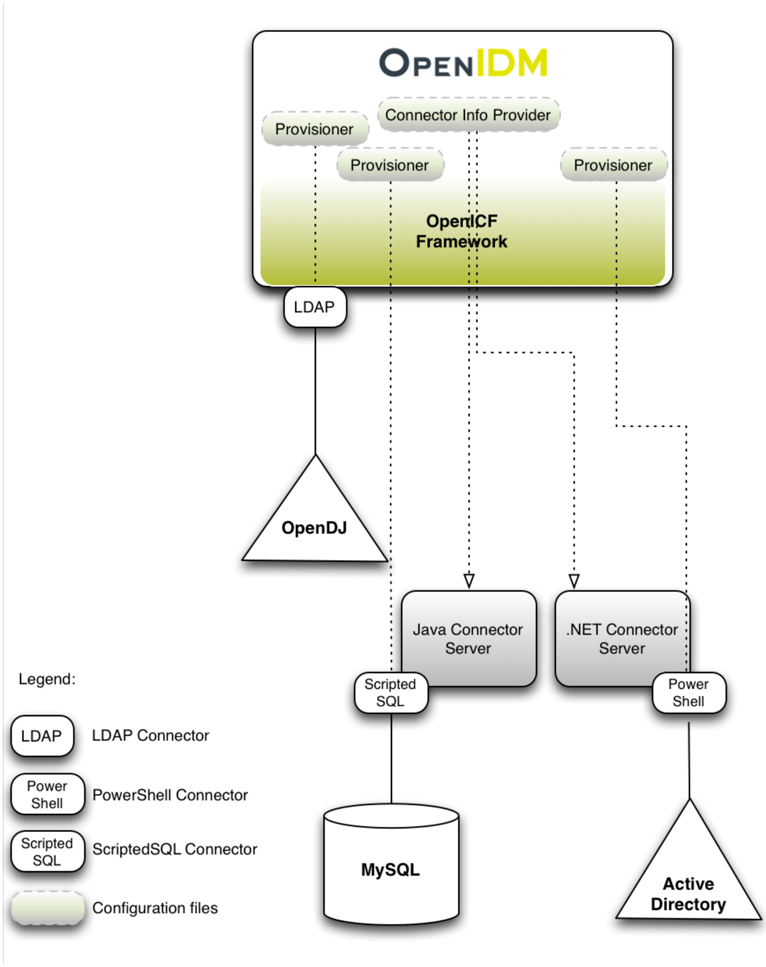
Connectors themselves are configured through *provisioner* files. One provisioner file must exist for each connector. Provisioner files are named `provisioner.openicf-name` where *name* corresponds to the name of the connector, and are also located in the `conf/` directory.

A number of sample connector configurations are available in the `openidm/samples/provisioners` directory. To use these connectors, edit the configuration files as required, and copy them to your project's `conf/` directory.

The following figure shows how OpenIDM connects to resources by using connectors and remote connector servers. The figure shows one local connector (LDAP) and two remote connectors (Scripted

SQL and PowerShell). In this example, the remote Scripted SQL connector uses a remote Java connector server. The remote PowerShell connector always requires a remote .NET connector server.

How OpenIDM Uses the OpenICF Framework and Connectors



Tip

Connectors that use the .NET framework *must* run remotely. Java connectors can be run locally or remotely. You might run a Java connector remotely for security reasons (firewall constraints), for geographical reasons, or if the JVM version that is required by the connector conflicts with the JVM version that is required by OpenIDM.

13.2. Accessing Remote Connectors

When you configure a remote connector, you use the *connector info provider service* to connect through a remote connector server. The connector info provider service configuration is stored in the file `project-dir/conf/provisioner.openicf.connectorinfoprovider.json`. A sample configuration file is provided in the `openidm/samples/provisioners/` directory. To use this sample configuration, edit the file as required, and copy it to your project's `conf/` directory.

The sample connector info provider configuration is as follows:

```
{
   "remoteConnectorServers" :
      [
         {
            "name" : "dotnet",
            "host" : "127.0.0.1",
            "port" : 8759,
            "useSSL" : false,
            "timeout" : 0,
            "protocol" : "websocket",
            "key" : "Passw0rd"
         }
      ]
}
```

You can configure the following remote connector server properties:

`name`

string, required

The name of the remote connector server object. This name is used to identify the remote connector server in the list of connector reference objects.

`host`

string, required

The remote host to connect to.

`port`

integer, optional

The remote port to connect to. The default remote port is 8759.

`heartbeatInterval`

integer, optional

The interval, in seconds, at which heartbeat packets are transmitted. If the connector server is unreachable based on this heartbeat interval, all services that use the connector server are made unavailable until the connector server can be reached again. The default interval is 60 seconds.

`useSSL`

boolean, optional

Specifies whether to connect to the connector server over SSL. The default value is `false`.

`timeout`

integer, optional

Specifies the timeout (in milliseconds) to use for the connection. The default value is `0`, which means that there is no timeout.

`protocol`

string

Version 1.5.2.0 of the OpenICF framework supports a new communication protocol with remote connector servers. This protocol is enabled by default, and its value is `websocket` in the default configuration.

For compatibility reasons, you might want to enable the legacy protocol for specific remote connectors. For example, if you deploy the connector server on a Java 5 or 6 JVM, you must use the old protocol. In this case, remove the `protocol` property from the connector server configuration.

For the .NET connector server, the service with the new protocol listens on port 8759 and the service with the legacy protocol listens on port 8760 by default. For more information on running the connector server in legacy mode, see "Running the .NET Connector Server in Legacy Mode".

For the Java connector server, the service listens on port 8759 by default, for both the new and legacy protocols. The new protocol runs by default. To run the service with the legacy protocol, you must change the main class that is executed in the `ConnectorServer.sh` or `ConnectorServer.bat` file. The class that starts the websocket protocol is `MAIN_CLASS=org.forgerock.openicf.framework.server.Main`. The class that starts the legacy protocol is `MAIN_CLASS=org.identityconnectors.framework.server.Main`. To change the port on which the Java connector server listens, change the `connectorserver.port` property in the `openicf/conf/ConnectorServer.properties` file.

`key`

string, required

The secret key, or password, to use to authenticate to the remote connector server.

To run remotely, the connector .jar itself must be copied to the `openicf/bundles` directory on the remote machine.

The following example provides a configuration for reconciling managed users with objects in a remote CSV file.

Using the CSV Connector to Reconcile Users in a Remote CSV Data Store

This example demonstrates reconciliation of users stored in a CSV file on a remote machine. The remote Java Connector Server enables OpenIDM to synchronize the internal OpenIDM repository with the remote CSV repository.

The example assumes that a remote Java Connector Server is installed on a host named `remote-host`. For instructions on setting up the remote Java Connector Server, see "Installing a Remote Java Connector Server for Unix/Linux" or "Installing a Remote Java Connector Server for Windows".

Configuring the Remote Connector Server for the CSV Connector Example

This example assumes that the Java Connector Server is running on the machine named `remote-host`. The example uses the small CSV data set provided with the *Getting Started* sample (`hr.csv`). The CSV connector runs as a *remote connector*, that is, on the remote host on which the Java Connector Server is installed. Before you start, copy the sample data file, and the CSV connector itself over to the remote machine.

1. Shut down the remote connector server, if it is running. In the connector server terminal window, type `q`:

   ```
   q
   INFO: Stopped listener bound to [0.0.0.0:8759]
   May 30, 2016 12:33:24 PM INFO  o.f.o.f.server.ConnectorServer: Server is
    shutting down org.forgerock.openicf.framework.server.ConnectorServer@171ba877
   ```

2. Copy the CSV data file from the *Getting Started* sample (`/path/to/openidm/samples/getting-started/data/hr.csv`) to an accessible location on the machine that hosts the remote Java Connector Server. For example:

   ```
   $ cd /path/to/openidm/samples/getting-started/data/
   $ scp hr.csv testuser@remote-host:/home/testuser/csv-sample/data/
   Password:********
   hr.csv     100%  651     0.6KB/s   00:00
   ```

3. Copy the CSV connector .jar from the OpenIDM installation to the `openicf/bundles` directory on the remote host:

```
$ cd path/to/openidm
$ scp connectors/csvfile-connector-1.5.1.4.jar testuser@remote-host:/path/to/openicf/bundles/
Password:********
csvfile-connector-1.5.1.4.jar    100%   40KB  39.8KB/s   00:00
```

4. The CSV connector depends on the Super CSV library, that is bundled with OpenIDM. Copy the Super CSV library `super-csv-2.4.0.jar` from the `openicf/bundle` directory to the `openicf/lib` directory on the remote server:

```
$ cd path/to/openidm
$ scp bundle/super-csv-2.4.0.jar testuser@remote-host:/path/to/openicf/lib/
Password:********
super-csv-2.4.0.jar              100%   96KB  95.8KB/s   00:00
```

5. On the remote host, restart the Connector Server so that it picks up the new CSV connector and its dependent libraries:

   ```
   $ cd /path/to/openicf
   $ bin/ConnectorServer.sh /run
   ...
   May 30, 2016 3:58:29 PM INFO  o.i.f.i.a.l.LocalConnectorInfoManagerImpl: Add ConnectorInfo
    ConnectorKey(
    bundleName=org.forgerock.openicf.connectors.csvfile-connector bundleVersion="[1.5.1.4,1.6.0.0)"
    connectorName=org.forgerock.openicf.csvfile.CSVFileConnector ) to Local Connector Info Manager from
    file:/path/to/openicf/bundles/csvfile-connector-1.5.1.4.jar
   May 30, 2016 3:58:30 PM org.glassfish.grizzly.http.server.NetworkListener start
   INFO: Started listener bound to [0.0.0.0:8759]
   May 30, 2016 3:58:30 PM org.glassfish.grizzly.http.server.HttpServer start
   INFO: [OpenICF Connector Server] Started.
   May 30, 2016 3:58:30 PM INFO  o.f.openicf.framework.server.Main: ConnectorServer
    listening on: ServerListener[0.0.0.0:8759 - plain]
   ```

   The connector server logs are noisy by default. You should, however, notice the addition of the CSV connector.

Configuring OpenIDM for the Remote CSV Connector Example

Before you start, copy the following files to your `/path/to/openidm/conf` directory:

-

  A customised mapping file required for this example.

- `/openidm/samples/provisioners/provisioner.openicf.connectorinfoprovider.json` The sample connector server configuration file.

- `/openidm/samples/provisioners/provisioner.openicf-csv.json`

  The sample connector configuration file.

1. Edit the remote connector server configuration file (`provisioner.openicf.connectorinfoprovider.json`) to match your network setup.

   The following example indicates that the Java connector server is running on the host `remote-host`, listening on the default port, and configured with a secret key of `Passw0rd`:

```
{
    "remoteConnectorServers" : [
        {
            "name" : "csv",
            "host" : "remote-host",
            "port" : 8759,
            "useSSL" : false,
            "timeout" : 0,
            "protocol" : "websocket",
            "key" : "Passw0rd"
        }
    ]
}
```

The `name` that you set in this file will be referenced in the `connectorHostRef` property of the connector configuration, in the next step.

The `key` that you specify here must match the password that you set when you installed the Java connector server.

2. Edit the CSV connector configuration file (`provisioner.openicf-csv.json`) as follows:

```
{
    "name" : "csvfile",
    "connectorRef" : {
        "connectorHostRef" : "csv",
        "bundleName" : "org.forgerock.openicf.connectors.csvfile-connector",
        "bundleVersion" : "[1.5.1.4,1.6.0.0)",
        "connectorName" : "org.forgerock.openicf.connectors.csv.CSVFileConnector"
    },
    ...
    "configurationProperties" : {
        "csvFile" : "/home/testuser/csv-sample/data/hr.csv"
    },
}
```

   - The `connectorHostRef` property indicates which remote connector server to use, and refers to the `name` property you specified in the `provisioner.openicf.connectorinfoprovider.json` file.

   - The `bundleVersion : "[1.5.1.4,1.6.0.0)",` must either be exactly the same as the version of the CSV connector that you are using or, if you specify a range, the CSV connector version must be included in this range.

   - The `csvFile` property must specify the absolute path to the CSV data file that you copied to the remote host on which the Java Connector Server is running.

3. Start OpenIDM:

```
$ cd /path/to/openidm
$ ./startup.sh
```

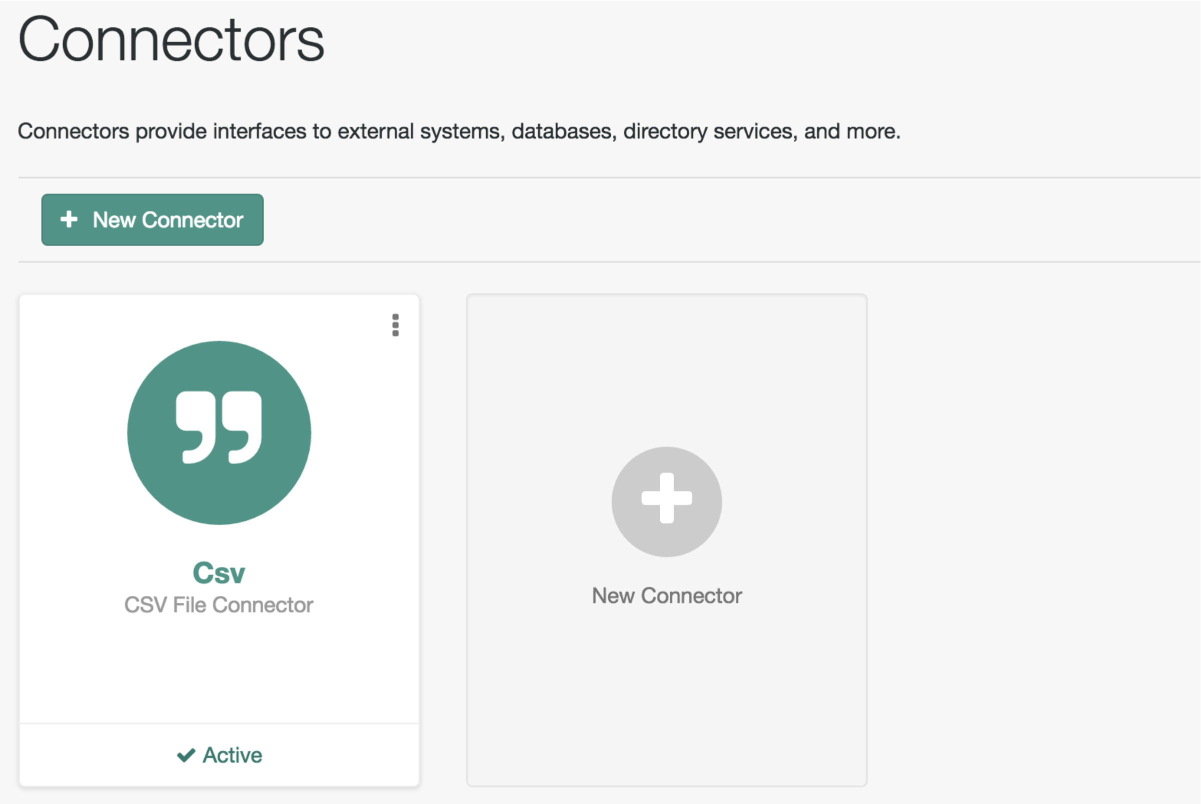
4. Verify that OpenIDM can reach the remote connector server and that the CSV connector has been configured correctly:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system?_action=test"
[
  {
    "name": "csv",
    "enabled": true,
    "config": "config/provisioner.openicf/csv",
    "objectTypes": [
      "__ALL__",
      "account"
    ],
    "connectorRef": {
      "bundleName": "org.forgerock.openicf.connectors.csvfile-connector",
      "connectorName": "org.forgerock.openicf.csvfile.CSVFileConnector",
      "bundleVersion": "[1.5.1.4,1.6.0.0)"
    },
    "displayName": "CSV File Connector",
    "ok": true
  }
]
```

The connector must return `"ok": true`.

Alternatively, use the Admin UI to verify that OpenIDM can reach the remote connector server and that the CSV connector is active. Log in to the Admin UI (`https://localhost:8443/openidm/admin`) and select Configure > Connectors. The CSV connector should be listed on the Connectors page, and its status should be Active.

Connectors Tab Showing an Active CSV Connector



Connectors

Connectors provide interfaces to external systems, databases, directory services, and more.

5. To test that the connector has been configured correctly, run a reconciliation operation as follows:

    1. Select Configure > Mappings and click the systemCsvAccounts_managedUser mapping.

    2. Click Reconcile Now.

    If the reconciliation is successful, the three users from the remote CSV file should have been added to the managed user repository.

    To check this, select Manage > User.

13.2.1. Configuring Failover Between Remote Connector Servers

To prevent the connector server from being a single point of failure, you can specify a list of remote connector servers that the connector can target. This failover configuration is included in your project's `conf/provisioner.openicf.connectorinfoprovider.json` file. The connector attempts to contact the first connector server in the list. If that connector server is down, it proceeds to the next connector server.

The following sample configuration defines two remote connector servers, on hosts `remote-host-1` and `remote-host-2`. These servers are listed, by their `name` property in a group, specified in the `remoteConnectorServersGroups` property. You can configure multiple servers per group, and multiple groups in a single remote connector server configuration file.

```
{
    "connectorsLocation" : "connectors",
    "remoteConnectorServers" : [
        {
            "name" : "dotnet1",
            "host" : "remote-host-1",
            "port" : 8759,
            "protocol" : "websocket",
            "useSSL" : false,
            "timeout" : 0,
            "key" : "password"
        },
        {
            "name" : "dotnet2",
            "host" : "remote-host-2",
            "port" : 8759,
            "protocol" : "websocket",
            "useSSL" : false,
            "timeout" : 0,
            "key" : "password"
         }
    ],
    "remoteConnectorServersGroups" : [
        {
            "name" : "dotnet-ha",
            "algorithm" : "failover",
            "serversList" : [
                {"name": "dotnet1"},
                {"name": "dotnet2"}
            ]
        }
    ]
}
```

The `algorithm` can be either `failover` or `roundrobin`. If the algorithm is `failover`, requests are always sent to the first connector server in the list, unless it is unavailable, in which case requests are sent to the next connector server in the list. If the algorithm is `roundrobin`, requests are distributed equally between the connector servers in the list, in the order in which they are received.

Your connector configuration file (`provisioner.openicf-`*`connector-name`*`.json`) references the remote connector server group, rather than a single remote connector server. For example, the following excerpt of a PowerShell connector configuration file references the `dotnet-ha` connector server group from the previous configuration:

```
{
  "connectorRef" : {
    "bundleName" : "MsPowerShell.Connector",
    "connectorName" : "Org.ForgeRock.OpenICF.Connectors.MsPowerShell.MsPowerShellConnector",
    "connectorHostRef" : "dotnet-ha",
    "bundleVersion" : "[1.4.2.0,1.5.0.0)"
  },
  ...
```

> **Note**
>
> Failover is not supported between connector servers that are running in legacy mode. Therefore, the configuration of each connector server that is part of the failover group must have the `protocol` property set to `websocket`.

13.3. Configuring Connectors

Connectors are configured through the OpenICF provisioner service. Each connector configuration is stored in a file in your project's `conf/` directory, and accessible over REST at the `openidm/conf` endpoint. Configuration files are named *`project-dir`*`/conf/provisioner.openicf-`*`name`* where *name* corresponds to the name of the connector. A number of sample connector configurations are available in the `openidm/samples/provisioners` directory. To use these connector configurations, edit the configuration files as required, and copy them to your project's `conf` directory.

If you are creating your own connector configuration files, *do not include additional dash characters (`-`) in the connector name*, as this might cause problems with the OSGi parser. For example, the name `provisioner.openicf-hrdb.json` is fine. The name `provisioner.openicf-hr-db.json` is not.

The following example shows a connector configuration for an XML file resource:

```
{
 "name"                       : "xml",
 "connectorRef"               : connector-ref-object,
 "producerBufferSize"         : integer,
 "connectorPoolingSupported"  : boolean, true/false,
 "poolConfigOption"           : pool-config-option-object,
 "operationTimeout"           : operation-timeout-object,
 "configurationProperties"    : configuration-properties-object,
 "syncFailureHandler"         : sync-failure-handler-object,
 "resultsHandlerConfig"       : results-handler-config-object,
 "objectTypes"                : object-types-object,
 "operationOptions"           : operation-options-object
}
```

The `name` property specifies the name of the system to which you are connecting. This name *must* be alphanumeric.

13.3.1. Setting the Connector Reference Properties

The following example shows a connector reference object:

```
{
  "bundleName"       : "org.forgerock.openicf.connectors.xml-connector",
  "bundleVersion"    : "[1.1.0.3,1.2.0.0)",
  "connectorName"    : "org.forgerock.openicf.connectors.xml.XMLConnector",
  "connectorHostRef" : "host"
}
```

`bundleName`

string, required

The `ConnectorBundle-Name` of the OpenICF connector.

`bundleVersion`

string, required

The `ConnectorBundle-Version` of the OpenICF connector. The value can be a single version (such as`1.4.0.0`) or a range of versions, which enables you to support multiple connector versions in a single project.

You can specify a range of versions as follows:

- `[1.1.0.0,1.4.0.0]` indicates that all connector versions from 1.1 to 1.4, inclusive, are supported.
- `[1.1.0.0,1.4.0.0)` indicates that all connector versions from 1.1 to 1.4, including 1.1 but excluding 1.4, are supported.
- `(1.1.0.0,1.4.0.0]` indicates that all connector versions from 1.1 to 1.4, excluding 1.1 but including 1.4, are supported.
- `(1.1.0.0,1.4.0.0)` indicates that all connector versions from 1.1 to 1.4, exclusive, are supported.

When a range of versions is specified, OpenIDM uses the latest connector that is available within that range. If your project requires a specific connector version, you must explicitly state the version in your connector configuration file, or constrain the range to address only the version that you need.

`connectorName`

string, required

The connector implementation class name.

`connectorHostRef`

string, optional

If the connector runs remotely, the value of this field must match the `name` field of the `RemoteConnectorServers` object in the connector server configuration file (`provisioner.openicf.connectorinfoprovider.json`). For example:

```
...
    "remoteConnectorServers" :
        [
            {
                "name" : "dotnet",
...
```

If the connector runs locally, the value of this field can be one of the following:

- If the connector .jar is installed in `openidm/connectors/`, the value must be `"#LOCAL"`. This is currently the default, and recommended location.
- If the connector .jar is installed in `openidm/bundle/` (not recommended), the value must be `"osgi:service/org.forgerock.openicf.framework.api.osgi.ConnectorManager"`.

13.3.2. Setting the Pool Configuration

The `poolConfigOption` specifies the pool configuration for poolable connectors only (connectors that have `"connectorPoolingSupported" : true`). Non-poolable connectors ignore this parameter.

The following example shows a pool configuration option object for a poolable connector:

```
{
  "maxObjects"                 : 10,
  "maxIdle"                    : 10,
  "maxWait"                    : 150000,
  "minEvictableIdleTimeMillis" : 120000,
  "minIdle"                    : 1
}
```

`maxObjects`

The maximum number of idle and active instances of the connector.

`maxIdle`

The maximum number of idle instances of the connector.

`maxWait`

The maximum time, in milliseconds, that the pool waits for an object before timing out. A value of `0` means that there is no timeout.

`minEvictableIdleTimeMillis`

The maximum time, in milliseconds, that an object can be idle before it is removed. A value of `0` means that there is no idle timeout.

`minIdle`

The minimum number of idle instances of the connector.

13.3.3. Setting the Operation Timeouts

The operation timeout property enables you to configure timeout values per operation type. By default, no timeout is configured for any operation type. A sample configuration follows:

```
{
  "CREATE"              : -1,
  "TEST"                : -1,
  "AUTHENTICATE"        : -1,
  "SEARCH"              : -1,
  "VALIDATE"            : -1,
  "GET"                 : -1,
  "UPDATE"              : -1,
  "DELETE"              : -1,
  "SCRIPT_ON_CONNECTOR" : -1,
  "SCRIPT_ON_RESOURCE"  : -1,
  "SYNC"                : -1,
  "SCHEMA"              : -1
}
```

operation-name

Timeout in milliseconds

A value of `-1` disables the timeout.

13.3.4. Setting the Connection Configuration

The `configurationProperties` object specifies the configuration for the connection between the connector and the resource, and is therefore resource specific.

The following example shows a configuration properties object for the default XML sample resource connector:

```
"configurationProperties" : {
    "xsdIcfFilePath" : "&{launcher.project.location}/data/resource-schema-1.xsd",
    "xsdFilePath" : "&{launcher.project.location}/data/resource-schema-extension.xsd",
    "xmlFilePath" : "&{launcher.project.location}/data/xmlConnectorData.xml"
}
```

property

Individual properties depend on the type of connector.

13.3.5. Setting the Synchronization Failure Configuration

The `syncFailureHandler` object specifies what should happen if a liveSync operation reports a failure for an operation. The following example shows a synchronization failure configuration:

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

`maxRetries`

positive integer or `-1`, required

The number of attempts that OpenIDM should make to process a failed modification. A value of zero indicates that failed modifications should not be reattempted. In this case, the post retry action is executed immediately when a liveSync operation fails. A value of -1 (or omitting the `maxRetries` property, or the entire `syncFailureHandler` object) indicates that failed modifications should be retried an infinite number of times. In this case, no post retry action is executed.

`postRetryAction`

string, required

The action that should be taken if the synchronization operation fails after the specified number of attempts. The post retry action can be one of the following:

- `logged-ignore` indicates that OpenIDM should ignore the failed modification, and log its occurrence.
- `dead-letter-queue` indicates that OpenIDM should save the details of the failed modification in a table in the repository (accessible over REST at `repo/synchronisation/deadLetterQueue/provisioner-name`).
- `script` specifies a custom script that should be executed when the maximum number of retries has been reached.

For more information, see "Configuring the LiveSync Retry Policy".

13.3.6. Configuring How Results Are Handled

The `resultsHandlerConfig` object specifies how OpenICF returns results. These configuration properties depend on the connector type and on the interfaces that are implemented by that connector type. For information the interfaces that each connector supports, see the *OpenICF Connector Configuration Reference*.

The following example shows a results handler configuration object:

```
{
    "enableNormalizingResultsHandler" : true,
    "enableFilteredResultsHandler" : false,
    "enableCaseInsensitiveFilter" : false,
    "enableAttributesToGetSearchResultsHandler" : false
}
```

`enableNormalizingResultsHandler`

boolean

If the connector implements the attribute normalizer interface, you can enable this interface by setting this configuration property to `true`. If the connector does not implement the attribute normalizer interface, the value of this property has no effect.

`enableFilteredResultsHandler`

boolean

If the connector uses the filtering and search capabilities of the remote connected system, you can set this property to `false`. If the connector does not use the remote system's filtering and search capabilities (for example, the CSV file connector), you *must* set this property to `true`, otherwise the connector performs an additional, case-sensitive search, which can cause problems.

`enableCaseInsensitiveFilter`

boolean

By default, the filtered results handler (described previously) is case-sensitive. If the filtered results handler is enabled, you can use this property to enable case-insensitive filtering. If you do not enable case-insensitive filtering, a search will not return results unless the case matches exactly. For example, a search for `lastName = "Jensen"` will not match a stored user with `lastName : jensen`.

`enableAttributesToGetSearchResultsHandler`

boolean

By default, OpenIDM determines which attributes should be retrieved in a search. If the `enableAttributesToGetSearchResultsHandler` property is set to `true` the OpenICF framework removes all attributes from the READ/QUERY response, except for those that are specifically requested. For performance reasons, you should set this property to `false` for local connectors and to `true` for remote connectors.

13.3.7. Specifying the Supported Object Types

The `object-types` configuration specifies the objects (user, group, and so on) that are supported by the connector. The property names set here define the `objectType` that is used in the URI. For example:

```
system/systemName/objectType
```

This configuration is based on the JSON Schema with the extensions described in the following section.

Attribute names that start or end with `__` are regarded as *special attributes* by OpenICF. The purpose of the special attributes in OpenICF is to enable someone who is developing a *new* connector to create a contract regarding how a property can be referenced, regardless of the application that is using the connector. In this way, the connector can map specific object information between an arbitrary application and the resource, without knowing how that information is referenced in the application.

These attributes have no specific meaning in the context of OpenIDM, although some of the connectors that are bundled with OpenIDM use these attributes. The generic LDAP connector, for example, can be used with OpenDJ, Active Directory, OpenLDAP, and other LDAP directories. Each of these directories might use a different attribute name to represent the same type of information. For example, Active Directory uses `unicodePassword` and OpenDJ uses `userPassword` to represent the same thing, a user's password. The LDAP connector uses the special OpenICF `__PASSWORD__` attribute to abstract that difference. In the same way, the LDAP connector maps the `__NAME__` attribute to an LDAP `dn`.

The OpenICF `__UID__` is a special case. The `__UID__` *must not* be included in the OpenIDM configuration or in any update or create operation. This attribute denotes the unique identity attribute of an object and OpenIDM always maps it to the `_id` of the object.

The following excerpt shows the configuration of an `account` object type:

```
{
  "account" :
  {
    "$schema" : "http://json-schema.org/draft-03/schema",
    "id" : "__ACCOUNT__",
    "type" : "object",
    "nativeType" : "__ACCOUNT__",
    "properties" :
    {
      "name" :
      {
        "type" : "string",
        "nativeName" : "__NAME__",
        "nativeType" : "JAVA_TYPE_PRIMITIVE_LONG",
        "flags" :
        [
          "NOT_CREATABLE",
          "NOT_UPDATEABLE",
          "NOT_READABLE",
          "NOT_RETURNED_BY_DEFAULT"
        ]
      },
      "groups" :
      {
        "type" : "array",
        "items" :
        {
          "type" : "string",
          "nativeType" : "string"
        },
        "nativeName" : "__GROUPS__",
        "nativeType" : "string",
        "flags" :
        [
          "NOT_RETURNED_BY_DEFAULT"
        ]
      },
      "givenName" : {
         "type" : "string",
         "nativeName" : "givenName",
         "nativeType" : "string"
         },
    }
  }
}
```

OpenICF supports an `__ALL__` object type that ensures that objects of every type are included in a synchronization operation. The primary purpose of this object type is to prevent synchronization errors when multiple changes affect more than one object type.

For example, imagine a deployment synchronizing two external systems. On system A, the administrator creates a user, `jdoe`, then adds the user to a group, `engineers`. When these changes are synchronized to system B, if the `__GROUPS__` object type is synchronized first, the synchronization will fail, because the group contains a user that does not yet exist on system B. Synchronizing the `__ALL__` object type ensures that user `jdoe` is created on the external system before he is added to the group `engineers`.

The `__ALL__` object type is assumed by default - you do not need to declare it in your provisioner configuration file. If it is not declared, the object type is named `__ALL__`. If you want to map a different name for this object type, declare it in your provisioner configuration. The following excerpt from a sample provisioner configuration uses the name `allobjects`:

```
"objectTypes": {
    "allobjects": {
        "$schema": "http://json-schema.org/draft-03/schema",
        "id": "__ALL__",
        "type": "object",
        "nativeType": "__ALL__"
    },
...
```

A liveSync operation invoked with no object type assumes an object type of `__ALL__`. For example, the following call invokes a liveSync operation on all defined object types in an LDAP system:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system/ldap?_action=liveSync"
```

Note

Using the `__ALL__` object type requires a mechanism to ensure the order in which synchronization changes are processed. Servers that use the `cn=changelog` mechanism to order sync changes (such as OpenDJ, Oracle DSEE, and the legacy Sun Directory Server) cannot use the `__ALL__` object type by default, and must be forced to use time stamps to order their sync changes. For these LDAP server types, set `useTimestampsForSync` to `true` in the provisioner configuration.

LDAP servers that use timestamps by default (such as Active Directory GCs and OpenLDAP) can use the `__ALL__` object type without any additional configuration. Active Directory and Active Directory LDS, which use Update Sequence Numbers, can also use the `__ALL__` object type without additional configuration.

13.3.7.1. Extending the Object Type Configuration

`nativeType`

string, optional

The native OpenICF object type.

The list of supported native object types is dependent on the resource, or on the connector. For example, an LDAP connector might have object types such as `__ACCOUNT__` and `__GROUP__`.

13.3.7.2. Extending the Property Type Configuration

`nativeType`

string, optional

The native OpenICF attribute type.

The following native types are supported:

```
JAVA_TYPE_BIGDECIMAL
JAVA_TYPE_BIGINTEGER
JAVA_TYPE_BYTE
JAVA_TYPE_BYTE_ARRAY
JAVA_TYPE_CHAR
JAVA_TYPE_CHARACTER
JAVA_TYPE_DATE
JAVA_TYPE_DOUBLE
JAVA_TYPE_FILE
JAVA_TYPE_FLOAT
JAVA_TYPE_GUARDEDBYTEARRAY
JAVA_TYPE_GUARDEDSTRING
JAVA_TYPE_INT
JAVA_TYPE_INTEGER
JAVA_TYPE_LONG
JAVA_TYPE_OBJECT
JAVA_TYPE_PRIMITIVE_BOOLEAN
JAVA_TYPE_PRIMITIVE_BYTE
JAVA_TYPE_PRIMITIVE_DOUBLE
JAVA_TYPE_PRIMITIVE_FLOAT
JAVA_TYPE_PRIMITIVE_LONG
JAVA_TYPE_STRING
```

> **Note**
>
> The `JAVA_TYPE_DATE` property is deprecated. Functionality may be removed in a future release. This property-level extension is an alias for `string`. Any dates assigned to this extension should be formatted per ISO 8601.

`nativeName`

string, optional

The native OpenICF attribute name.

`flags`

string, optional

The native OpenICF attribute flags. OpenICF supports the following attribute flags:

- `MULTIVALUED` - specifies that the property can be multivalued. This flag sets the `type` of the attribute as follows:

  ```
  "type" : "array"
  ```

  If the attribute type is `array`, an additional `items` field specifies the supported type for the objects in the array. For example:

  ```
  "groups" :
      {
          "type" : "array",
          "items" :
          {
            "type" : "string",
            "nativeType" : "string"
          },
      ....
  ```

- `NOT_CREATABLE`, `NOT_READABLE`, `NOT_RETURNED_BY_DEFAULT`, `NOT_UPDATEABLE`

  In some cases, the connector might not support manipulating an attribute because the attribute can only be changed directly on the remote system. For example, if the `name` attribute of an account can only be created by Active Directory, and *never* changed by OpenIDM, you would add `NOT_CREATABLE` and `NOT_UPDATEABLE` to the provisioner configuration for that attribute.

  Certain attributes such as LDAP groups or other calculated attributes might be expensive to read. You might want to avoid returning these attributes in a default read of the object, unless they are explicitly requested. In this case, you would add the `NOT_RETURNED_BY_DEFAULT` flag to the provisioner configuration for that attribute.

- `REQUIRED` - specifies that the property is required in create operations. This flag sets the `required` property of an attribute as follows:

  ```
  "required" : true
  ```

Note

Do not use the dash character (`-`) in property names, like `last-name`. Dashes in names make JavaScript syntax more complex. If you cannot avoid the dash, write `source['last-name']` instead of `source.last-name` in your JavaScript scripts.

13.3.8. Configuring the Operation Options

The `operationOptions` object enables you to deny specific operations on a resource. For example, you can use this configuration object to deny `CREATE` and `DELETE` operations on a read-only resource to avoid OpenIDM accidentally updating the resource during a synchronization operation.

The following example defines the options for the `"SYNC"` operation:

```
"operationOptions" : {
  {
    "SYNC" :
    {
      "denied" : true,
      "onDeny" : "DO_NOTHING",
      "objectFeatures" :
      {
        "__ACCOUNT__" :
        {
          "denied" : true,
          "onDeny" : "THROW_EXCEPTION",
          "operationOptionInfo" :
          {
            "$schema" : "http://json-schema.org/draft-03/schema",
            "id" : "FIX_ME",
            "type" : "object",
            "properties" :
            {
              "_OperationOption-float" :
              {
                 "type" : "number",
                 "nativeType" : "JAVA_TYPE_PRIMITIVE_FLOAT"
              }
            }
          }
        },
        "__GROUP__" :
        {
          "denied" : false,
          "onDeny" : "DO_NOTHING"
        }
      }
    }
  }
...
```

The OpenICF Framework supports the following operations:

- `AUTHENTICATE`

- `CREATE`
- `DELETE`
- `GET`
- `RESOLVEUSERNAME`
- `SCHEMA`
- `SCRIPT_ON_CONNECTOR`
- `SCRIPT_ON_RESOURCE`
- `SEARCH`
- `SYNC`
- `TEST`
- `UPDATE`
- `VALIDATE`

For detailed information on these operations, see the OpenICF API documentation.

The `operationOptions` object has the following configurable properties:

`denied`

boolean, optional

This property prevents operation execution if the value is `true`.

`onDeny`

string, optional

If `denied` is `true`, then the service uses this value. Default value: `DO_NOTHING`.

- `DO_NOTHING`: On operation the service does nothing.
- `THROW_EXCEPTION`: On operation the service throws a `ForbiddenException` exception.

13.4. Installing and Configuring Remote Connector Servers

Connectors that use the .NET framework *must* run remotely. Java connectors can run locally or remotely. Connectors that run remotely require a connector server to enable OpenIDM to access the connector.

For a list of supported versions, and compatibility between versions, see "Supported Connectors, Connector Servers, and Plugins" in the *Release Notes*.

This section describes the steps to install a .NET connector server and a remote Java Connector Server.

13.4.1. Installing and Configuring a .NET Connector Server

A .NET connector server is useful when an application is written in Java, but a connector bundle is written using C#. Because a Java application (for example, a J2EE application) cannot load C# classes, you must deploy the C# bundles under a .NET connector server. The Java application can communicate with the C# connector server over the network, and the C# connector server acts as a proxy to provide access to the C# bundles that are deployed within the C# connector server, to any authenticated application.

By default, the connector server outputs log messages to a file named `connectorserver.log`, in the `C:\path\to\openicf` directory. To change the location of the log file set the `initializeData` parameter in the configuration file, before you install the connector server. For example, the following excerpt sets the log directory to `C:\openicf\logs\connectorserver.log`:

```
<add name="file"
   type="System.Diagnostics.TextWriterTraceListener"
   initializeData="C:\openicf\logs\connectorserver.log"
   traceOutputOptions="DateTime">
     <filter type="System.Diagnostics.EventTypeFilter" initializeData="Information"/>
     </add>
```

Installing the .NET Connector Server

1. Download the OpenICF .NET Connector Server from the ForgeRock BackStage site.

   The .NET connector server is distributed in two formats. The `.msi` file is a wizard that installs the Connector Server as a Windows Service. The `.zip` file is simply a bundle of all the files required to run the Connector Server.

   - If you do *not* want to run the Connector Server as a Windows service, download and extract the `.zip` file, then move on to "Configuring the .NET Connector Server".

   - If you have deployed the `.zip` file and then decide to run the Connector Server as a service, install the service manually with the following command:

     ```
     .\ConnectorServerService.exe /install /serviceName service-name
     ```

     Then proceed to "Configuring the .NET Connector Server".

   - To install the Connector Server as a Windows service automatically, follow the remaining steps in this section.

2. Execute the `openicf-zip-1.5.2.0-dotnet.msi` installation file and complete the wizard.

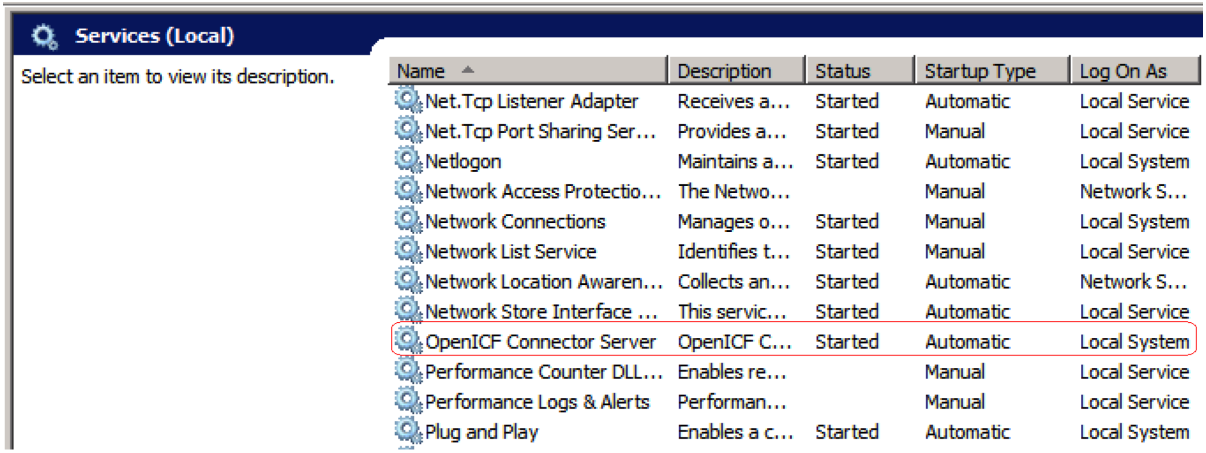
   You must run the wizard as a user who has permissions to start and stop a Windows service, otherwise the service will not start.

When you choose the Setup Type, select Typical unless you require backward compatibility with the 1.4.0.0 connector server. If you need backward compatibility, select Custom, and install the Legacy Connector Service.

When the wizard has completed, the Connector Server is installed as a Windows Service.

3. Open the Microsoft Services Console and make sure that the Connector Server is listed there.

   The name of the service is `OpenICF Connector Server`, by default.



Services (Local)

Select an item to view its description.

Name	Description	Status	Startup Type	Log On As
Net.Tcp Listener Adapter	Receives a...	Started	Automatic	Local Service
Net.Tcp Port Sharing Ser...	Provides a...	Started	Manual	Local Service
Netlogon	Maintains a...	Started	Automatic	Local System
Network Access Protectio...	The Netwo...		Manual	Network S...
Network Connections	Manages o...	Started	Manual	Local System
Network List Service	Identifies t...	Started	Manual	Local Service
Network Location Awaren...	Collects an...	Started	Automatic	Network S...
Network Store Interface ...	This servic...	Started	Automatic	Local Service
OpenICF Connector Server	OpenICF C...	Started	Automatic	Local System
Performance Counter DLL...	Enables re...		Manual	Local Service
Performance Logs & Alerts	Performan...		Manual	Local Service
Plug and Play	Enables a c...	Started	Automatic	Local System

Running the .NET Connector Server in Legacy Mode

1. If you are installing the .NET Connector Server from the `.msi` distribution, select Custom for the Setup Type, and install the Legacy Connector Service.

2. If you are installing the .NET Connector Server from the `.zip` distribution, launch the Connector Server by running the `ConnectorServer.exe` command, and *not* the `ConnectorServerService.exe` command.

3. Adjust the `port` parameter in your OpenIDM remote connector server configuration file. In legacy mode, the connector server listens on port `8760` by default.

4. Remove the `"protocol" : "websocket",` from your OpenIDM remote connector server configuration file to specify that the connector server should use the legacy protocol.

5. In the commands shown in "Configuring the .NET Connector Server", replace `ConnectorServerService.exe` with `ConnectorServer.exe`.

Configuring the .NET Connector Server

After you have installed the .NET Connector Server, as described in the previous section, follow these steps to configure the Connector Server:

1. Make sure that the Connector Server is not currently running. If it is running, use the Microsoft Services Console to stop it.

2. At the command prompt, change to the directory where the Connector Server was installed:

   ```
   c:\> cd "c:\Program Files (x86)\ForgeRock\OpenICF"
   ```

3. Run the **ConnectorServerService /setkey** command to set a secret key for the Connector Server. The key can be any string value. This example sets the secret key to `Passw0rd`:

   ```
   ConnectorServerService /setkey Passw0rd
   Key has been successfully updated.
   ```

   This key is used by clients connecting to the Connector Server. The key that you set here must also be set in the OpenIDM connector info provider configuration file (`conf/provisioner.openicf.connectorinfoprovider.json`). For more information, see "Configuring OpenIDM to Connect to the .NET Connector Server".

4. Edit the Connector Server configuration.

   The Connector Server configuration is saved in a file named `ConnectorServerService.exe.Config` (in the directory in which the Connector Server is installed).

   Check and edit this file, as necessary, to reflect your installation. Specifically, verify that the `baseAddress` reflects the host and port on which the connector server is installed:

   ```
   <system.serviceModel>
     <services>
       <service name="Org.ForgeRock.OpenICF.Framework.Service.WcfServiceLibrary.WcfWebsocket">
         <host>
           <baseAddresses>
             <add baseAddress="http://0.0.0.0:8759/openicf" />
           </baseAddresses>
         <host>
       </service>
     </services>
   </system.serviceModel>
   ```

   The `baseAddress` specifies the host and port on which the Connector Server listens, and is set to `http://0.0.0.0:8759/openicf` by default. If you set a host value other than the default `0.0.0.0`, connections from all IP addresses other than the one specified are denied.

   If Windows firewall is enabled, you must create an inbound port rule to open the TCP port for the connector server (8759 by default). If you do not open the TCP port, OpenIDM will be unable to contact the Connector Server. For more information, see the Microsoft documentation on creating an inbound port rule.

5. Optionally, configure the Connector Server to use SSL:

   a. Use an existing CA certificate, or use the `makecert` utility to create an exportable self-signed Root CA Certificate:

```
c:\"Program Files (x86)"\"Windows Kits"\8.1\bin\x64\makecert.exe
 ^
-pe -r -sky signature -cy authority -a sha1 -n "CN=Dev Certification Authority"
 ^
-ss Root -sr LocalMachine -sk RootCA signroot.cer
```

b. Create an exportable server authentication certificate:

```
c:\"Program Files (x86)"\"Windows Kits"\8.1\bin\x64\makecert.exe
 ^
-pe -sky exchange -cy end -n "CN=localhost" -b 01/01/2015 -e 01/01/2050 -eku 1.3.6.1.5.5.7.3.1
 ^
-ir LocalMachine -is Root -ic signroot.cer -ss My -sr localMachine -sk server
 ^
-sp "Microsoft RSA SChannel Cryptographic Provider" -sy 12 server.cer
```

c. Retrieve and set the certificate thumbprint:

```

        c:\Program Files (x86)\ForgeRock\OpenICF>ConnectorServerService.exe /setCertificate
Select certificate you want to use:
Index  Issued To         Thumbprint
-----  ---------         -------------------------
  0)   localhost         4D01BE385BF079DD4B9C5A416E7B535904855E0A

Certificate Thumbprint has been successfully updated to 4D01BE385BF079DD4B9C5A416E7B535904855E0A.
```

d. Bind the certificate to the Connector Server port. For example:

```
netsh http add sslcert ipport=0.0.0.0:8759 ^
certhash=4D01BE385BF079DD4B9C5A416E7B535904855E0A ^
appid={bca0631d-cab1-48c8-bd2a-eb049d7d3c55}
```

e. Execute Service as a non-administrative user:

```
netsh http add urlacl url=https://+:8759/ user=EVERYONE
```

f. Change the Connector Server configuration to use HTTPS and not HTTP:

```
<add baseAddress="https://0.0.0.0:8759/openicf" />
```

6. Check the trace settings, in the same Connector Server configuration file, under the `system.diagnostics` item:

```
<system.diagnostics>
  <trace autoflush="true" indentsize="4">
    <listeners>
      <remove name="Default" />
      <add name="console" />
      <add name="file" />
    </listeners>
  </trace>
  <sources>
    <source name="ConnectorServer" switchName="switch1">
      <listeners>
        <remove name="Default" />
        <add name="file" />
      </listeners>
    </source>
  </sources>
  <switches>
    <add name="switch1" value="Information" />
  </switches>
  <sharedListeners>
    <add name="console" type="System.Diagnostics.ConsoleTraceListener" />
    <add name="file" type="System.Diagnostics.TextWriterTraceListener"
            initializeData="logs\ConnectorServerService.log"
            traceOutputOptions="DateTime">
        <filter type="System.Diagnostics.EventTypeFilter" initializeData="Information" />
    </add>
  </sharedListeners>
</system.diagnostics>
```

The Connector Server uses the standard .NET trace mechanism. For more information about tracing options, see Microsoft's .NET documentation for `System.Diagnostics`.

The default trace settings are a good starting point. For less tracing, set the EventTypeFilter's `initializeData` to `Warning` or `Error`. For very verbose logging set the value to `Verbose` or `All`. The logging level has a direct effect on the performance of the Connector Servers, so take care when setting this level.

Starting the .NET Connector Server

Start the .NET Connector Server in one of the following ways:

1. Start the server as a Windows service, by using the Microsoft Services Console.

   Locate the connector server service (`OpenICF Connector Server`), and click `Start the service` or `Restart the service`.

   The service is executed with the credentials of the "run as" user (`System`, by default).

2. Start the server as a Windows service, by using the command line.

   In the Windows Command Prompt, run the following command:

   ```
   net start ConnectorServerService
   ```

To stop the service in this manner, run the following command:

```
net stop ConnectorServerService
```

3. Start the server without using Windows services.

   In the Windows Command Prompt, change directory to the location where the Connector Server was installed. The default location is `c:\> cd "c:\Program Files (x86)\ForgeRock\OpenICF"`.

   Start the server with the following command:

   ```
   ConnectorServerService.exe /run
   ```

   Note that this command starts the Connector Server with the credentials of the current user. It does not start the server as a Windows service.

Configuring OpenIDM to Connect to the .NET Connector Server

The connector info provider service configures one or more remote connector servers to which OpenIDM can connect. The connector info provider configuration is stored in a file named `project-dir/conf/provisioner.openicf.connectorinfoprovider.json`. A sample connector info provider configuration file is located in `openidm/samples/provisioners/`.

To configure OpenIDM to use the remote .NET connector server, follow these steps:

1. Start OpenIDM, if it is not already running.

2. Copy the sample connector info provider configuration file to your project's `conf/` directory:

   ```
   $ cd /path/to/openidm
   $ cp samples/provisioners/provisioner.openicf.connectorinfoprovider.json project-dir/conf/
   ```

3. Edit the connector info provider configuration, specifying the details of the remote connector server:

   ```
   "remoteConnectorServers" : [
       {
           "name" : "dotnet",
           "host" : "192.0.2.0",
           "port" : 8759,
           "useSSL" : false,
           "timeout" : 0,
           "protocol" : "websocket",
           "key" : "Passw0rd"
       }
   ```

   Configurable properties are as follows:

`name`

Specifies the name of the connection to the .NET connector server. The name can be any string. This name is referenced in the `connectorHostRef` property of the connector configuration file (`provisioner.openicf-ad.json`).

`host`

Specifies the IP address of the host on which the Connector Server is installed.

`port`

Specifies the port on which the Connector Server listens. This property matches the `connectorserver.port` property in the `ConnectorServerService.exe.config` file.

For more information, see "Configuring the .NET Connector Server".

`useSSL`

Specifies whether the connection to the Connector Server should be secured. This property matches the `"connectorserver.usessl"` property in the `ConnectorServerService.exe.config` file.

`timeout`

Specifies the length of time, in seconds, that OpenIDM should attempt to connect to the Connector Server before abandoning the attempt. To disable the timeout, set the value of this property to `0`.

`protocol`

Version 1.5.2.0 of the OpenICF framework supports a new communication protocol with remote connector servers. This protocol is enabled by default, and its value is `websocket` in the default configuration.

`key`

Specifies the connector server key. This property matches the `key` property in the `ConnectorServerService.exe.config` file. For more information, see "Configuring the .NET Connector Server".

The string value that you enter here is encrypted as soon as the file is saved.

13.4.2. Installing and Configuring a Remote Java Connector Server

In certain situations, it might be necessary to set up a remote Java Connector Server. This section provides instructions for setting up a remote Java Connector Server on Unix/Linux and Windows.

Installing a Remote Java Connector Server for Unix/Linux

1. Download the OpenICF Java Connector Server from ForgeRock's BackStage site.

2. Change to the appropriate directory and unpack the zip file. The following command unzips the file in the current directory:

   ```
   $ unzip openicf-zip-1.5.2.0.zip
   ```

3. Change to the `openicf` directory:

   ```
   $ cd path/to/openicf
   ```

4. The Java Connector Server uses a `key` property to authenticate the connection. The default key value is `changeit`. To change the value of the secret key, run a command similar to the following. This example sets the key value to `Passw0rd`:

   ```
   $ cd /path/to/openicf
   $  bin/ConnectorServer.sh /setkey Passw0rd
   Key has been successfully updated.
   ```

5. Review the `ConnectorServer.properties` file in the `/path/to/openicf/conf` directory, and make any required changes. By default, the configuration file has the following properties:

   ```
   connectorserver.port=8759
   connectorserver.libDir=lib
   connectorserver.usessl=false
   connectorserver.bundleDir=bundles
   connectorserver.loggerClass=org.forgerock.openicf.common.logging.slf4j.SLF4JLog
   connectorserver.key=xOS4IeeE6eb/AhMbhxZEC37PgtE\=
   ```

   The `connectorserver.usessl` parameter indicates whether client connections to the connector server should be over SSL. This property is set to `false` by default.

   To secure connections to the connector server, set this property to `true` and set the following properties before you start the connector server:

   ```
   java -Djavax.net.ssl.keyStore=mySrvKeystore -Djavax.net.ssl.keyStorePassword=Passw0rd
   ```

6. Start the Java Connector Server:

   ```
   $ bin/ConnectorServer.sh /run
   ```

   The connector server is now running, and listening on port 8759, by default.

   Log files are available in the `/path/to/openicf/logs` directory.

   ```
   $ ls logs/
   Connector.log  ConnectorServer.log  ConnectorServerTrace.log
   ```

7. If required, stop the Java Connector Server by pressing CTRL-C.

Installing a Remote Java Connector Server for Windows

1. Download the OpenICF Java Connector Server from ForgeRock's BackStage site.
2. Change to the appropriate directory and unpack the zip file.
3. In a Command Prompt window, change to the `openicf` directory:

   ```
   C:\>cd C:\path\to\openicf\bin
   ```

4. If required, secure the communication between OpenIDM and the Java Connector Server. The Java Connector Server uses a `key` property to authenticate the connection. The default key value is `changeit`.

   To change the value of the secret key, use the `bin\ConnectorServer.bat /setkey` command. The following example sets the key to `Passw0rd`:

   ```
   c:\path\to\openicf>bin\ConnectorServer.bat /setkey Passw0rd
   lib\framework\connector-framework.jar;lib\framework\connector-framework-
   internal
   .jar;lib\framework\groovy-all.jar;lib\framework\icfl-over-slf4j.jar;lib\framework
   \slf4j-api.jar;lib\framework\logback-core.jar;lib\framework\logback-classic.jar
   ```

5. Review the `ConnectorServer.properties` file in the `path\to\openicf\conf` directory, and make any required changes. By default, the configuration file has the following properties:

   ```
   connectorserver.port=8759
   connectorserver.libDir=lib
   connectorserver.usessl=false
   connectorserver.bundleDir=bundles
   connectorserver.loggerClass=org.forgerock.openicf.common.logging.slf4j.SLF4JLog
   connectorserver.key=xOS4IeeE6eb/AhMbhxZEC37PgtE\=
   ```

6. You can either run the Java Connector Server as a Windows service, or start and stop it from the command-line.

   - To install the Java Connector Server as a Windows service, run the following command:

     ```
     c:\path\to\openicf>bin\ConnectorServer.bat /install
     ```

     If you install the connector server as a Windows service you can use the Microsoft Services Console to start, stop and restart the service. The Java Connector Service is named `OpenICFConnectorServerJava`.

     To uninstall the Java Connector Server as a Windows service, run the following command:

     ```
     c:\path\to\openicf>bin\ConnectorServer.bat /uninstall
     ```

7. To start the Java Connector Server from the command line, enter the following command:

   ```
   c:\path\to\openicf>bin\ConnectorServer.bat /run
   ```

   The connector server is now running, and listening on port 8759, by default.

Log files are available in the `\path\to\openicf\logs` directory.

8. If required, stop the Java Connector Server by pressing `^C`.

13.5. Supported Connectors

OpenIDM provides several connectors by default, in the `path/to/openidm/connectors` directory. You can download the connectors that are not bundled with OpenIDM from ForgeRock's BackStage site.

For details about the connectors that are supported for use with OpenIDM 5, see Connectors Guide.

13.6. Creating Default Connector Configurations

You have three ways to create provisioner files:

- Start with the sample provisioner files in the `/path/to/openidm/samples/provisioners` directory. For more information, see "Supported Connectors".
- Set up connectors with the help of the Admin UI. To start this process, navigate to `https://localhost:8443/admin` and log in to OpenIDM. Continue with "Adding New Connectors from the Admin UI".
- Use the service that OpenIDM exposes through the REST interface to create basic connector configuration files, or use the **cli.sh** or **cli.bat** scripts to generate a basic connector configuration. To see how this works continue with "Adding New Connectors from the Command Line".

13.6.1. Adding New Connectors from the Admin UI

You can include several different connectors in an OpenIDM configuration. In the Admin UI, select Configure > Connector. Try some of the different connector types in the screen that appears. Observe as the Admin UI changes the configuration options to match the requirements of the connector type.

The list of connectors shown in the Admin UI does not include all supported connectors. For information and examples of how each supported connector is configured, see "Supported Connectors".

When you have filled in all required text boxes, the Admin UI allows you to validate the connector configuration.

If you want to configure a different connector through the Admin UI, you could copy the provisioner file from the `/path/to/openidm/samples/provisioners` directory. However, additional configuration may be required, as described in "Supported Connectors".

Alternatively, some connectors are included with the configuration of a specific sample. For example, if you want to build a ScriptedSQL connector, read "Using the Connector Bundler to Build a ScriptedSQL Connector" in the *Samples Guide*.

13.6.2. Adding New Connectors from the Command Line

This section describes how to create connector configurations over the REST interface. For instructions on how to create connector configurations from the command line, see "Using the **configureconnector** Subcommand".

You create a new connector configuration file in three stages:

1. List the available connectors.
2. Generate the core configuration.
3. Connect to the target system and generate the final configuration.

List the available connectors by using the following command:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system?_action=availableConnectors"
```

Available connectors are installed in `openidm/connectors`. OpenIDM bundles the following connectors:

- CSV File Connector
- Database Table Connector
- Scripted Groovy Connector Toolkit, which includes the following sample implementations:
  - Scripted SQL Connector
  - Scripted CREST Connector
  - Scripted REST Connector
- LDAP Connector
- XML Connector
- GoogleApps Connector
- Salesforce Connector

The preceding command therefore returns the following output:

```
{
  "connectorRef": [
    {
      "connectorName": "org.forgerock.openicf.connectors.xml.XMLConnector",
      "displayName": "XML Connector",
      "bundleName": "org.forgerock.openicf.connectors.xml-connector",
      "systemType": "provisioner.openicf",
      "bundleVersion": "1.1.0.3"
    },
    {
      "connectorName": "org.identityconnectors.ldap.LdapConnector",
      "displayName": "LDAP Connector",
      "bundleName": "org.forgerock.openicf.connectors.ldap-connector",
      "systemType": "provisioner.openicf",
      "bundleVersion": "1.4.3.0"
    },
    {
      "connectorName": "org.forgerock.openicf.connectors.scriptedsql.ScriptedSQLConnector",
      "displayName": "Scripted SQL Connector",
      "bundleName": "org.forgerock.openicf.connectors.groovy-connector",
      "systemType": "provisioner.openicf",
      "bundleVersion": "1.4.3.0"
    },
    {
      "connectorName": "org.forgerock.openicf.connectors.scriptedrest.ScriptedRESTConnector",
      "displayName": "Scripted REST Connector",
      "bundleName": "org.forgerock.openicf.connectors.groovy-connector",
      "systemType": "provisioner.openicf",
      "bundleVersion": "1.4.3.0"
    },
    {
      "connectorName": "org.forgerock.openicf.connectors.scriptedcrest.ScriptedCRESTConnector",
      "displayName": "Scripted CREST Connector",
      "bundleName": "org.forgerock.openicf.connectors.groovy-connector",
      "systemType": "provisioner.openicf",
      "bundleVersion": "1.4.3.0"
    },
    {
      "connectorName": "org.forgerock.openicf.connectors.groovy.ScriptedPoolableConnector",
      "displayName": "Scripted Poolable Groovy Connector",
      "bundleName": "org.forgerock.openicf.connectors.groovy-connector",
      "systemType": "provisioner.openicf",
      "bundleVersion": "1.4.3.0"
    },
    {
      "connectorName": "org.forgerock.openicf.connectors.groovy.ScriptedConnector",
      "displayName": "Scripted Groovy Connector",
      "bundleName": "org.forgerock.openicf.connectors.groovy-connector",
      "systemType": "provisioner.openicf",
      "bundleVersion": "1.4.3.0"
    },
    {
      "connectorName": "org.identityconnectors.databasetable.DatabaseTableConnector",
      "displayName": "Database Table Connector",
      "bundleName": "org.forgerock.openicf.connectors.databasetable-connector",
      "systemType": "provisioner.openicf",
      "bundleVersion": "1.1.0.2"
    },
    {
      "connectorName": "org.forgerock.openicf.csvfile.CSVFileConnector",
      "displayName": "CSV File Connector",
      "bundleName": "org.forgerock.openicf.connectors.csvfile-connector",
      "systemType": "provisioner.openicf",
      "bundleVersion": "1.5.1.4"
    }
  ]
}
```

To generate the core configuration, choose one of the available connectors by copying one of the JSON objects from the generated list into the body of the REST command, as shown in the following command for the XML connector:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--header "Content-Type: application/json"
 \
--request POST
 \
--data '{"connectorRef":
    {"connectorName": "org.forgerock.openicf.connectors.xml.XMLConnector",
    "displayName": "XML Connector",
    "bundleName": "org.forgerock.openicf.connectors.xml-connector",
    "bundleVersion": "[1.1.0.3,1.2.0.0)"}
 }' \
 "http//localhost:8080/openidm/system?_action=createCoreConfig"
```

This command returns a core connector configuration, similar to the following:

```
{
    "poolConfigOption": {
    "minIdle": 1,
    "minEvictableIdleTimeMillis": 120000,
    "maxWait": 150000,
    "maxIdle": 10,
    "maxObjects": 10
  },
    "resultsHandlerConfig": {
    "enableAttributesToGetSearchResultsHandler": true,
    "enableFilteredResultsHandler": true,
    "enableNormalizingResultsHandler": true
  },
  "operationTimeout": {
    "SCHEMA": -1,
    "SYNC": -1,
    "VALIDATE": -1,
    "SEARCH": -1,
    "AUTHENTICATE": -1,
    "CREATE": -1,
    "UPDATE": -1,
    "DELETE": -1,
    "TEST": -1,
    "SCRIPT_ON_CONNECTOR": -1,
    "SCRIPT_ON_RESOURCE": -1,
    "GET": -1,
    "RESOLVEUSERNAME": -1
  },
  "configurationProperties": {
    "xsdIcfFilePath": null,
    "xsdFilePath": null,
    "createFileIfNotExists": false,
    "xmlFilePath": null
  },
  "connectorRef": {
    "bundleVersion": "[1.1.0.3,1.2.0.0)",
    "bundleName": "org.forgerock.openicf.connectors.xml-connector",
    "displayName": "XML Connector",
    "connectorName": "org.forgerock.openicf.connectors.xml.XMLConnector"
  }
}
```

The configuration that is returned is not yet functional. Notice that it does not contain the required system-specific `configurationProperties`, such as the host name and port, or the `xmlFilePath` for the XML file-based connector. In addition, the configuration does not include the complete list of `objectTypes` and `operationOptions`.

To generate the final configuration, add values for the `configurationProperties` to the core configuration, and use the updated configuration as the body for the next command:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--header "Content-Type: application/json"
 \
--request POST
 \
--data '{
  "configurationProperties":
    {
      "xsdIcfFilePath" : "samples/sample1/data/resource-schema-1.xsd",
      "xsdFilePath" : "samples/sample1/data/resource-schema-extension.xsd",
      "xmlFilePath" : "samples/sample1/data/xmlConnectorData.xml",
      "createFileIfNotExists": false
    },
    "operationTimeout": {
      "SCHEMA": -1,
      "SYNC": -1,
      "VALIDATE": -1,
      "SEARCH": -1,
      "AUTHENTICATE": -1,
      "CREATE": -1,
      "UPDATE": -1,
      "DELETE": -1,
      "TEST": -1,
      "SCRIPT_ON_CONNECTOR": -1,
      "SCRIPT_ON_RESOURCE": -1,
      "GET": -1,
      "RESOLVEUSERNAME": -1
    },
    "resultsHandlerConfig": {
      "enableAttributesToGetSearchResultsHandler": true,
      "enableFilteredResultsHandler": true,
      "enableNormalizingResultsHandler": true
    },
    "poolConfigOption": {
      "minIdle": 1,
      "minEvictableIdleTimeMillis": 120000,
      "maxWait": 150000,
      "maxIdle": 10,
      "maxObjects": 10
    },
    "connectorRef": {
      "bundleVersion": "[1.1.0.3,1.2.0.0)",
      "bundleName": "org.forgerock.openicf.connectors.xml-connector",
      "displayName": "XML Connector",
      "connectorName": "org.forgerock.openicf.connectors.xml.XMLConnector"
    }
  }' \
"http://localhost:8080/openidm/system?_action=createFullConfig"
```

Note

Notice the single quotes around the argument to the `--data` option in the preceding command. For most UNIX shells, single quotes around a string prevent the shell from executing the command when encountering a new line in the content. You can therefore pass the `--data '...'` option on a single line, or including line feeds.

OpenIDM attempts to read the schema, if available, from the external resource in order to generate output. OpenIDM then iterates through schema objects and attributes, creating JSON representations for `objectTypes` and `operationOptions` for supported objects and operations.

The output includes the basic `--data` input, along with `operationOptions` and `objectTypes`.

Because OpenIDM produces a full property set for all attributes and all object types in the schema from the external resource, the resulting configuration can be large. For an LDAP server, OpenIDM can generate a configuration containing several tens of thousands of lines, for example. You might therefore want to reduce the schema to a minimum on the external resource before you run the `createFullConfig` command.

When you have the complete connector configuration, save that configuration in a file named `provisioner.openicf-name.json` (where name corresponds to the name of the connector) and place it in the `conf` directory of your project. For more information, see "Configuring Connectors".

13.7. Checking the Status of External Systems Over REST

After a connection has been configured, external systems are accessible over the REST interface at the URL `http://localhost:8080/openidm/system/connector-name`. Aside from accessing the data objects within the external systems, you can test the availability of the systems themselves.

To list the external systems that are connected to an OpenIDM instance, use the `test` action on the URL `http://localhost:8080/openidm/system/`. The following example shows the connector configuration for an external LDAP system:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system?_action=test"
[
  {
    "ok": true,
    "displayName": "LDAP Connector",
    "connectorRef": {
      "bundleVersion": "[1.4.0.0,2.0.0.0)",
      "bundleName": "org.forgerock.openicf.connectors.ldap-connector",
      "connectorName": "org.identityconnectors.ldap.LdapConnector"
    },
    "objectTypes": [
      "__ALL__",
      "group",
      "account"
    ],
    "config": "config/provisioner.openicf/ldap",
    "enabled": true,
    "name": "ldap"
  }
]
```

The status of the system is provided by the `ok` parameter. If the connection is available, the value of this parameter is `true`.

To obtain the status for a single system, include the name of the connector in the URL, for example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system/ldap?_action=test"
{
  "ok": true,
  "displayName": "LDAP Connector",
  "connectorRef": {
    "bundleVersion": "[1.4.0.0,2.0.0.0)",
    "bundleName": "org.forgerock.openicf.connectors.ldap-connector",
    "connectorName": "org.identityconnectors.ldap.LdapConnector"
  },
  "objectTypes": [
    "__ALL__",
    "group",
    "account"
  ],
  "config": "config/provisioner.openicf/ldap",
  "enabled": true,
  "name": "ldap"
}
```

If there is a problem with the connection, the `ok` parameter returns `false`, with an indication of the error. In the following example, the LDAP server named `ldap`, running on `localhost:1389`, is down:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system/ldap?_action=test"
{
  "ok": false,
  "error": "localhost:1389",
  "displayName": "LDAP Connector",
  "connectorRef": {
    "bundleVersion": "[1.4.0.0,2.0.0.0)",
    "bundleName": "org.forgerock.openicf.connectors.ldap-connector",
    "connectorName": "org.identityconnectors.ldap.LdapConnector"
  },
  "objectTypes": [
    "__ALL__",
    "group",
    "account"
  ],
  "config": "config/provisioner.openicf/ldap",
  "enabled": true,
  "name": "ldap"
}
```

To test the validity of a connector configuration, use the `testConfig` action and include the configuration in the command. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --data '{
    "name" : "xmlfile",
    "connectorRef" : {
        "bundleName" : "org.forgerock.openicf.connectors.xml-connector",
        "bundleVersion" : "[1.1.0.3,1.2.0.0)",
        "connectorName" : "org.forgerock.openicf.connectors.xml.XMLConnector"
    },
    "producerBufferSize" : 100,
    "connectorPoolingSupported" : true,
    "poolConfigOption" : {
        "maxObjects" : 10,
        "maxIdle" : 10,
        "maxWait" : 150000,
        "minEvictableIdleTimeMillis" : 120000,
        "minIdle" : 1
    },
    "operationTimeout" : {
        "CREATE" : -1,
        "TEST" : -1,
        "AUTHENTICATE" : -1,
        "SEARCH" : -1,
        "VALIDATE" : -1,
        "GET" : -1,
        "UPDATE" : -1,
        "DELETE" : -1,
        "SCRIPT_ON_CONNECTOR" : -1,
        "SCRIPT_ON_RESOURCE" : -1,
        "SYNC" : -1,
        "SCHEMA" : -1
    },
    "configurationProperties" : {
        "xsdIcfFilePath" : "samples/sample1/data/resource-schema-1.xsd",
        "xsdFilePath" : "samples/sample1/data/resource-schema-extension.xsd",
        "xmlFilePath" : "samples/sample1/data/xmlConnectorData.xml"
    },
    "syncFailureHandler" : {
        "maxRetries" : 5,
        "postRetryAction" : "logged-ignore"
    },
    "objectTypes" : {
        "account" : {
            "$schema" : "http://json-schema.org/draft-03/schema",
            "id" : "__ACCOUNT__",
            "type" : "object",
            "nativeType" : "__ACCOUNT__",
            "properties" : {
                "description" : {
                    "type" : "string",
                    "nativeName" : "__DESCRIPTION__",
                    "nativeType" : "string"
                },
                "firstname" : {
                    "type" : "string",
                    "nativeName" : "firstname",
                    "nativeType" : "string"
                },
                "email" : {
                    "type" : "string",
                    "nativeName" : "email",
                    "nativeType" : "string"
                },
                "_id" : {
                    "type" : "string",
                    "nativeName" : "__UID__"
                },
                "password" : {
                    "type" : "string",
                    "nativeName" : "password",
                    "nativeType" : "string"
                },
                "name" : {
                    "type" : "string",
                    "required" : true,
                    "nativeName" : "__NAME__",
                    "nativeType" : "string"
                },
                "lastname" : {
                    "type" : "string",
                    "required" : true,
                    "nativeName" : "lastname",
                    "nativeType" : "string"
                },
                "mobileTelephoneNumber" : {
                    "type" : "string",
                    "required" : true,
                    "nativeName" : "mobileTelephoneNumber",
                    "nativeType" : "string"
                },
                "securityQuestion" : {
                    "type" : "string",
                    "required" : true,
                    "nativeName" : "securityQuestion",
                    "nativeType" : "string"
                },
                "securityAnswer" : {
                    "type" : "string",
                    "required" : true,
                    "nativeName" : "securityAnswer",
                    "nativeType" : "string"
                },
                "roles" : {
                    "type" : "string",
                    "required" : false,
                    "nativeName" : "roles",
                    "nativeType" : "string"
                }
            }
        }
    },
    "operationOptions" : { }
}' \
 --request POST \
 "http://localhost:8080/openidm/system?_action=testConfig"
```

If the configuration is valid, the command returns `"ok": true`, for example:

```
{
   "ok": true,
   "name": "xmlfile"
}
```

If the configuration is not valid, the command returns an error, indicating the problem with the configuration. For example, the following result is returned when the LDAP connector configuration is missing a required property (in this case, the `baseContexts` to synchronize):

```
{
  "error": "org.identityconnectors.framework.common.exceptions.ConfigurationException:
           The list of base contexts cannot be empty",
  "name": "OpenDJ",
  "ok": false
}
```

The `testConfig` action requires a running OpenIDM instance, as it uses the REST API, but does not require an active connector instance for the connector whose configuration you want to test.

13.8. Adding Attributes to Connector Configurations

You can add the attributes of your choice to a connector configuration file. Specifically, if you want to set up "Extending the Property Type Configuration" to one of the `objectTypes` such as `account`, use the format shown under "Specifying the Supported Object Types".

You can configure connectors to enable provisioning of arbitrary property level extensions (such as image files) to system resources. For example, if you want to set up image files such as account avatars, open the appropriate provisioner file. Look for an `account` section similar to:

```
"account" : {
    "$schema" : "http://json-schema.org/draft-03/schema",
    "id" : "__ACCOUNT__",
    "type" : "object",
    "nativeType" : "__ACCOUNT__",
    "properties" : {...
```

Under `properties`, add one of the following code blocks. The first block works for a single photo encoded as a base64 string. The second block would address multiple photos encoded in the same way:

```
"attributeByteArray" : {
    "type" : "string",
    "nativeName" : "attributeByteArray",
    "nativeType" : "JAVA_TYPE_BYTE_ARRAY"
},
```

```
"attributeByteArrayMultivalue": {
    "type": "array",
    "items": {
        "type": "string",
        "nativeType": "JAVA_TYPE_BYTE_ARRAY"
    },
    "nativeName": "attributeByteArrayMultivalue"
},
```

Chapter 14
Synchronizing Data Between Resources

One of the core services of OpenIDM is synchronizing identity data between different resources. In this chapter, you will learn about the different types of synchronization, and how to configure OpenIDM's flexible synchronization mechanism.

14.1. Types of Synchronization

Synchronization happens either when OpenIDM receives a change directly, or when OpenIDM discovers a change on an external resource. An *external resource* can be any system that holds identity data, such as Active Directory, OpenDJ, a CSV file, a JDBC database, and others. OpenIDM connects to external resources by using OpenICF connectors. For more information, see "*Connecting to External Resources*".

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

- OpenIDM discovers and synchronizes changes from external resources by using *reconciliation* and *liveSync*.
- OpenIDM synchronizes changes made to its internal repository with external resources by using *implicit synchronization*.

Reconciliation

> *Reconciliation* is the process of ensuring that the objects in two different data stores are synchronized. Traditionally, reconciliation applies mainly to user objects, but OpenIDM can reconcile any objects, such as groups, roles, and devices.
>
> In any reconciliation operation, there is a *source system* (the system that contains the changes) and a *target system* (the system to which the changes will be propagated). The source and target system are defined in a *mapping*. OpenIDM can be either the source or the target in a mapping. You can configure multiple mappings for one OpenIDM instance, depending on the external resources to which OpenIDM connects.
>
> To perform reconciliation, OpenIDM analyzes both the source system *and* the target system, to discover the differences that it must reconcile. Reconciliation can therefore be a heavyweight

process. When working with large data sets, finding all changes can be more work than processing the changes.

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

LiveSync

LiveSync captures the changes that occur on a remote system, then pushes those changes to OpenIDM. OpenIDM uses the defined mappings to replay the changes where they are required; either in the OpenIDM repository, or on another remote system, or both. Unlike reconciliation, liveSync uses a polling system, and is intended to react quickly to changes as they happen.

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

> **Note**
>
> In the case of OpenDJ, the change log (`cn=changelog`) can be read only by `cn=directory manager` by default. If you are configuring liveSync with OpenDJ, the `principal` that is defined in the LDAP connector configuration must have access to the change log. For information about allowing a regular user to read the change log, see To Allow a User to Read the Change Log in the *Administration Guide* for OpenDJ.

Implicit synchronization

Implicit synchronization automatically pushes changes that are made in the OpenIDM internal repository to external systems.

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

OpenIDM uses mappings, configured in your project's `conf/sync.json` file, to determine which data to synchronize, and how that data must be synchronized. You can schedule reconciliation operations, and the frequency with which OpenIDM polls for liveSync changes, as described in "*Scheduling Tasks and Events*".

OpenIDM logs reconciliation and synchronization operations in the audit logs by default. For information about querying the reconciliation and synchronization logs, see "Querying Audit Logs Over REST".

14.2. Defining Your Data Mapping Model

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

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

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

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

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

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

You can, for example, choose to follow the data model defined in the Simple Cloud Identity Management (SCIM) specification. The following object represents a SCIM user:

```
{
    "userName": "james1",
    "familyName": "Berg",
    "givenName": "James",
    "email": [
        "james1@example.com"
    ],
    "description": "Created by OpenIDM REST.",
    "password": "asdfkj23",
    "displayName": "James Berg",
    "phoneNumber": "12345",
    "employeeNumber": "12345",
    "userType": "Contractor",
    "title": "Vice President",
    "active": true
}
```

Note

Avoid using the dash character (`-`) in property names, like `last-name`, as dashes in names make JavaScript syntax more complex. If you cannot avoid the dash, then write `source['last-name']` instead of `source.last-name` in your JavaScript.

14.3. Configuring Synchronization Between Two Resources

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

1. Set up the connector configuration.

   Connector configurations are defined in `conf/provisioner-*.json` files. One provisioner file must be defined for each external resource to which you are connecting.

2. Map source objects to target objects.

   Mappings are defined in the `conf/sync.json` file. There is only one `sync.json` file per OpenIDM instance, but multiple mappings can be defined in that file.

3. Configure any scripts that are required to check source and target objects, and to manipulate attributes.

4. In addition to these configuration elements, OpenIDM stores a `links` table in its repository. The links table maintains a record of relationships established between source and target objects.

14.3.1. Setting Up the Connector Configuration

Connector configuration files map external resource objects to OpenIDM objects, and are described in detail in "*Connecting to External Resources*". Connector configuration files are stored in the `conf/` directory of your project, and are named `provisioner.resource-name.json`, where *resource-name* reflects the connector technology and the external resource, for example, `openicf-xml`.

You can create and modify connector configurations through the Admin UI or directly in the configuration files, as described in the following sections.

14.3.1.1. Setting up and Modifying Connector Configurations in the Admin UI

The easiest way to set up and modify connector configurations is to use the Admin UI.

To add or modify a connector configuration in the Admin UI:

1. Log in to the UI (`http://localhost:8080/admin`) as an administrative user. The default administrative username and password is `openidm-admin` and `openidm-admin`.

2. Select Configure > Connectors.

3. Click on the connector that you want to modify (if there is an existing connector configuration) or click New Connector to set up a new connector configuration.

14.3.1.2. Editing Connector Configuration Files

A number of sample provisioner files are provided in `path/to/openidm/samples/provisioners`. To modify connector configuration files directly, edit one of the sample provisioner files that corresponds to the resource to which you are connecting.

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

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

For OpenIDM to access external resource objects and attributes, the object and its attributes must match the connector configuration. Note that the connector file only maps external resource objects to OpenIDM objects. To construct attributes and to manipulate their values, you use the synchronization mappings file, described in the following section.

14.3.2. Mapping Source Objects to Target Objects

A synchronization mapping specifies a relationship between objects and their attributes in two data stores. A typical attribute mapping, between objects in an external LDAP directory and an internal Managed User data store, is:

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

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

The core configuration for OpenIDM synchronization is defined in your project's synchronization mappings file (`conf/sync.json`). The mappings file contains one or more mappings for every resource that must be synchronized.

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

- LDAP Server > Local Repository

- Local Repository > LDAP Server

With bidirectional synchronization, OpenIDM includes a `links` property that enables you to reuse the links established between objects, for both mappings. For more information, see "Reusing Links Between Mappings".

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

14.3.2.1. Specifying the Resource Mapping

Objects in external resources are specified in a mapping as `system/name/object-type`, where *name* is the name used in the connector configuration file, and *object-type* is the object defined in the connector configuration file list of object types. Objects in OpenIDM's internal repository are specified in the mapping as `managed/object-type`, where *object-type* is defined in your project's managed objects configuration file (`conf/managed.json`).

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

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

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

14.3.2.1.1. Specifying Resource Mapping in the Admin UI

You can also configure synchronization mappings in the Admin UI. To do so, navigate to `http://localhost:8080/admin`, and click Configure > Mappings. The Admin UI serves as a front end to OpenIDM configuration files, so, the changes you make to mappings in the Admin UI are written to your project's `conf/sync.json` file.

14.3.2.2. Creating Attributes in a Mapping

You can use a mapping to *create* attributes on the target resource. In the preceding example, the mapping creates a `phoneExtension` attribute with a default value of `0047` on the target object.

In other words, the `default` property specifies a value to assign to the attribute on the target object. Before OpenIDM determines the value of the target attribute, it first evaluates any applicable conditions, followed by any transformation scripts. If the `source` property and the `transform` script yield a null value, it then applies the default value, create and update actions. The default value overrides the target value, if one exists.

To set up attributes with default values in the Admin UI:

1. Select Configure > Mappings, and click on the Mapping you want to edit.

2. Click on the Target Property that you want to create (`phoneExtension` in the previous example), select the Default Values tab, and enter a default value for that property mapping.

14.3.2.3. Transforming Attributes in a Mapping

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

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

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

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

To set up a transformation script in the Admin UI:

1. Select Configure > Mappings, and select the Mapping.
2. Select the line with the target attribute whose value you want to set.
3. On the Transformation Script tab, select `Javascript` or `Groovy`, and enter the transformation as an `Inline Script` or specify the path to the file containing your transformation script.

14.3.2.4. Using Scriptable Conditions in a Mapping

By default, OpenIDM synchronizes all attributes in a mapping. To facilitate more complex relationships between source and target objects, you can define conditions for which OpenIDM maps certain attributes. OpenIDM supports two types of mapping conditions:

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

  Examples of condition filters include:

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

To set up mapping conditions in the Admin UI, select Configure > Mappings. Click the mapping for which you want to configure conditions. On the Properties tab, click on the attribute that you want to map, then select the Conditional Updates tab.

Configure the filtered condition on the `Condition Filter` tab, or a scriptable condition on the `Script` tab.

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

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

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

Tip

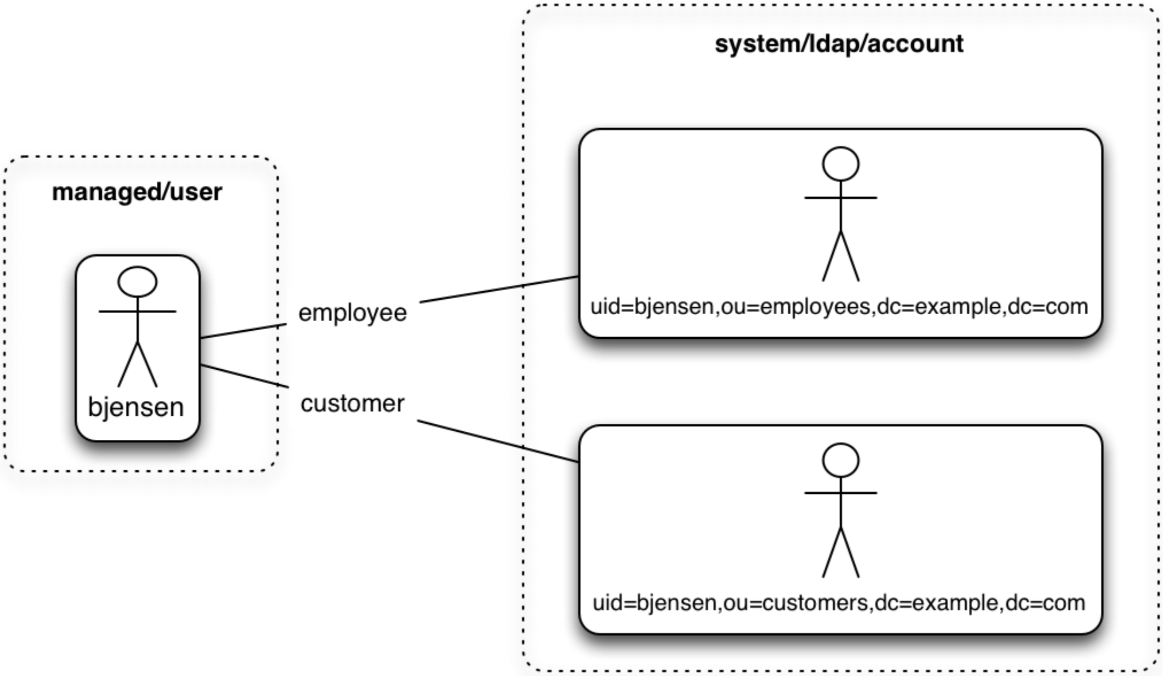
You can add comments to JSON files. While this example includes a property named `comment`, you can use any unique property name, as long as it is not used elsewhere in the server. OpenIDM ignores unknown property names in JSON configuration files.

14.3.2.5. Mapping a Single Source Object to Multiple Target Objects

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

OpenIDM uses *link qualifiers* to manage this one-to-many scenario. To map a single source object to multiple target objects, you indicate how the source object should be linked to the target object by defining link qualifiers. A link qualifier is essentially a label that identifies the *type* of link (or relationship) between each object.

In the previous example, you would define two link qualifiers that enable you to link both of bjensen's LDAP accounts to her managed user object, as shown in the following diagram:


system/ldap/account
managed/user
bjensen
employee
customer
uid=bjensen,ou=employees,dc=example,dc=com
uid=bjensen,ou=customers,dc=example,dc=com

Note from this diagram that the link qualifier is a property of the *link* between the source and target object, and not a property of the source or target object itself.

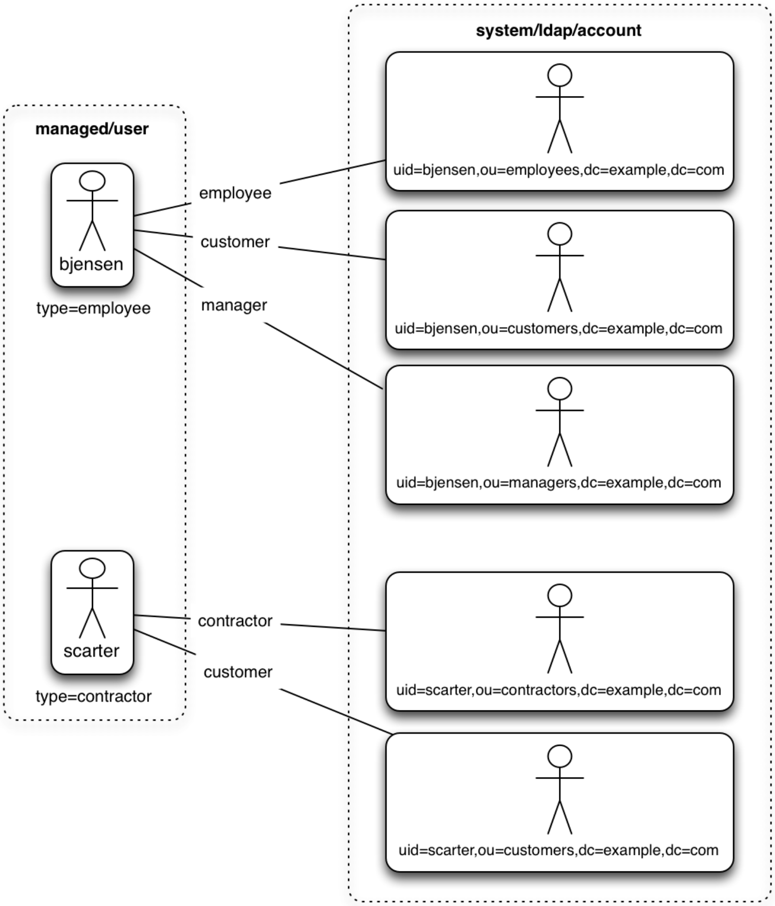
Link qualifiers are defined as part of the mapping (in your project's `conf/sync.json` file). Each link qualifier must be unique within the mapping. If no link qualifier is specified (when only one possible matching target object exists), OpenIDM uses a default link qualifier with the value `default`.

Link qualifiers can be defined as a static list, or dynamically, using a script. The following excerpt from a sample mapping shows the two static link qualifiers, `employee` and `customer`, described in the previous example:

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

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

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


system/ldap/account
managed/user
bjensen
type=employee
employee
customer
manager
uid=bjensen,ou=employees,dc=example,dc=com
uid=bjensen,ou=customers,dc=example,dc=com
uid=bjensen,ou=managers,dc=example,dc=com
scarter
type=contractor
contractor
customer
uid=scarter,ou=contractors,dc=example,dc=com
uid=scarter,ou=customers,dc=example,dc=com

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

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

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

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

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

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

Dynamic link qualifier scripts must return all valid link qualifiers when the `returnAll` global variable is true. The `returnAll` variable is used during the target reconciliation phase to check whether there are any target records that are unassigned, for each known link qualifier. For a list of the variables available to a dynamic link qualifier script, see "Script Triggers Defined in `sync.json`".

On their own, link qualifiers have no functionality. However, they can be referenced by various aspects of reconciliation to manage the situations where a single source object maps to multiple target objects. The following examples show how link qualifiers can be used in reconciliation operations:

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

  The following excerpt of a sample mapping defines a transformation script that generates the value of the `dn` attribute on an LDAP system. If the link qualifier is `employee`, the value of the target `dn` is set to `"uid=userName,ou=employees,dc=example,dc=com"`. If the link qualifier is `customer`, the value of the target `dn` is set to `"uid=userName,ou=customers,dc=example,dc=com"`. The reconciliation operation iterates through the link qualifiers for each source record. In this case, two LDAP objects, with different `dn`s would created for each managed user object.

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

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

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

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

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

For more information about correlation queries, see "Correlating Source Objects With Existing Target Objects".

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

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

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

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

Tip

To set up link qualifiers in the Admin UI select Configure > Mappings. Select a mapping, and click Properties > Link Qualifiers.

For an example that uses link qualifiers in conjunction with roles, see "*The Multi-Account Linking Sample*" in the *Samples Guide*.

14.3.2.6. Correlating Source Objects With Existing Target Objects

When OpenIDM creates an object on a target system in a synchronization process, it also creates a *link* between the source and target object. OpenIDM then uses that link to determine the object's *synchronization situation* during later synchronization operations. For a list of synchronization situations, see "How OpenIDM Assesses Synchronization Situations".

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

Correlation queries and correlation scripts are defined in your project's mapping (`conf/sync.json`) file. Each query or script is specific to the mapping for which it is configured. You can also configure correlation by using the Admin UI. Select Configure > Mappings, and click on the mapping for which you want to correlate. On the Association tab, expand Association Rules, and select Correlation Queries or Correlation Script from the list.

The following sections describe how to write correlation queries and scripts.

14.3.2.6.1. Writing Correlation Queries

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

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

Correlation queries can be expressed using a query filter (`_queryFilter`), a predefined query (`_queryId`), or a native query expression (`_queryExpression`). For more information on these query types, see "Defining and Calling Queries". The synchronization process executes the correlation query to search through the target system for objects that match the current source object.

The preferred syntax for a correlation query is a filtered query, using the `_queryFilter` keyword. Filtered queries should work in the same way on any backend, whereas other query types are generally specific to the backend. Predefined queries (using `_queryId`) and native queries (using `_queryExpression`) can also be used for correlation queries on managed resources. Note that `system` resources do not support native queries or predefined queries other than `query-all-ids` (which serves no purpose in a correlation query).

To configure a correlation query, define a script whose source returns a query that uses the `_queryFilter`, `_queryId`, or `_queryExpression` keyword. For example:

- For a `_queryId`, the value is the named query. Named parameters in the query map are expected by that query.

  ```
  {'_queryId' : 'for-userName', 'uid' : source.name}
  ```

- For a `_queryFilter`, the value is the abstract filter string:

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

- For a `_queryExpression`, the value is the system-specific query expression, such as raw SQL.

```
{'_queryExpression': 'select * from managed_user where givenName = \"' + source.firstname + '\"' }
```

Caution

Using a query expression in this way is not recommended as it exposes your system to SQL injection exploits.

14.3.2.6.1.1. Using Filtered Queries to Correlate Objects

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

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

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

  If the target object is a system object, this attribute must be referred to by its OpenIDM name rather than its OpenICF `nativeName`. For example, given the following provisioner configuration excerpt, the attribute to use in the correlation query would be `uid` and not `__NAME__`:

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

- The value to search for in the query.

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

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

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

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

The query can return zero or more objects. The situation that OpenIDM assigns to the source object depends on the number of target objects that are returned, and on the presence of any *link qualifiers* in the query. For information about synchronization situations, see "How OpenIDM Assesses Synchronization Situations". For information about link qualifiers, see "Mapping a Single Source Object to Multiple Target Objects".

14.3.2.6.1.2. Using Predefined Queries to Correlate Objects

For correlation queries on *managed objects*, you can use a query that has been predefined in the database table configuration file for the repository, either `conf/repo.jdbc.json` or `conf/repo.orientdb.json`. You reference the query ID in your project's `conf/sync.json` file.

The following example shows a query defined in the OrientDB repository configuration (`conf/repo.orientdb.json`) that can be used as the basis for a correlation query:

```
"for-userName" : "SELECT * FROM ${unquoted:_resource} WHERE userName = ${uid}
     SKIP ${unquoted:_pagedResultsOffset} LIMIT ${unquoted:_pageSize}"
```

By default, a `${value}` token replacement is assumed to be a quoted string. If the value is not a quoted string, use the `unquoted:` prefix, as shown above.

You would call this query in the mapping (`sync.json`) file as follows:

```
{
    "correlationQuery": {
      "type": "text/javascript",
      "source":
        "var qry = {'_queryId' : 'for-userName', 'uid' : source.name}; qry;"
    }
  }
```

In this correlation query, the `_queryId` property value (`for-userName`) matches the name of the query specified in `conf/repo.orientdb.json`. The `source.name` value replaces `${uid}` in the query. OpenIDM replaces `${unquoted:_resource}` in the query with the name of the table that holds managed objects.

14.3.2.6.1.3. Using the Expression Builder to Create Correlation Queries

OpenIDM provides a declarative correlation option, the expression builder, that makes it easier to configure correlation queries.

The easiest way to use the expression builder to create a correlation query is through the Admin UI:

1. Select Configure > Mappings and select the mapping for which you want to configure a correlation query.

2. On the Association tab, expand the Association Rules item and select Correlation Queries.

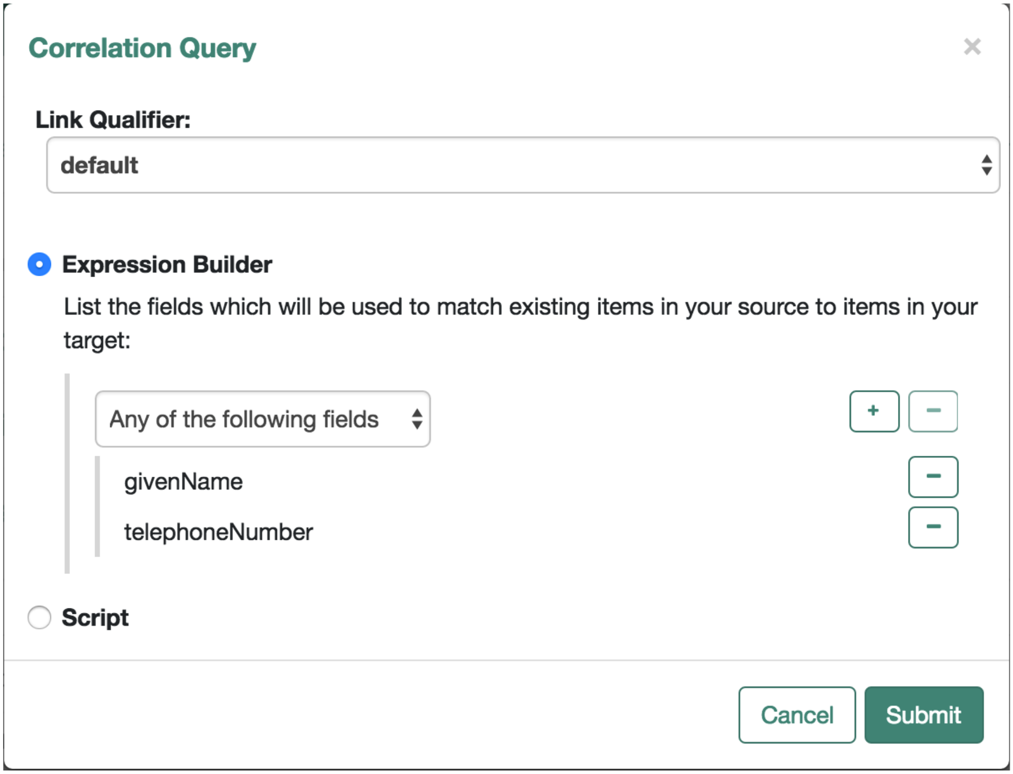
3. Click Add Correlation query.

4. In the Correlation Query window, select a link qualifier.

   If you do not need to correlate multiple potential target objects per source object, select the `default` link qualifier. For more information about linking to multiple target objects, see "Mapping a Single Source Object to Multiple Target Objects".

5. Select Expression Builder, and add or remove the fields whose values in the source and target must match.

   The following image shows how you can use the expression builder to build a correlation query for a mapping from `managed/user` to `system/ldap/accounts` objects. The query will create a match between the source (managed) object and the target (LDAP) object if the value of the `givenName` or the `telephoneNumber` of those objects is the same.



Correlation Query

Link Qualifier: default

● Expression Builder

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

Any of the following fields

givenName

telephoneNumber

○ Script

6. Click Submit to exit the Correlation Query pop-up then click Save.

The correlation query created in the previous steps displays as follows in the mapping configuration (`sync.json`):

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

14.3.2.6.2. Writing Correlation Scripts

If you need a more powerful correlation mechanism than a simple query can provide, you can write a correlation script with additional logic. Correlation scripts are generally more complex than correlation queries and impose no restrictions on the methods used to find matching objects. A correlation script must execute a query and return the result of that query.

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

In general, a correlation query should meet the requirements of most deployments. Correlation scripts can be useful, however, if your query needs extra processing, such as fuzzy-logic matching or out-of-band verification with a third-party service over REST.

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

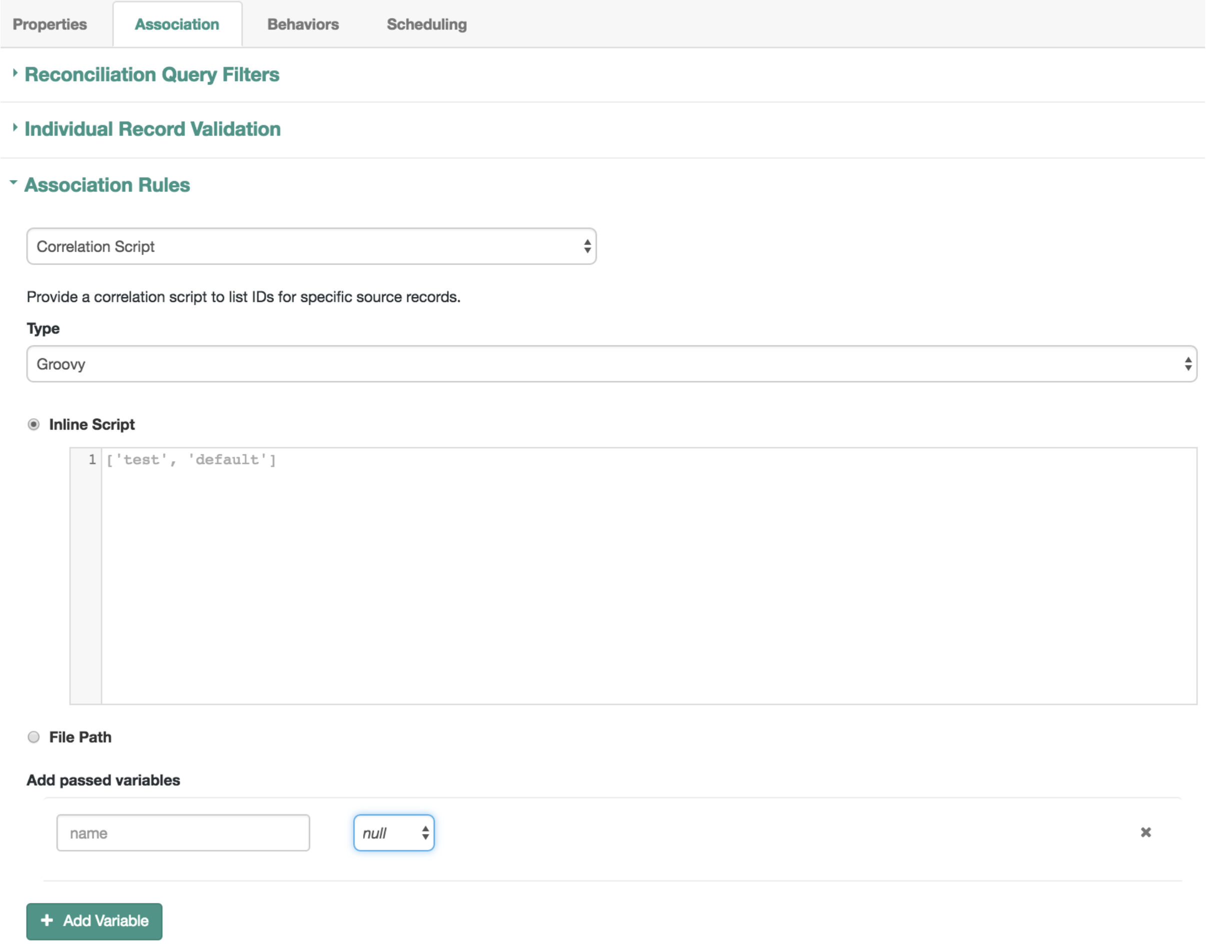
Correlation Script Using Link Qualifiers

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

To configure a correlation script in the Admin UI, follow these steps:

1. Select Configure > Mappings and select the mapping for which you want to configure the correlation script.

2. On the Association tab, expand the Association Rules item and select Correlation Script from the list.



Properties | Association | Behaviors | Scheduling

Reconciliation Query Filters

Individual Record Validation

Association Rules

Correlation Script

Provide a correlation script to list IDs for specific source records.

Type

Groovy

● Inline Script

```
['test', 'default']
```

○ File Path

Add passed variables

name | null

3. Select a script type (either JavaScript or Groovy) and either enter the script source in the Inline Script box, or specify the path to a file that contains the script.

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

4. Click Save to save the script as part of the mapping.

14.3.3. Filtering Synchronized Objects

By default, OpenIDM synchronizes all objects that match those defined in the connector configuration for the resource. Many connectors allow you to limit the scope of objects that the connector accesses. For example, the LDAP connector allows you to specify base DNs and LDAP filters so that you do not need to access every entry in the directory. You can also filter the source or target objects that are included in a synchronization operation. To apply these filters, use the `validSource`, `validTarget`, or `sourceCondition` properties in your mapping:

`validSource`

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

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

`validTarget`

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

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

`sourceCondition`

The `sourceCondition` element defines an additional filter that must be met for a source object's inclusion in a mapping.

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

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

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

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

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

In addition, the `sourceCondition` filter has the `linkQualifier` variable in its scope.

For more information about scripting, see "*Scripting Reference*".

14.3.4. Configuring Synchronization Filters With User Preferences

For all regular users (other than `openidm-admin`), you can set up preferences, such as those related to marketing and news updates. You can then use those preferences as a filter when reconciling users to a target repository.

OpenIDM includes default user preferences defined for the managed user object, available in the Admin UI and configured in the `managed.json` file.

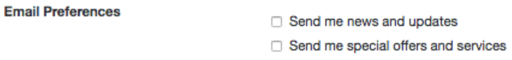
14.3.4.1. Configuring End User Preferences

In the default project, common marketing preference options are included for the managed user object. To find these preferences in the Admin UI, select Configure > Managed Objects and select the User managed object. Under the Preferences tab, you'll see keys and descriptions. You can also see these preferences in the `managed.json` file, illustrated here:

```
"preferences" : {
    "title" : "Preferences",
    "viewable" : true,
    "searchable" : false,
    "userEditable" : true,
    "type" : "object",
        "properties" : {
            "updates" : {
                "description" : "Send me news and updates",
                "type" : "boolean"
            },
            "marketing" : {
                "description" : "Send me special offers and services",
                "type" : "boolean"
            }
        },
    "order" : [
        "updates",
        "marketing"
    ],
    "required" : [ ]
},
```

14.3.4.2. Reviewing Preferences as an End User

When regular users log into the self-service UI, they'll see the preferences described in the following section: "Configuring End User Preferences". To review those preferences, log into the end user UI and select Profile > Preferences.



Email Preferences

□ Send me news and updates

□ Send me special offers and services

End users who accept these preferences get the following entries in their managed user data:

```
"preferences" : {
     "updates" : true,
     "marketing" : true
},
```

You can configure reconciliation to validate users who have chosen to accept the noted preferences.

14.3.4.3. User Preferences and Reconciliation

You can configure user preferences as a filter for reconciliation. For example, if some of your users do not want marketing emails, you can filter those users out of any reconciliation operation.

1. To configure user preferences as a filter, log into the Admin UI.

2. Select Configure > Mappings. Choose a mapping.

3. Under the Association tab, select Individual Record Validation.

4. Based on the options in the Valid Source drop down text box, you can select `Validate based on user preferences`. Users who have selected a preference such as `Send me special offers` will then be reconciled from the source to the target repository.

> **Note**
>
> What OpenIDM does during this reconciliation depends on the policy associated with the `UNQUALIFIED` situation for a `validSource`. The default action is to delete the target object (user). For more information, see "How OpenIDM Assesses Synchronization Situations".

Alternatively, you can edit the `sync.json` file directly. The following code block includes `preferences` as conditions to define a `validSource` on an individual record validation. OpenIDM applies these conditions at the next reconciliation.

```
"validSource" : {
    "type" : "text/javascript",
    "globals" : {
        "preferences" : [
            "updates",
            "marketing"
        ]
    },
    "file" : "ui/preferenceCheck.js"
},
"validTarget" : {
    "type" : "text/javascript",
    "globals" : { },
    "source" : ""
}
```

14.3.5. Preventing Accidental Deletion of a Target System

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

```
2015-06-05 10:41:18:918 WARN Cannot reconcile from an empty data
    source, unless allowEmptySourceSet is true.
```

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

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

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

```
{
    "mappings" : [
        {
        "name" : "systemXmlfileAccounts_managedUser",
        "source" : "system/xmlfile/account",
        "allowEmptySourceSet" : true,
        ...
```

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

14.3.5.1. Preventing Accidental Deletion in the Admin UI

To change the `allowEmptySourceSet` option in the Admin UI, choose Configure > Mappings. Select the desired mapping. In the Advanced tab, enable or disable the following option:

- Allow Reconciliations From an Empty Source

14.4. Constructing and Manipulating Attributes With Scripts

OpenIDM provides a number of *script hooks* to construct and manipulate attributes. These scripts can be triggered during various stages of the synchronization process, and are defined as part of the mapping, in the `sync.json` file.

The scripts can be triggered when a managed or system object is created (`onCreate`), updated (`onUpdate`), or deleted (`onDelete`). Scripts can also be triggered when a link is created (`onLink`) or removed (`onUnlink`).

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

Note

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

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

The following sample extract of a `sync.json` file derives a DN for an LDAP entry when the entry is created in the internal repository:

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

14.5. Advanced Use of Scripts in Mappings

"Constructing and Manipulating Attributes With Scripts" shows how to manipulate attributes with scripts when objects are created and updated. You might want to trigger scripts in response to other synchronization actions. For example, you might not want OpenIDM to delete a managed user directly when an external account record is deleted, but instead unlink the objects and deactivate the user in another resource. (Alternatively, you might delete the object in OpenIDM but nevertheless execute a script.) The following example shows a more advanced mapping configuration that exposes the script hooks available during synchronization.

```
{
    "mappings": [
        {
            "name": "systemLdapAccount_managedUser",
            "source": "system/ldap/account",
            "target": "managed/user",
            "validSource": {
                "type": "text/javascript",
                "file": "script/isValid.js"
            },
            "correlationQuery" : {
                "type" : "text/javascript",
                "source" : "var map = {'_queryFilter': 'uid eq \"' +
                     source.userName + '\"'}; map;"
            },
            "properties": [
                {
                    "source": "uid",
                    "transform": {
                        "type": "text/javascript",
                        "source": "source.toLowerCase()"
                    },
                    "target": "userName"
                },
                {
                    "source": "",
                    "transform": {
                        "type": "text/javascript",
                        "source": "if (source.myGivenName)
                            {source.myGivenName;} else {source.givenName;}"
                    },
                    "target": "givenName"
                },
                {
                    "source": "",
                    "transform": {
                        "type": "text/javascript",
                        "source": "if (source.mySn)
                            {source.mySn;} else {source.sn;}"
                    },
                    "target": "familyName"
                },
                {
                    "source": "cn",
                    "target": "fullname"
                },
                {
                    "comment": "Multi-valued in LDAP, single-valued in AD.
                        Retrieve first non-empty value.",
                    "source": "title",
                    "transform": {
                        "type": "text/javascript",
                        "file": "script/getFirstNonEmpty.js"
                    },
                    "target": "title"
                },
                {
                    "condition": {
                        "type": "text/javascript",
                        "source": "var clearObj = openidm.decrypt(object);
                            ((clearObj.password != null) &&
                            (clearObj.ldapPassword != clearObj.password))"
                    },
                    "transform": {
                        "type": "text/javascript",
                        "source": "source.password"
                    },
                    "target": "__PASSWORD__"
                }
            ],
            "onCreate": {
                "type": "text/javascript",
                "source": "target.ldapPassword = null;
                    target.adPassword = null;
                    target.password = null;
                    target.ldapStatus = 'New Account'"
            },
            "onUpdate": {
                "type": "text/javascript",
                "source": "target.ldapStatus = 'OLD'"
            },
            "onUnlink": {
                "type": "text/javascript",
                "file": "script/triggerAdDisable.js"
            },
            "policies": [
                {
                    "situation": "CONFIRMED",
                    "action": "UPDATE"
                },
                {
                    "situation": "FOUND",
                    "action": "UPDATE"
                },
                {
                    "situation": "ABSENT",
                    "action": "CREATE"
                },
                {
                    "situation": "AMBIGUOUS",
                    "action": "EXCEPTION"
                },
                {
                    "situation": "MISSING",
                    "action": "EXCEPTION"
                },
                {
                    "situation": "UNQUALIFIED",
                    "action": "UNLINK"
                },
                {
                    "situation": "UNASSIGNED",
                    "action": "EXCEPTION"
                }
            ]
        }
    ]
}
```

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

Triggered by Situation

onCreate, onUpdate, onDelete, onLink, onUnlink

Object Filter

validSource, validTarget

Correlating Objects

correlationQuery

Triggered on Reconciliation

result

Scripts Inside Properties

condition, transform

Your scripts can get data from any connected system at any time by using the `openidm.read(id)` function, where `id` is the identifier of the object to read.

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

```
repoUser = openidm.read("managed/user/ddoe");
```

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

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

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

14.6. Reusing Links Between Mappings

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

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

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

14.7. Managing Reconciliation

Reconciliation is the synchronization of objects between two data stores. You can trigger, cancel, and monitor reconciliation operations over REST, using the REST endpoint `http://localhost:8080/openidm/recon`. You can also perform most of these actions through the Admin UI.

14.7.1. Triggering a Reconciliation

The following example triggers a reconciliation operation over REST based on the `systemLdapAccounts_managedUser` mapping. The mapping is defined in the file `conf/sync.json`:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/recon?_action=recon&mapping=systemLdapAccounts_managedUser"
```

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

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

```
{"_id":"9f4260b6-553d-492d-aaa5-ae3c63bd90f0-14","state":"ACTIVE"}
```

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/recon?
_action=recon&mapping=systemLdapAccounts_managedUser&waitForCompletion=true"
```

14.7.1.1. Triggering a Reconciliation in the Admin UI

You can also trigger this reconciliation through the Admin UI. Select Configure > Mappings. In the mapping of your choice select Reconcile.

If you're reconciling a large number of items, the Admin UI shares the following message with you, possibly with numbers for entries reconciled and total entries.

```
In progress: reconciling source entries
```

Note

In the Admin UI, if you select `Cancel Reconciliation` before it is complete, you'll have to start the process again.

14.7.2. Canceling a Reconciliation

With a REST call, you can cancel a reconciliation in progress, by specifying the reconciliation run ID. The following REST call cancels the reconciliation run initiated in the previous section:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request POST \
"http://localhost:8080/openidm/recon/0890ad62-4738-4a3f-8b8e-f3c83bbf212e?_action=cancel"
```

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

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

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

14.7.2.1. Cancelling a Reconciliation in the Admin UI

In the Admin UI, you can cancel a reconciliation run in progress. When you select Configure > Mappings, and select Reconcile from a mapping, the `Cancel Reconciliation` button appears while the reconciliation is in progress.

14.7.3. Listing a History of Reconciliations

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

The following example displays all reconciliation runs:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
"http://localhost:8080/openidm/recon"
```

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

```
{
  "reconciliations": [
    {
      "ended": "2014-03-06T06:14:11.845Z",
      "_id": "4286510e-986a-4521-bfa4-8cd1e039a7f5",
      "mapping": "systemLdapAccounts_managedUser",
      "state": "SUCCESS",
      "stage": "COMPLETED_SUCCESS",
      "stageDescription": "reconciliation completed.",
      "progress": {
        "links": {
          "created": 1,
          "existing": {
          "total": "0",
          "processed": 0
        }
      },
      "target": {
        "created": 1,
        "existing": {
          "total": "2",
          "processed": 2
        }
      },
      "source": {
        "existing": {
          "total": "1",
          "processed": 1
        }
      }
    },
    "situationSummary": {
      "UNASSIGNED": 2,
      "TARGET_IGNORED": 0,
      "SOURCE_IGNORED": 0,
      "MISSING": 0,
      "FOUND": 0,
      "AMBIGUOUS": 0,
      "UNQUALIFIED": 0,
      "CONFIRMED": 0,
      "SOURCE_MISSING": 0,
      "ABSENT": 1
    },
    "started": "2014-03-06T06:14:04.722Z"
  },]
}
```

In contrast, the Admin UI displays the results of only the most recent reconciliation. For more information, see "Obtaining the Details of a Reconciliation in the Admin UI".

Each reconciliation run includes the following properties:

`_id`

The ID of the reconciliation run.

`mapping`

The name of the mapping, defined in the `conf/sync.json` file.

`state`

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

- `ACTIVE`

  The reconciliation run is in progress.

- `CANCELED`

  The reconciliation run was successfully canceled.

- `FAILED`

  The reconciliation run was terminated because of failure.

- `SUCCESS`

  The reconciliation run completed successfully.

`stage`

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

- `ACTIVE_INITIALIZED`

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

- `ACTIVE_QUERY_ENTRIES`

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

- `ACTIVE_RECONCILING_SOURCE`

  Reconciling the set of IDs retrieved from the mapping source.

- `ACTIVE_RECONCILING_TARGET`

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

- `ACTIVE_LINK_CLEANUP`

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

- `ACTIVE_PROCESSING_RESULTS`

  Post-processing of reconciliation results.

- `ACTIVE_CANCELING`

  Attempting to abort a reconciliation run in progress.

- `COMPLETED_SUCCESS`

  Successfully completed processing the reconciliation run.

- `COMPLETED_CANCELED`

  Completed processing because the reconciliation run was aborted.

- `COMPLETED_FAILED`

  Completed processing because of a failure.

`stageDescription`

A description of the stages described previously.

`progress`

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

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

14.7.4. Obtaining the Details of a Reconciliation

Display the details of a specific reconciliation over REST, by including the reconciliation run ID in the URL. The following call shows the details of the reconciliation run initiated in "Triggering a Reconciliation".

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/recon/0890ad62-4738-4a3f-8b8e-f3c83bbf212e"
{
  "ended": "2014-03-06T07:00:32.094Z",
  "_id": "7a07c100-4f11-4d7e-bf8e-fa4594f99d58",
  "mapping": "systemLdapAccounts_managedUser",
  "state": "SUCCESS",
  "stage": "COMPLETED_SUCCESS",
  "stageDescription": "reconciliation completed.",
  "progress": {
     "links": {
       "created": 0,
       "existing": {
         "total": "1",
         "processed": 1
       }
     },
     "target": {
       "created": 0,
       "existing": {
         "total": "3",
         "processed": 3
       }
     },
     "source": {
       "existing": {
         "total": "1",
         "processed": 1
       }
     }
  },
  "situationSummary": {
     "UNASSIGNED": 2,
     "TARGET_IGNORED": 0,
     "SOURCE_IGNORED": 0,
     "MISSING": 0,
     "FOUND": 0,
     "AMBIGUOUS": 0,
     "UNQUALIFIED": 0,
     "CONFIRMED": 1,
     "SOURCE_MISSING": 0,
     "ABSENT": 0
  },
  "started": "2014-03-06T07:00:31.907Z"
}
```

14.7.4.1. Obtaining the Details of a Reconciliation in the Admin UI

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

```
Completed: Last reconciled July 29, 2016 14:13
```

When you select this option, the details of the reconciliation appear.

14.7.5. Triggering LiveSync Over REST

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

There are two ways to trigger liveSync over REST:

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

  ```
  $ curl \
   --header "X-OpenIDM-Username: openidm-admin" \
   --header "X-OpenIDM-Password: openidm-admin" \
   --request POST \
   "http://localhost:8080/openidm/system/ldap/account?_action=liveSync"
  ```

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

  The following example calls the same liveSync operation as the previous example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system?_action=liveSync&source=system/ldap/account"
```

A successful liveSync operation returns the following response:

```
{
    "_rev": "4",
    "_id": "SYSTEMLDAPACCOUNT",
    "connectorData": {
        "nativeType": "integer",
        "syncToken": 1
    }
}
```

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

To troubleshoot a liveSync operation that has not succeeded, include an optional parameter (`detailedFailure`) to return additional information. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system/ldap/account?_action=liveSync&detailedFailure=true"
```

Note

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

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

14.8. Restricting Reconciliation By Using Queries

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

You can restrict reconciliation to specific entries by defining explicit source or target queries in the mapping configuration.

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

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

The format of the query can be any query type that is supported by the resource, and can include additional parameters, if applicable. OpenIDM supports the following query types.

For queries on managed objects:

- `_queryId` for arbitrary predefined, parameterized queries
- `_queryFilter` for arbitrary filters, in common filter notation
- `_queryExpression` for client-supplied queries, in native query format

For queries on system objects:

- `_queryId=query-all-ids` (the only supported predefined query)
- `_queryFilter` for arbitrary filters, in common filter notation

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

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

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{"sourceQuery": {"_queryId" : "query-all-ids"}}' \
 "http://localhost:8080/openidm/recon?_action=recon&mapping=managedUser_systemLdapAccounts"
```

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

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

14.8.1. Restricting Reconciliation in the Admin UI, With Queries

You can also restrict reconciliation by using queries through the Admin UI. Select Configure > Mappings, select a Mapping > Association > Reconciliation Query Filters. You can then specify desired source and target queries.

14.9. Restricting Reconciliation to a Specific ID

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

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

The following example is based on the data from Sample 2b, which maps an LDAP server with the OpenIDM repository. The example reconciles only the user bjensen, using the managedUser_systemLdapAccounts mapping to update the user account in LDAP with the data from the OpenIDM repository. The _id for bjensen in this example is b3c2f414-e7b3-46aa-8ce6-f4ab1e89288c. The example assumes that implicit synchronization has been disabled and that a reconciliation operation is required to copy changes made in the repository to the LDAP system:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/recon?
_action=reconById&mapping=managedUser_systemLdapAccounts&ids=b3c2f414-e7b3-46aa-8ce6-f4ab1e89288c"
```

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

14.10. Configuring the LiveSync Retry Policy

You can specify the results when a liveSync operation reports a failure. Configure the liveSync retry policy to specify the number of times a failed modification should be reattempted and what should happen if the modification is unsuccessful after the specified number of attempts. If no retry policy is configured, OpenIDM reattempts the change an infinite number of times until the change is successful. This behavior can increase data consistency in the case of transient failures (for example, when the connection to the database is temporarily lost). However, in situations where the cause of the failure is permanent (for example, if the change does not meet certain policy requirements) the change will never succeed, regardless of the number of attempts. In this case, the infinite retry behavior can effectively block subsequent liveSync operations from starting.

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

The liveSync retry policy is configured in the connector configuration file (`provisioner.openicf-*.json`). The sample connector configuration files have a retry policy defined as follows:

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

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

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

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

- `logged-ignore` indicates that OpenIDM should ignore the failed modification, and log its occurrence.
- `dead-letter-queue` indicates that OpenIDM should save the details of the failed modification in a table in the repository (accessible over REST at `repo/synchronisation/deadLetterQueue/`*`provisioner-name`*).
- `script` specifies a custom script that should be executed when the maximum number of retries has been reached. For information about using custom scripts in the configuration, see "*Scripting Reference*".

In addition to the regular objects described in "*Scripting Reference*", the following objects are available in the script scope:

`syncFailure`

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

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

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

`failureCause`

Provides the exception that caused the original liveSync failure.

`failureHandlers`

OpenIDM currently provides two synchronization failure handlers out of the box:

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

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

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

Two sample scripts are provided in `path/to/openidm/samples/syncfailure/script`, one that logs failures, and one that sends them to the dead letter queue in the repository.

The following sample provisioner configuration file extract shows a liveSync retry policy that specifies a maximum of four retries before the failed modification is sent to the dead letter queue:

```
...
"connectorName" : "org.identityconnectors.ldap.LdapConnector"
    },

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

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

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

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

```
INFO: sync retries = 4/4, retries exhausted
Jul 19, 2013 11:59:30 AM
    org.forgerock.openidm.provisioner.openicf.syncfailure.DeadLetterQueueHandler invoke
INFO: uid=jdoe,ou=people,dc=example,dc=com saved to dead letter queue
```

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

You can view the failed modification in the dead letter queue, over the REST interface, as follows:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/repo/synchronisation/deadLetterQueue/ldap?_queryId=query-all-ids"
{
    "query-time-ms": 2,
    "result":
        [
            {
                "_id": "4",
                "_rev": "0"
            }
        ],
    "conversion-time-ms": 0
}
```

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/repo/synchronisation/deadLetterQueue/ldap/4"
{
  "objectType": "account",
  "systemIdentifier": "ldap",
  "failureCause": "org.forgerock.openidm.sync.SynchronizationException:
            org.forgerock.openidm.objset.ConflictException:
            org.forgerock.openidm.sync.SynchronizationException:
            org.forgerock.openidm.script.ScriptException:
            ReferenceError: \"bad\" is not defined.
            (PropertyMapping/mappings/0/properties/3/condition#1)",
  "token": 4,
  "failedRecord": "complete record, in xml format"
  "uid": "uid=jdoe,ou=people,dc=example,dc=com",
  "_rev": "0",
  "_id": "4"
}
```

14.11. Disabling Automatic Synchronization Operations

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

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

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

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

14.12. Configuring Synchronization Failure Compensation

When implicit synchronization is used to push a large number of changes from the managed object repository to several external repositories, the process can take some time. Problems such as lost connections might happen, resulting in the changes being only partially synchronized.

For example, if a Human Resources manager adds a group of new employees in one database, a partial synchronization might mean that some of those employees do not have access to their email or other systems.

You can configure implicit synchronization to revert a reconciliation operation if it is not completely successful. This is known as *failure compensation*. An example of such a configuration is shown in "Sample 5b - Failure Compensation With Multiple Resources" in the *Samples Guide*. That sample demonstrates how OpenIDM compensates when synchronization to an external resource fails.

Failure compensation works by using the optional `onSync` hook, which can be specified in the `conf/managed.json` file. The `onSync` hook can be used to provide failure compensation as follows:

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

The `onSync` hook references a script (`compensate.js`), that is located in the `/path/to/openidm/bin/defaults/script` directory.

When a managed object is changed, an implicit synchronization operation attempts to synchronize the change (and any other pending changes) with any external data store(s) for which a mapping is configured. Note that implicit synchronization is enabled by default. To disable implicit synchronization, see "Disabling Automatic Synchronization Operations".

The implicit synchronization process proceeds with each mapping, in the order in which the mappings are specified in `sync.json`.

The `compensate.js` script is designed to avoid partial synchronization. If synchronization is successful for all configured mappings, OpenIDM exits from the script.

If an implicit synchronization operation fails for a particular resource, the `onSync` hook invokes the `compensate.js` script. This script attempts to revert the original change by performing another update to the managed object. This change, in turn, triggers another implicit synchronization operation to all external resources for which mappings are configured.

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

If failure compensation is not configured, any issues with connections to an external resource can result in out of sync data stores, as discussed in the earlier Human Resources example.

With the `compensate.js` script, any such errors will result in each data store using the information it had before implicit synchronization started. OpenIDM stores that information, temporarily, in the `oldObject` variable.

In the previous Human Resources example, managers should see that new employees are not shown in their database. Then, the OpenIDM administrators can check log files for errors, address them, and restart implicit synchronization with a new REST call.

14.13. Synchronization Situations and Actions

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

The action that is taken for each situation is defined in the `policies` section of your synchronization mapping. The following excerpt of the `sync.json` file from Sample 2b shows the defined actions in that sample:

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

You can also define these actions in the Admin UI. Select Configure > Mappings, click on the required Mapping, then select the Behaviors tab to specify different actions per situation.

If you do not define an action for a particular situation, OpenIDM takes the *default action* for that situation. The following section describes how situations are assessed, lists all possible situations and describes the default actions taken for each situation.

14.13.1. How OpenIDM Assesses Synchronization Situations

Reconciliation is performed in two phases:

1. *Source reconciliation*, where OpenIDM accounts for source objects and associated links based on the configured mapping.

2. *Target reconciliation*, where OpenIDM iterates over the target objects that were not processed in the first phase.

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

During source reconciliation OpenIDM iterates through the objects in the source resource and evaluates the following conditions:

1. Is the source object valid?

   Valid source objects are categorized `qualifies=1`. Invalid source objects are categorized `qualifies=0`. Invalid objects include objects that were filtered out by a `validSource` script or `sourceCondition`. For more information, see "Filtering Synchronized Objects".

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

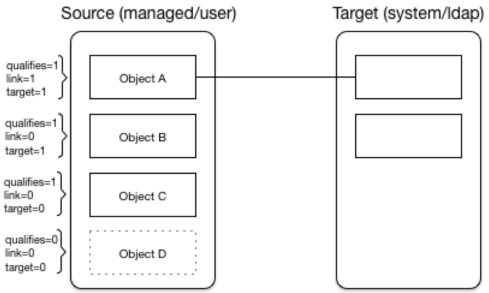
   Source objects that have a corresponding link in the repository's `links` table are categorized `link=1`. Source objects that do not have a corresponding link are categorized `link=0`.

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

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

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

Object Categorization During the Source Synchronization Phase



Based on the categorizations of source objects during the source reconciliation phase, OpenIDM assesses a *situation* for each source object. Not all situations are detected in all synchronization types. The following list describes the set of synchronization situations, when they can be detected, the default action taken for that situation, and valid alternative actions that can be defined for the situation:

Situations detected during reconciliation and source change events

`CONFIRMED` (qualifies=1, link=1, target=1)

The source object qualifies for a target object, and is linked to an existing target object.

Default action: `UPDATE` the target object.

Other valid actions: `IGNORE, REPORT, NOREPORT, ASYNC`

`FOUND` (qualifies=1, link=0, target=1)

The source object qualifies for a target object and is not linked to an existing target object. There is a single target object that correlates with this source object, according to the logic in the correlation.

Default action: `UPDATE` the target object.

Other valid actions: `EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC`

`FOUND_ALREADY_LINKED` (qualifies=1, link=1, target=1)

The source object qualifies for a target object and is not linked to an existing target object. There is a single target object that correlates with this source object, according to the logic in the correlation, but that target object is already linked to a different source object.

Default action: throw an `EXCEPTION`.

Other valid actions: `IGNORE, REPORT, NOREPORT, ASYNC`

`ABSENT` (qualifies=1, link=0, target=0)

The source object qualifies for a target object, is not linked to an existing target object, and no correlated target object is found.

Default action: `CREATE` a target object.

Other valid actions: `EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC`

`UNQUALIFIED` (qualifies=0, link=0 or 1, target=1 or >1)

The source object is unqualified (by the `validSource` script). One or more target objects are found through the correlation logic.

Default action: `DELETE` the target object or objects.

Other valid actions: `EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC`

`AMBIGUOUS` (qualifies=1, link=0, target>1)

The source object qualifies for a target object, is not linked to an existing target object, but there is more than one correlated target object (that is, more than one possible match on the target system).

Default action: throw an `EXCEPTION`.

Other valid actions: `IGNORE, REPORT, NOREPORT, ASYNC`

`MISSING` (qualifies=1, link=1, target=0)

The source object qualifies for a target object, and is linked to a target object, but the target object is missing.

Default action: throw an `EXCEPTION`.

Other valid actions: `CREATE, UNLINK, DELETE, IGNORE, REPORT, NOREPORT, ASYNC`

> **Note**
>
> When a target object is deleted, the link from the target to the corresponding source object is not deleted automatically. This allows IDM to detect and report items that might have been removed without permission or might need review. If you need to remove the corresponding link when a target object is deleted, change the action to UNLINK to remove the link, or to DELETE to remove the target object and the link.

`SOURCE_IGNORED` (qualifies=0, link=0, target=0)

The source object is unqualified (by the `validSource` script), no link is found, and no correlated target exists.

Default action: `IGNORE` the source object.

Other valid actions: `EXCEPTION, REPORT, NOREPORT, ASYNC`

Situations detected only during source change events:

`TARGET_IGNORED` (qualifies=0, link=0 or 1, target=1)

The source object is unqualified (by the `validSource` script). One or more target objects are found through the correlation logic.

This situation differs from the `UNQUALIFIED` situation, based on the status of the link and the target. If there is a link, the target is not valid. If there is no link and exactly one target, that target is not valid.

Default action: `IGNORE` the target object until the next full reconciliation operation.

Other valid actions: `DELETE, UNLINK, EXCEPTION, REPORT, NOREPORT, ASYNC`

`LINK_ONLY` (qualifies=n/a, link=1, target=0)

The source may or may not be qualified. A link is found, but no target object is found.

Default action: throw an `EXCEPTION`.

Other valid actions: `UNLINK, IGNORE, REPORT, NOREPORT, ASYNC`

`ALL_GONE` (qualifies=n/a, link=0, cannot-correlate)

The source object has been removed. No link is found. Correlation is not possible, for one of the following reasons:

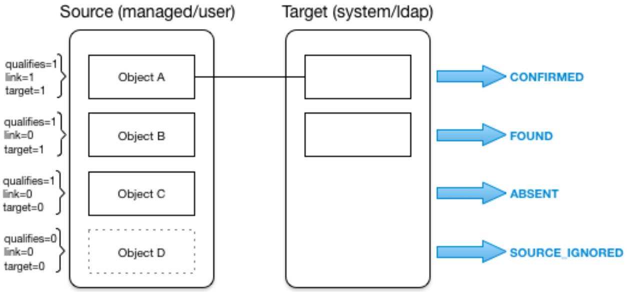
- No previous source object can be found.
- There is no correlation logic.
- A previous source object was found, and correlation logic exists, but no corresponding target was found.

Default action: `IGNORE` the source object.

Other valid actions: `EXCEPTION, REPORT, NOREPORT, ASYNC`

Based on this list, OpenIDM would assign the following situations to the previous diagram:

Situation Assignment During the Source Synchronization Phase



During target reconciliation, OpenIDM iterates through the objects in the target resource that were not accounted for during source reconciliation, and evaluates the following conditions:

1. Is the target object valid?

   Valid target objects are categorized `qualifies=1`. Invalid target objects are categorized `qualifies=0`. Invalid objects include objects that were filtered out by a `validTarget` script. For more information, see "Filtering Synchronized Objects".

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

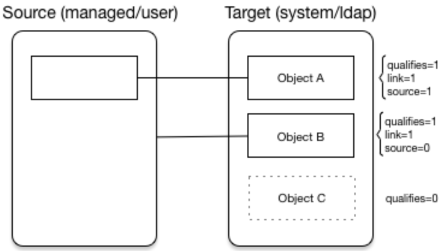
   Target objects that have a corresponding link in the repository's `links` table are categorized `link=1`. Target objects that do not have a corresponding link are categorized `link=0`.

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

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

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

Object Categorization During the Target Synchronization Phase



Based on the categorizations of target objects during the target reconciliation phase, OpenIDM assesses a *situation* for each remaining target object. Not all situations are detected in all synchronization types. The following list describes the set of synchronization situations, when they can be detected, the default action taken for that situation, and valid alternative actions that can be defined for the situation:

Situations detected only during reconciliation:

`TARGET_IGNORED` **(qualifies=0)**

During target reconciliation, the target becomes unqualified by the `validTarget` script.

Default action: `IGNORE` the target object.

Other valid actions: `DELETE, UNLINK, REPORT, NOREPORT, ASYNC`

`UNASSIGNED` **(qualifies=1, link=0)**

A valid target object exists but does not have a link.

Default action: throw an `EXCEPTION`.

Other valid actions: `IGNORE, REPORT, NOREPORT, ASYNC`

`CONFIRMED` **(qualifies=1, link=1, source=1)**

The target object qualifies, and a link to a source object exists.

Default action: `UPDATE` the target object.

Other valid actions: `IGNORE, REPORT, NOREPORT`

Situations detected during reconciliation and target change events:

`UNQUALIFIED` **(qualifies=0, link=1, source=1, but source does not qualify)**

The target object is unqualified (by the `validTarget` script). There is a link to an existing source object, which is also unqualified.

Default action: `DELETE` the target object.

Other valid actions: `UNLINK, EXCEPTION, IGNORE, REPORT, NOREPORT, ASYNC`

`SOURCE_MISSING` **(qualifies=1, link=1, source=0)**

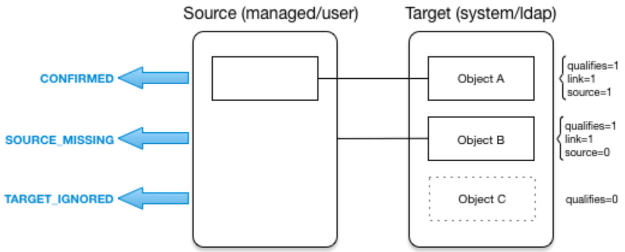
The target object qualifies and a link is found, but the source object is missing.

Default action: throw an `EXCEPTION`.

Other valid actions: `DELETE, UNLINK, IGNORE, REPORT, NOREPORT, ASYNC`

Based on this list, OpenIDM would assign the following situations to the previous diagram:

Situation Assignment During the Target Synchronization Phase



The following sections walk you through how OpenIDM assigns situations during source and target reconciliation.

14.13.2. Source Reconciliation

OpenIDM starts reconciliation and liveSync by reading a list of objects from the resource. For reconciliation, the list includes all objects that are available through the connector. For liveSync, the list contains only changed objects. OpenIDM can filter objects from the list by using the script specified in the `validSource` property, or the query specified in the `sourceCondition` property.

OpenIDM then iterates the list, checking each entry against the `validSource` and `sourceCondition` filters, and classifying objects according to their situations as described in "How OpenIDM Assesses Synchronization Situations". OpenIDM uses the list of links for the current mapping to classify objects. Finally, OpenIDM executes the action that is configured for each situation.

The following table shows how OpenIDM assigns the appropriate situation during source reconciliation, depending on whether a valid source exists (Source Qualifies), whether a link exists in the repository (Link Exists), and the number of target objects found, based either on links or on the results of the correlation.

Resolving Source Reconciliation Situations

Source Qualifies?		Link Exists?		Target Objects Found[a]			Situation
Yes	No	Yes	No	0	1	> 1	
	X		X		X		SOURCE_MISSING
	X		X			X	UNQUALIFIED
	X	X		X			UNQUALIFIED
	X	X			X		TARGET_IGNORED
	X	X				X	UNQUALIFIED
X			X	X			ABSENT
X			X		X		FOUND

Source Qualifies?		Link Exists?		Target Objects Found[a]			Situation
Yes	No	Yes	No	0	1	> 1	
X			X[b]		X		FOUND_ALREADY_LINKED
X			X			X	AMBIGUOUS
X		X		X			MISSING
X		X			X		CONFIRMED

[a] If no link exists for the source object, then OpenIDM executes correlation logic. If no previous object is available, OpenIDM cannot correlate.

[b] A link exists from the target object but it is not for this specific source object.

14.13.3. Target Reconciliation

During source reconciliation, OpenIDM cannot detect situations where no source object exists, such as the `UNASSIGNED` situation. When no source object exists, OpenIDM detects the situation during the second reconciliation phase, target reconciliation. During target reconciliation, OpenIDM iterates all target objects that do not have a representation on the source, checking each object against the `validTarget` filter, determining the appropriate situation and executing the action configured for the situation.

The following table shows how OpenIDM assigns the appropriate situation during target reconciliation, depending on whether a valid target exists (Target Qualifies), whether a link with an appropriate type exists in the repository (Link Exists), whether a source object exists (Source Exists), and whether the source object qualifies (Source Qualifies). Not all situations assigned during source reconciliation are assigned during target reconciliation.

Resolving Target Reconciliation Situations

Target Qualifies?		Link Exists?		Source Exists?		Source Qualifies?		Situation
Yes	No	Yes	No	Yes	No	Yes	No	
	X							TARGET_IGNORED
X			X		X			UNASSIGNED
X		X		X		X		CONFIRMED
X		X		X			X	UNQUALIFIED
X		X			X			SOURCE_MISSING

14.13.4. Situations Specific to Implicit Synchronization and LiveSync

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

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

Resolving Implicit Sync and LiveSync Delete Situations

Source Qualifies?		**Link Exists?**		**Target Objects Found** [a]			**Situation**
Yes	**No**	**Yes**	**No**	**0**	**1**	**> 1**	
N/A	N/A	X		X			LINK_ONLY
N/A	N/A		X	X			ALL_GONE
X			X			X	AMBIGUOUS
	X		X			X	UNQUALIFIED

[a] If no link exists for the source object, OpenIDM executes any included correlation logic. If a link exists, correlation does not apply.

14.13.5. Synchronization Actions

When a situation has been assigned to an object, OpenIDM takes the actions configured in the mapping. If no action is configured, OpenIDM takes the default action for the situation. OpenIDM supports the following actions:

`CREATE`

Create and link a target object.

`UPDATE`

Link and update a target object.

`DELETE`

Delete and unlink the target object.

`LINK`

Link the correlated target object.

`UNLINK`

Unlink the linked target object.

`EXCEPTION`

Flag the link situation as an exception.

Do not use this action for liveSync mappings.

`IGNORE`

Do not change the link or target object state.

`REPORT`

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

`NOREPORT`

Do not perform any action or generate any report.

`ASYNC`

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

14.13.6. Launching a Script As an Action

In addition to the static synchronization actions described in the previous section, you can provide a script that is run in specific synchronization situations. The script can be either JavaScript or Groovy, and can be provided inline (with the `"source"` property), or referenced from a file, (with the `"file"` property).

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

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

The variables available to a script that is called as an action are `source`, `target`, `linkQualifier`, and `recon` (where `recon.actionParam` contains information about the current reconciliation operation). For more information about the variables available to scripts, see "Variables Available to Scripts".

The result obtained from evaluating this script must be a string whose value is one of the synchronization actions listed in "Synchronization Actions". This resulting action will be shown in the reconciliation log.

To launch a script as a synchronization action in the Admin UI:

1. Select Configure > Mappings.
2. Select the mapping that you want to change.
3. On the Behaviors tab, click the pencil icon next to the situation whose action you want to change.
4. On the Perform this Action tab, click Script, then enter the script that corresponds to the action.

14.13.7. Launching a Workflow As an Action

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

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

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

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

To launch a workflow as a synchronization action in the Admin UI:

1. Select Configure > Mappings.
2. Select the mapping that you want to change.
3. On the Behaviors tab, click the pencil icon next to the situation whose action you want to change.
4. On the Perform this Action tab, click Workflow, then enter the details of the workflow you want to launch.

14.14. Asynchronous Reconciliation

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

For example, you might want a reconciliation process to assess new user accounts that need to be created on a target resource. However, new user account creation might require some kind of approval from a manager before the accounts are actually created. The initial reconciliation process can assess the accounts that need to be created, launch a workflow to request management approval for those accounts, and then relaunch the reconciliation process to create the accounts, after the management approval has been received.

In this scenario, the defined script returns IGNORE for new accounts and the reconciliation engine does not continue processing the given object. The script then initiates an asynchronous process which calls back and completes the reconciliation process at a later stage.

A sample configuration for this scenario is available in `openidm/samples/sample9`, and described in "Demonstrating Asynchronous Reconciliation Using a Workflow" in the *Samples Guide*.

Configuring asynchronous reconciliation using a workflow involves the following steps:

1. Create the workflow definition file (`.xml or .bar` file) and place it in the `openidm/workflow` directory. For more information about creating workflows, see "*Integrating Business Processes and Workflows*".

2. Modify the `conf/sync.json` file for the situation or situations that should call the workflow. Reference the workflow name in the configuration for that situation.

   For example, the following `sync.json` extract calls the `managedUserApproval` workflow if the situation is assessed as `ABSENT`:

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

3. In the sample configuration, the workflow calls a second, explicit reconciliation process as a final step. This reconciliation process is called on the `sync` context path, with the `performAction` action (`openidm.action('sync', 'performAction', params)`).

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

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

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

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/sync?_action=performAction&sourceId=uid%3Dbjensen%2Cou%3DPeople%2Cdc
%3Dexample%2Cdc%3Dcom&mapping=
 systemLdapAccounts_ManagedUser&action=DELETE"
{
    "status": "OK"
}
```

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

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/sync?_action=performAction&action=LINK
   &sourceId=4b39f74d-92c1-4346-9322-d86cb2d828a8&targetId=scarter.user
   &mapping=managedUser_systemXmlfileAccounts&linkQualifier=user"
{
    "status": "OK"
}
```

For more information about linking to multiple accounts, see "Mapping a Single Source Object to Multiple Target Objects".

14.15. Configuring Case Sensitivity For Data Stores

OpenIDM is case-sensitive, which means that an upper case ID is considered different from an otherwise identical lower case ID during reconciliation. In contrast, OpenDJ is case-insensitive. This can lead to problems, as the ID of links created by reconciliation may not match the case of the IDs expected by OpenIDM.

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

Alternatively, you can address case-sensitivity issues from a datastore in one of the following ways:

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

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

  You may also need to modify the OpenICF provisioner to make it case-insensitive. To do so, open your provisioner configuration file, and set the `enableFilteredResultsHandler` property to `false`:

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

Caution

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

- Use a case-insensitive option from your datastore. For example, in MySQL, you can change the collation of `managedobjectproperties.propvalue` to `utf8_general_ci`.

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

14.16. Optimizing Reconciliation Performance

By default, reconciliation is configured to function optimally, with regard to performance. Some of these optimizations might, however, be unsuitable for your environment. The following sections describe the default optimizations and how they can be configured, as well as additional methods you can use to improve the performance of reconciliation operations.

14.16.1. Correlating Empty Target Sets

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

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

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

14.16.1.1. Correlating Empty Target Sets in the Admin UI

To change the `correlateEmptyTargetSet` option in the Admin UI, choose Configure > Mappings. Select the desired mapping. In the Advanced tab, enable or disable the following option:

- Correlate Empty Target Objects

14.16.2. Prefetching Links

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

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

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

14.16.2.1. Prefetching Links in the Admin UI

To change the `prefetchLinks` option in the Admin UI, choose Configure > Mappings. Select the desired mapping. In the Advanced tab, enable or disable the following option:

- Pre-fetch Links

14.16.3. Parallel Reconciliation Threads

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

- The hardware has many cores and supports more concurrent threads. As a rule of thumb for performance tuning, start with setting the thread number to two times the number of cores.

- The source or target is an external system with high latency or slow response times. Threads may then spend considerable time waiting for a response from the external system. Increasing the available threads enables the system to prepare or continue with additional objects.

To change the number of threads, set the `taskThreads` property in the `conf/sync.json` file, for example:

```
    "mappings" : [
        {
            "name" : "systemXmlfileAccounts_managedUser",
            "source" : "system/xmlfile/account",
            "target" : "managed/user",
            "taskThreads" : 20
            ...
         }
    ]
}
```

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

14.16.3.1. Parallel Reconciliation Threads in the Admin UI

To change the `taskThreads` option in the Admin UI, choose Configure > Mappings. Select the desired mapping. In the Advanced tab, adjust the number of threads in the following text box:

- Threads Per Reconciliation

14.16.4. Improving Reconciliation Query Performance

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

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

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

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

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

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

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

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

Setting these properties to `true` indicates that the complete object is returned. This setting is typically required only for very small objects, for which the number of returned fields does not reach the threshold required for the auto-detection mechanism to assume that it is a full object. In this case, the query result includes all the details required to pre-load the full object.

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

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

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

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

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

14.16.5. Improving Role-Based Provisioning Performance With an `onRecon` Script

OpenIDM provides an `onRecon` script that runs once, at the beginning of each reconciliation. This script can perform any setup or initialization operations that are appropriate for the reconciliation run.

In addition, OpenIDM provides a `reconContext` that is added to a request's context chain when reconciliation runs. The `reconContext` can store pre-loaded data that can be used by other OpenIDM components (such as the managed object service) to increase performance.

The default `onRecon` script (`openidm/bin/default/script/roles/onRecon.groovy`) loads the `reconContext` with all the roles and assignments that are required for the current mapping. The `effectiveAssignments` script checks the `reconContext` first. If a `reconContext` is present, the script uses that `reconContext` to populate the array of `effectiveAssignments`. This prevents a read operation to `managed/role` or `managed/assignment` every time reconciliation runs, and greatly improves the overall performance for role-based provisioning.

You can customize the `onRecon`, `effectiveRoles`, and `effectiveAssignments` scripts to provide additional business logic during reconciliation. If you customize these scripts, copy the default scripts from `openidm/bin/defaults/scripts` into your project's `script` directory, and make the changes there.

14.16.6. Paging Reconciliation Query Results

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

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

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

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

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

14.17. Scheduling Synchronization

You can schedule synchronization operations, such as liveSync and reconciliation, using Quartz **cronTrigger** syntax. For more information about **cronTrigger**, see the corresponding Quartz documentation.

This section describes scheduling specifically for reconciliation and liveSync. You can use OpenIDM's scheduler service to schedule any other event by supplying a link to a script file, in which that event is defined. For information about scheduling other events, see "*Scheduling Tasks and Events*".

14.17.1. Configuring Scheduled Synchronization

Each scheduled reconciliation and liveSync task requires a schedule configuration file in your project's `conf` directory. By convention, schedule configuration files are named `schedule-*schedule-name*.json`, where *schedule-name* is a logical name for the scheduled synchronization operation, such as `reconcile_systemXmlAccounts_managedUser`.

Schedule configuration files have the following format:

```
{
 "enabled"       : true,
 "persisted"     : true,
 "type"          : "cron",
 "startTime"     : "(optional) time",
 "endTime"       : "(optional) time",
 "schedule"      : "cron expression",
 "misfirePolicy" : "optional, string",
 "timeZone"      : "(optional) time zone",
 "invokeService" : "service identifier",
 "invokeContext" : "service specific context info"
}
```

These properties are specific to the scheduler service, and are explained in "*Scheduling Tasks and Events*".

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

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

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

The `mapping` is either referenced by its name in the `conf/sync.json` file, or defined inline by using the `mapping` property, as shown in the example in "Specifying the Mapping as Part of the Schedule".

For liveSync, the properties are set as follows:

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

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

Important

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

Daylight Savings Time (DST) can cause problems for scheduled liveSync operations. For more information, see "Schedules and Daylight Savings Time".

14.17.2. Specifying the Mapping as Part of the Schedule

Mappings for synchronization operations are usually stored in your project's `sync.json` file. You can, however, provide the mapping for scheduled synchronization operation by including it as part of the `invokeContext` of the schedule configuration, as shown in the following example:

```
{
    "enabled": true,
    "type": "cron",
    "schedule": "0 08 16 * * ?",
    "persisted": true,
    "invokeService": "sync",
    "invokeContext": {
        "action": "reconcile",
        "mapping": {
            "name": "CSV_XML",
            "source": "system/Ldap/account",
            "target": "managed/user",
            "properties": [
                {
                    "source": "firstname",
                    "target": "firstname"
                },
                ...
            ],
            "policies": [...]
        }
    }
}
```

Chapter 15
Extending Functionality By Using Scripts

Scripting enables you to customize various aspects of OpenIDM functionality, for example, by providing custom logic between source and target mappings, defining correlation rules, filters, and triggers, and so on.

OpenIDM supports scripts written in JavaScript and Groovy. Script options, and the locations in which OpenIDM expects to find scripts, are configured in the `conf/script.json` file for your project. For more information, see "Setting the Script Configuration".

OpenIDM includes several default scripts in the following directory *`install-dir`*`/bin/defaults/script/`. Do not modify or remove any of the scripts in this directory. OpenIDM needs these scripts to run specific services. Scripts in this folder are not guaranteed to remain constant between product releases.

If you develop custom scripts, copy them to the `script/` directory for your project, for example, `path/to/openidm/samples/sample2/script/`.

15.1. Validating Scripts Over REST

OpenIDM exposes a `script` endpoint over which scripts can be validated, by specifying the script parameters as part of the JSON payload. This functionality enables you to 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 enables you to call one from the other on the OpenIDM 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 "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 "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 "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
        (3864142CB836831FAB8EAB662F566139CDC22BF2#1)
        in 3864142CB836831FAB8EAB662F566139CDC22BF2
        at line number 1 at column number 39"
}
```

15.2. Creating Custom Endpoints to Launch Scripts

Custom endpoints enable you to run arbitrary scripts through the OpenIDM 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/customendpoint` 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.

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

Endpoint configuration files and scripts are discussed further in the following sections.

15.2.1. Creating a 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/customendpoint/conf/endpoint-echo.json`, the route to the endpoint is `endpoint/echo`.

Note that this `context` path is not the same as the *context chain* of the request. For information about the request context chain, see "Understanding the 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);"
```

You must set authorization appropriately for any custom endpoints that you add, for example, by restricting the appropriate methods to the appropriate roles. For more information, see "Authorization".

15.2.2. Writing Custom Endpoint Scripts

The custom endpoint script files in the `samples/customendpoint/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 OpenIDM 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,
            queryExpression: request.queryExpression,
            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 "Understanding the Request Context Chain".

Paging parameters

The `pagedResultsCookie`, `pagedResultsOffset` and `pageSize` parameters are specific to `query` methods. For more information see "Paging and Counting Query Results".

Query parameters

The `queryExpression`, `queryId` and `queryFilter` parameters are specific to `query` methods. For more information see "Constructing Queries".

15.2.3. Setting Up Exceptions in Scripts

When you create a custom endpoint script, you might need to build 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"
    ]
]))
```

15.3. Registering Custom Scripted Actions

OpenIDM enables you to 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.

Custom scripted actions have access to the following variables: `context`, `request`, `resourcePath`, and `object`. For more information, see "Variables Available to Scripts".

Custom scripted actions facilitate arbitrary behavior on managed objects. For example, imagine a scenario where you want your managed users to be able to indicate whether they receive update notifications. You can define an *action* that toggles the value of a specific property on the user object. You can implement this scenario by following these steps:

- Add an `updates` property to the managed user schema (in your project's `conf/managed.json` file) as follows:

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

- Add an action named `toggleUpdates` to the managed user object definition as follows:

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

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

- Call the script by specifying the ID of the action in a POST request on the user object, for example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/managed/user/ID?_actionId=toggleUpdate"
```

You can test this functionality as follows:

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{
  "_id":"bjensen",
  "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": "bjensen",
  "_rev": "1",
  "userName": "bjensen",
  "sn": "Jensen",
  "givenName": "Barbara",
  "mail": "bjensen@example.com",
  "telephoneNumber": "5556787",
  "description": "Created by OpenIDM REST.",
  "updates": true,
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

2. Run the `toggleUpdates` action on bjensen's entry:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/managed/user/bjensen?_action=toggleUpdates"
{
  "_id": "bjensen",
  "_rev": "2",
  "userName": "bjensen",
  "sn": "Jensen",
  "givenName": "Barbara",
  "mail": "bjensen@example.com",
  "telephoneNumber": "5556787",
  "description": "Created by OpenIDM REST.",
  "updates": false,
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

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

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.

Chapter 16
Scheduling Tasks and Events

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

The scheduler service supports **cronTrigger** syntax based on the Quartz Scheduler (bundled with OpenIDM). For more information about **cronTrigger**, see the corresponding Quartz documentation.

By default, OpenIDM picks up changes to scheduled tasks and events dynamically, during initialization and also at runtime. For more information, see "Changing the Default Configuration".

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

16.1. Configuring 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" : "&{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 and is `true` by default.

- `advancedProperties` (optional) enables you to configure additional properties for the Quartz Scheduler.

Note

In clustered environments, the scheduler service obtains an `instanceID`, and checkin and timeout settings from the cluster management service (defined in the *`project-dir`*`/conf/cluster.json` file).

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

Note

You can also control whether schedules are persisted in your project's `conf/boot/boot.properties` file. In the default `boot.properties` file, persistent schedules are enabled:

```
# enables the execution of persistent schedulers
openidm.scheduler.execute.persistent.schedules=true
```

Settings in `boot.properties` are not persisted in the repository. Therefore, you can use the `boot.properties` file to set different values for a property across different nodes in a cluster. For example, if your deployment has a four-node cluster and you want only two of those nodes to execute persisted schedules, you can disable persisted schedules in the `boot.properties` files of the remaining two nodes. If you set these values directly in the `scheduler.json` file, the values are persisted to the repository and are therefore applied to all nodes in the cluster.

Changing the value of the `openidm.scheduler.execute.persistent.schedules` property in the `boot.properties` file changes the scheduler that manages scheduled tasks on that node. Because the persistent and in-memory schedulers are managed separately, a situation can arise where two separate schedules have the same schedule name.

For more information about persistent schedules, see "Configuring Persistent Schedules".

16.2. Configuring Schedules

You can use the Admin UI or JSON configuration files to schedule tasks and events. To configure a schedule in the Admin UI, select Configure > Schedules and then click Add Schedule. If configure your schedules directly in JSON files, place these files in your project's `conf/` directory. By convention, OpenIDM 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_systemXmlAccounts_managedUser.json`. There are several example schedule configuration files in the `openidm/samples/schedules` directory.

Each schedule configuration file has the following format:

```
{
 "enabled"              : true,
 "persisted"            : true,
 "concurrentExecution" : false,
 "type"                 : "cron",
 "startTime"            : "(optional) time",
 "endTime"              : "(optional) time",
 "schedule"             : "cron expression",
 "misfirePolicy"        : "optional, string",
 "timeZone"             : "(optional) time zone",
 "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`, OpenIDM 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 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 "Configuring Persistent Schedules".

`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`

Currently OpenIDM supports only `cron`.

`startTime` **(optional)**

Used to start the schedule at some time in the future. If this 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`).

`endTime` **(optional)**

Used to plan the end of scheduling.

`schedule`

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.

`timeZone` **(optional)**

If not set, OpenIDM uses the system time zone.

`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 "Scanning 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": "systemLdapAccounts_managedUser"
    }
}
```

For a scheduled reconciliation task, you can define the mapping in one of two ways:

- Reference a mapping by its name in `sync.json`, as shown in the previous example. The mapping must exist in your project's `conf/sync.json` file.
- Add the mapping definition inline by using the `mapping` property, as shown in "Specifying the Mapping as Part of the Schedule".

The following example invokes a liveSync operation:

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

For scheduled liveSync tasks, the `source` property follows OpenIDM'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 string `It is working: 26` to the console. A similar sample schedule is provided in `schedule-script.json` in the `/path/to/openidm/samples/schedules` directory.

```
{
    "invokeService": "script",
    "invokeContext": {
        "script" : {
            "type" : "text/javascript",
            "source" : "java.lang.System.out.println('It is working: ' + input.edit);",
            "input": { "edit": 26}
        }
    }
}
```

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`

16.3. Schedules and Daylight Savings Time

The schedule service uses Quartz **cronTrigger** syntax. CronTrigger schedules 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, set the time zone of the schedule to Coordinated Universal Time (UTC). UTC is never subject to DST, so schedules will continue to fire as normal.

16.4. Configuring 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 OpenIDM instances). When scheduling is persistent, a particular schedule will be launched only once across the cluster, rather than once on every OpenIDM instance. For example, if your deployment includes a cluster of OpenIDM 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 OpenIDM 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 OpenIDM 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.

16.5. 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, starting on the hour:

```
{
    "enabled": false,
    "persisted": true,
    "type": "cron",
    "schedule": "0 0/30 * * * ?",
    "invokeService": "sync",
    "invokeContext": {
        "action": "reconcile",
        "mapping": "systemLdapAccounts_managedUser"
    }
}
```

The following example shows a schedule for liveSync enabled to run every 15 seconds, starting at the beginning of the minute. Note that the schedule is persisted, that is, stored in the internal 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": "cron",
    "schedule": "0/15 * * * * ?",
    "invokeService": "provisioner",
    "invokeContext": {
        "action": "liveSync",
        "source": "system/ldap/account"
    }
}
```

16.6. Managing 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 "Placing a Server in Maintenance Mode" in the *Installation 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.

16.6.1. Validating Schedule Syntax

Schedules are defined using Quartz cron syntax. You can validate your cron expression by sending the expression as a JSON object to the scheduler context path. For example:

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

16.6.2. Defining 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 while 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 schedule configuration is as described in "Configuring the Scheduler Service":

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --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,
  "misfirePolicy": "fireAndProceed",
  "schedule": "0/1 * * * * ?",
  "type": "cron",
  "invokeService": "org.forgerock.openidm.script",
  "invokeContext": {
    "script": {
      "type": "text/javascript",
      "file": "script/testlog.js"
    }
  },
  "invokeLogLevel": "info",
  "timeZone": null,
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [
    {
      "previous_state": 0,
      "name": "trigger-testlog-schedule",
      "state": 4,
      "nodeId": "node1",
      "acquired": true,
      "serialized": "rO0ABXNyABZvcmcucXVhcnR6L...30HhzcQB+ABx3CAAAAVdwIrfQeA==",
      "group": "scheduler-service-group",
      "_id": "scheduler-service-group_$x$x$_trigger-testlog-schedule",
      "_rev": "4"
    }
  ],
  "nextRunDate": "2016-09-28T09:31:47.000Z"
}
```

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 "Querying 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" \
 --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": "9858a39d-b1e7-4842-9874-0fb8179b149a",
  "enabled": true,
  "persisted": true,
  "misfirePolicy": "fireAndProceed",
  "schedule": "0/1 * * * * ?",
  "type": "cron",
  "invokeService": "org.forgerock.openidm.script",
  "invokeContext": {
    "script": {
      "type": "text/javascript",
      "file": "script/testlog.js"
    }
  },
  "invokeLogLevel": "info",
  "timeZone": null,
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [
    {
      "previous_state": 0,
      "name": "trigger-9858a39d-b1e7-4842-9874-0fb8179b149a",
      "state": 4,
      "nodeId": "node1",
      "acquired": true,
      "serialized": "...+XAeHNxAH4AHHcIAAABV2wX4dh4c3EAfgAcdwgAAAFXbBfh2Hg=...",
      "group": "scheduler-service-group",
      "_id": "scheduler-service-group_$x$x$_trigger-9858a39d-b1e7-4842-9874-0fb8179b149a",
      "_rev": "4"
    }
  ],
  "nextRunDate": "2016-09-27T14:41:28.000Z"
}
```

The output includes the generated `_id` of the schedule, in this case `"_id": "9858a39d-b1e7-4842-9874-0fb8179b149a"`.

16.6.3. Obtaining the Details of a Scheduled Job

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/scheduler/job/testlog-schedule"
{
  "_id": "testlog-schedule",
  "enabled": true,
  "persisted": true,
  "misfirePolicy": "fireAndProceed",
  "schedule": "0/1 * * * * ?",
  "type": "cron",
  "invokeService": "org.forgerock.openidm.script",
  "invokeContext": {
    "script": {
      "type": "text/javascript",
      "file": "script/testlog.js"
    }
  },
  "invokeLogLevel": "info",
  "timeZone": null,
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [
    {
      "previous_state": -1,
      "name": "trigger-testlog-schedule",
      "state": 0,
      "nodeId": "node1",
      "acquired": true,
      "serialized": "rO0ABXNyABZvcmcucXVhcnR6L.../AHhzcQB+ABx3CAAAAVdwIrfQeA==",
      "group": "scheduler-service-group",
      "_id": "scheduler-service-group_$x$x$_trigger-testlog-schedule",
      "_rev": "7550"
    }
  ],
  "nextRunDate": "2016-09-28T10:03:13.000Z"
}
```

16.6.4. Querying Scheduled Jobs

You can query defined and running scheduled jobs using a regular query filter or a parameterized query. Support for parameterized queries is restricted to `_queryId=query-all-ids`. For more information about query filters, see "Constructing Queries".

The following query returns the IDs of all defined schedules:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/scheduler/job?_queryId=query-all-ids"
{
  "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" \
 --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": "2016-09-28T10:11:06.000Z"
    }
  ],
  ...
}
```

16.6.5. Updating a Schedule

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

The following example disables the testlog schedule created in the previous section by setting `"enabled":false`:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --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,
  "misfirePolicy": "fireAndProceed",
  "schedule": "0/1 * * * * ?",
  "type": "cron",
  "invokeService": "org.forgerock.openidm.script",
  "invokeContext": {
    "script": {
      "type": "text/javascript",
      "file": "script/testlog.js"
    }
  },
  "invokeLogLevel": "info",
  "timeZone": null,
  "startTime": null,
  "endTime": null,
  "concurrentExecution": false,
  "triggers": [],
  "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.

16.6.6. Deleting 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" \
 --request DELETE \
 "http://localhost:8080/openidm/scheduler/job/testlog-schedule"
{
  "_id": "testlog-schedule",
  "enabled": true
,
...
}
```

The DELETE request returns the entire JSON object.

16.6.7. Obtaining a List of Running Scheduled Jobs

The following command returns a list of the jobs that are currently executing. This list enables you to 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" \
 --request POST \
 "http://localhost:8080/openidm/scheduler/job?_action=listCurrentlyExecutingJobs"
[
  {
    "enabled": true,
    "persisted": true,
    "misfirePolicy": "fireAndProceed",
    "schedule": "0 0/1 * * * ?",
    "type": "cron",
    "invokeService": "org.forgerock.openidm.sync",
    "invokeContext": {
      "action": "reconcile",
      "mapping": "systemLdapAccounts_managedUser"
    },
    "invokeLogLevel": "info",
    "timeZone": null,
    "startTime": null,
    "endTime": null,
    "concurrentExecution": false
  }
]
```

16.6.8. Pausing 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.

The following 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" \
 --request POST \
 "http://localhost:8080/openidm/scheduler/job?_action=pauseJobs"
{
    "success": true
}
```

16.6.9. Resuming 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.

The following 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" \
 --request POST \
 "http://localhost:8080/openidm/scheduler/job?_action=resumeJobs"
{
    "success": true
}
```

16.6.10. Querying 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" \
 --request GET \
 "http://localhost:8080/openidm/scheduler/trigger?_queryFilter=true"
{
  "result": [
    {
      "_id": "scheduler-service-group_$x$x$_trigger-testlog-schedule",
      "_rev": "20862",
      "previous_state": -1,
      "name": "trigger-testlog-schedule",
      "state": 0,
      "nodeId": "node1",
      "acquired": true,
      "serialized": "rO0ABXNyABZvcmcucXVhcnR6L.../iHhzcQB+ABx3CAAAAVdwXUAweA==",
      "group": "scheduler-service-group"
    },
    {
      "_id": "scheduler-service-group_$x$x$_trigger-reconcile_systemLdapAccounts_managedUser",
      "_rev": "1553",
      "previous_state": -1,
      "name": "trigger-reconcile_systemLdapAccounts_managedUser",
      "state": 0,
      "nodeId": null,
      "acquired": false,
      "serialized": "rO0ABXNyABZvcmcucXVhcnR6L...0gCB4c3EAfgAcdwgAAAFXcF6QIHg=",
      "group": "scheduler-service-group"
    }
  ],
  ...
}
```

The contents of a trigger object are as follows:

`_id`

The ID of the trigger. The trigger ID takes the form `group_$x$x$_trigger-schedule-id`

`_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.

`previous_state`

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, in the form `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.

`acquired`

Whether the trigger has already been acquired by a node. Boolean, true or false.

`serialized`

The Base64 serialization of the trigger class.

`group`

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

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

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/scheduler/trigger/scheduler-service-group_\$x\$x\$_trigger-testlog-
schedule"
{
  "_id": "scheduler-service-group_$x$x$_trigger-testlog-schedule",
  "_rev": "32088",
  "previous_state": -1,
  "name": "trigger-testlog-schedule",
  "state": 0,
  "nodeId": "node1",
  "acquired": true,
  "serialized": "rO0ABXNyABZvcmcucXVhcnR6L...2oHhzcQB+ABx3CAAAAVdwXUAweA==",
  "group": "scheduler-service-group"
}
```

Note that you need to escape the `$` signs in the URL.

To view the triggers that have been acquired, per node, send a GET request to the `scheduler/acquiredTriggers` context path. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/scheduler/acquiredTriggers"
{
  "_id": "acquiredTriggers",
  "_rev": "102",
  "node1": [
    "scheduler-service-group_$x$x$_trigger-testlog-schedule"
  ]
}
```

To view the triggers that have not yet been acquired by any node, send a GET request to the `scheduler/waitingTriggers` context path. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/scheduler/waitingTriggers"
{
  "_id": "waitingTriggers",
  "_rev": "1576",
  "names": [
    "scheduler-service-group_$x$x$_trigger-0da27688-7ece-4799-bca4-09e185a6b0f4",
    "scheduler-service-group_$x$x$_trigger-0eeaf861-604b-4cf4-a044-bbbc78377070",
    "scheduler-service-group_$x$x$_trigger-136b7a1a-3aee-4321-8b6a-3e860e7b0292",
    "scheduler-service-group_$x$x$_trigger-1f6b116b-aa06-41da-9c19-80314373a20f",
    "scheduler-service-group_$x$x$_trigger-659b2bb0-53b8-4a4e-8347-8ed1ed5286af",
    "scheduler-service-group_$x$x$_trigger-testlog-schedule",
    "scheduler-service-group_$x$x$_trigger-ad9db1c7-a06d-4dc9-83b9-0c2e405dde1f"
  ]
}
```

16.7. Managing Schedules Through the Admin UI

To manage schedules through the Admin UI, select Configure > Schedules. By default, only persisted schedules are shown in the Schedules list. To show non-persisted (in memory) schedules, select Filter by Type > In Memory.

16.8. Scanning Data to Trigger Tasks

In addition to the fine-grained scheduling facility described previously, OpenIDM provides a task scanning mechanism. The task scanner enables you to perform a batch scan on a specified property in OpenIDM, at a scheduled interval, and then to launch a task when the value of that property matches a specified value.

When the task scanner identifies a condition that should trigger the task, it can invoke a script created specifically to handle the task.

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.

16.8.1. Configuring 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" : "cron",
    "schedule" : "0 0 * * * ?",
    "persisted" : true,
    "concurrentExecution" : false,
    "invokeService" : "taskscanner",
    "invokeContext" : {
        "waitForCompletion" : false,
        "numberOfThreads" : 5,
        "scan" : {
            "_queryId" : "scan-tasks",
            "object" : "managed/user",
            "property" : "/sunset/date",

            "condition" : {
                "before": "${Time.now}"
            },
            "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. Below is an example command to add 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"
 \
--request POST
 \
--data '[{
 "operation" : "add",
 "field" : "sunset/date",
 "value" : "2017-06-20T22:58:36.272Z"
}]' \
"http://localhost:8080/openidm/managed/user?_action=patch&_queryId=for-userName&uid=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`

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

`_queryFilter, _queryId, or _queryExpression`

A query filter, predefined query, or query expression that identifies the entries for which this task should be run. Query filters are recommended but you can also use native query expressions and parameterized, or predefined queries to identify the entries to be scanned.

The query filter provided in the sample schedule configuration (`((/sunset/date lt \"${Time.now}\") AND !(${taskState.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.

`object`

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

`property`

Defines the property of the managed object, against which the query is performed. In the previous example, the `"property" : "/sunset/date"` indicates a JSON pointer that maps to the object attribute, and can be understood as `sunset: {"date" : "date"}`.

If you are using a JDBC repository, with a generic mapping, you must explicitly set this property as searchable so that it can be queried by the task scanner. For more information, see "Using Generic Mappings".

`condition` (optional)

Indicates the conditions that must be matched for the defined property.

In the previous example, the scanner scans for users whose `/sunset/date` is prior to the current timestamp (at the time the script is run).

You can use these fields to define any condition. For example, if you wanted to limit the scanned objects to a specified location, say, London, you could formulate a query to compare against object locations and then set the condition to be:

```
"condition" : {
    "location" : "London"
},
```

For time-based conditions, the `condition` property supports macro syntax, based on the `Time.now` object (which fetches the current time). You can specify any date/time in relation to the current time, using the `+` or `-` operator, and a duration modifier. For example: `${Time.now + 1d}` would return all user objects whose `/sunset/date` is the following day (current time plus one day). 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

`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 OpenIDM 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 OpenIDM 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/taskscanner/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 in OpenIDM, see "*Scripting Reference*".

16.8.2. Managing Scanning Tasks Over REST

You can trigger, cancel, and monitor scanning tasks over the REST interface, using the REST endpoint `http://localhost:8080/openidm/taskscanner`.

16.8.2.1. Triggering a Scanning Task

The following REST command runs a task named "taskscan_sunset". The task itself is defined in a file named `conf/schedule-taskscan_sunset.json`:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/taskscanner?_action=execute&name=schedule/taskscan_sunset"
```

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.

For example, the scanning task initiated previously would return something similar to the following, as soon as it was initiated:

```
{"_id":"edfaf59c-aad1-442a-adf6-3620b24f8385"}
```

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`). You can also set the property directly over the REST interface when the task is initiated. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/taskscanner?_action=execute&name=schedule/
taskscan_sunset&waitForCompletion=true"
```

16.8.2.2. Canceling a Scanning Task

You can cancel a scanning task by sending a REST call with the `cancel` action, specifying the task ID. For example, the following call cancels the scanning task initiated in the previous section:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/taskscanner/edfaf59c-aad1-442a-adf6-3620b24f8385?_action=cancel"
{
    "_id":"edfaf59c-aad1-442a-adf6-3620b24f8385",
    "action":"cancel",
    "status":"SUCCESS"
}
```

16.8.2.3. Listing Scanning Tasks

You can display a list of scanning tasks that have completed, and those that are in progress, by running a RESTful GET on the `openidm/taskscanner` context path. The following example displays all scanning tasks:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/taskscanner"
{
 "tasks": [
    {
      "ended": 1352455546182
      "started": 1352455546149,
      "progress": {
        "failures": 0
        "successes": 2400,
        "total": 2400,
        "processed": 2400,
        "state": "COMPLETED",
      },
      "_id": "edfaf59c-aad1-442a-adf6-3620b24f8385",
    }
  ]
}
```

Each scanning task has the following properties:

`ended`

The time at which the scanning task ended.

`started`

The time at which the scanning task started.

`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 overall state of the task, `INITIALIZED`, `ACTIVE`, `COMPLETED`, `CANCELLED`, or `ERROR`

`_id`

The ID of the scanning task.

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.

Chapter 17

Managing Passwords

OpenIDM provides password management features that help you enforce password policies, limit the number of passwords users must remember, and let users reset and change their passwords.

17.1. Enforcing Password Policy

A password policy is a set of rules defining what sequence of characters constitutes an acceptable password. Acceptable passwords generally are too complex for users or automated programs to generate or guess.

Password policies set requirements for password length, character sets that passwords must contain, dictionary words and other values that passwords must not contain. Password policies also require that users not reuse old passwords, and that users change their passwords on a regular basis.

OpenIDM enforces password policy rules as part of the general policy service. For more information about the policy service, see "*Using Policies to Validate Data*". The default password policy applies the following rules to passwords as they are created and updated:

- A password property is required for any user object.
- The value of a password cannot be empty.
- The password must include at least one capital letter.
- The password must include at least one number.
- The minimum length of a password is 8 characters.
- The password cannot contain the user name, given name, or family name.

You can remove these validation requirements, or include additional requirements, by configuring the policy for passwords. For more information, see "Configuring the Default Policy for Managed Objects".

The password validation mechanism can apply in many situations.

Password change and password reset

> Password change involves changing a user or account password in accordance with password policy. Password reset involves setting a new user or account password on behalf of a user.

By default, OpenIDM controls password values as they are provisioned.

To change the default administrative user password, `openidm-admin`, see "Replace Default Security Settings".

Password recovery

Password recovery involves recovering a password or setting a new password when the password has been forgotten.

OpenIDM provides a self-service end user interface for password changes, password recovery, and password reset. For more information, see "*Configuring User Self-Service*".

Password comparisons with dictionary words

You can add dictionary lookups to prevent use of password values that match dictionary words.

Password history

You can add checks to prevent reuse of previous password values. For more information, see "Creating a Password History Policy".

Password expiration

You can configure OpenIDM to call a workflow that ensures users are able to change expiring or to reset expired passwords.

17.1.1. Creating a Password History Policy

To create a password history policy, you must include customized scripts as described in "Storing Multiple Passwords For Managed Users" in the *Samples Guide*. Copy these scripts to your*`project-dir/script`* directory.

You must also modify the following configuration files:

- Modify the `sync.json` file to reference the custom `onCreate-onUpdate-sync.js` script:

  ```
  "onCreate" : {
      "type" : "text/javascript",
      "file" : "script/onCreate-onUpdate-sync.js"
  },
  "onUpdate" : {
      "type" : "text/javascript",
      "file" : "script/onCreate-onUpdate-sync.js"
  }
  ```

  If you have existing `onCreate` and `onUpdate` code blocks, you may need to consolidate options either in the applicable script file, or in a `source` entry.

- Modify the `router.json` file to include code blocks for the `managed/user` object and associated policy. These policies apply to the arbitrary `ldapPassword` parameter which you will also add to the `managed.json` file:

```
{
  "pattern" : "managed/user.*",
  "onRequest" : {
    "type" : "text/javascript",
    "file" : "script/set-additional-passwords.js",
    "additionalPasswordFields" : [
      "ldapPassword"
    ]
  },
  "methods" : [
    "create",
    "update"
  ]
},
{
  "pattern" : "policy/managed/user.*",
  "onRequest" : {
    "type" : "text/javascript",
    "file" : "script/set-additional-passwords.js",
    "additionalPasswordFields" : [
      "ldapPassword"
    ]
  },
  "methods" : [
    "action"
  ]
}
```

- In the `policy.json` file, include the `pwpolicy.js` file from your project's `script/` subdirectory, as `additionalFiles`:

```
"type" : "text/javascript",
"file" : "policy.js",
"additionalFiles": [ "script/pwpolicy.js" ]
```

- Now make the following changes to your project's `managed.json` file.

  - Find the `"name" : "user",` object code block, normally near the start of the file. Include the following code blocks for the `onValidate`, `onCreate`, and `onUpdate` scripts. The value for the `storedFields` and `historyFields` should match the `additionalPasswordFields` that you included in the `router.json` file.

    You may vary the value of `historySize`, depending on the number of recent passwords you want to record in the history for each user. A `historySize` of 2 means that users who change their passwords can't use their previous two passwords.

```
"name" : "user",
"onValidate" : {
    "type" : "groovy",
    "file" : "script/storeFields.groovy",
    "storedFields" : [
        "ldapPassword"
    ]
},
"onCreate" : {
    "type" : "text/javascript",
    "file" : "script/onCreate-user-custom.js",
    "historyFields" : [
        "ldapPassword"
    ],
    "historySize" : 2
},
"onUpdate" : {
    "type" : "text/javascript",
    "file" : "script/onUpdate-user-custom.js",
    "historyFields" : [
        "ldapPassword"
    ],
    "historySize" : 2
}
```

- In the `properties` of the user object, add the following code block for `ldapPassword`

```
"ldapPassword" : {
    "title" : "Password",
    "type" : "string",
    "viewable" : false,
    "searchable" : false,
    "minLength" : 8,
    "userEditable" : true,
    "secureHash" : {
        "algorithm" : "SHA-256"
    },
    "policies" : [
        {
            "policyId" : "at-least-X-capitals",
            "params" : {
                "numCaps" : 2
            }
        },
        {
            "policyId" : "at-least-X-numbers",
            "params" : {
                "numNums" : 1
            }
        },
        {
            "policyId" : "cannot-contain-others",
            "params" : {
                "disallowedFields" : [
                    "userName",
                    "givenName",
                    "sn"
                ]
            }
        },
        {
            "policyId" : "is-new",
            "params" : {
                "historyLength" : 2
            }
        }
    ]
}
```

- Add the following `fieldHistory` code block, which maps field names to a list of historical values for the field.

```
"fieldHistory" : {
    "title" : "Field History",
    "type" : "object",
    "viewable" : false,
    "searchable" : false,
    "minLength" : 8,
    "userEditable" : true,
    "scope" : "private"
},
```

After your next reconciliation, the password policies that you just set up in OpenIDM should apply.

17.2. Storing Separate Passwords Per Linked Resource

OpenIDM supports storing multiple passwords in a managed user entry, to enable synchronization of different passwords on different external resources.

To store multiple passwords, you must extend the managed user schema to include additional properties for each target resource. You can set separate policies on each of these new properties, to ensure that the stored passwords adhere to the password policies of the specific external resources.

The following addition to a sample `managed.json` configuration shows an `ldapPassword` property that has been added to managed user objects. This property will be mapped to the password property on an LDAP system:

```
"ldapPassword" : {
    "title" : "Password",
    "type" : "string",
    "viewable" : false,
    "searchable" : false,
    "minLength" : 8,
    "userEditable" : true,
    "scope" : "private",
    "secureHash" : {
        "algorithm" : "SHA-256"
    },
    "policies" : [
        {
            "policyId" : "at-least-X-capitals",
      "params" : {
        "numCaps" : 2
      }
    },
    {
      "policyId" : "at-least-X-numbers",
      "params" : {
        "numNums" : 1
      }
    },
    {
      "policyId" : "cannot-contain-others",
      "params" : {
        "disallowedFields" : [
          "userName",
          "givenName",
          "sn"
        ]
      }
    },
    {
      "policyId" : "is-new",
      "params" : {
        "historyLength" : 2
      }
    }
  ]
},
```

This property definition shows that the `ldapPassword` will be hashed, with an SHA-256 algorithm, and sets the policy that will be applied to values of this property.

To use this custom managed object property and its policies to update passwords on an external resource, you must make the corresponding configuration and script changes in your deployment. For a detailed sample that implements multiple passwords, see "Storing Multiple Passwords For Managed Users" in the *Samples Guide*. That sample can also help you set up password history policies.

17.3. Generating Random Passwords

There are many situations when you might want to generate a random password for one or more user objects.

OpenIDM provides a way to customize your user creation logic to include a randomly generated password that complies with the default password policy. This functionality is included in the default crypto script, `bin/defaults/script/crypto.js`, but is not invoked by default. For an example of how this functionality might be used, see the `openidm/bin/defaults/script/ui/onCreateUser.js` script. The following section of that file (commented out by default) means that users created by using the Admin UI, or directly over the REST interface, will have a randomly generated, password added to their entry:

```
if (!object.password) {

    // generate random password that aligns with policy requirements
    object.password = require("crypto").generateRandomString([
        { "rule": "UPPERCASE", "minimum": 1 },
        { "rule": "LOWERCASE", "minimum": 1 },
        { "rule": "INTEGERS", "minimum": 1 },
        { "rule": "SPECIAL", "minimum": 1 }
    ], 16);

}
```

Comment out this section to invoke the random password generation when users are created. Note that changes made to scripts take effect after the time set in the `recompile.minimumInterval`, described in "Setting the Script Configuration".

The generated password can be encrypted, or hashed, in accordance with the managed user schema, defined in `conf/managed.json`. For more information, see "Encoding Attribute Values".

You can use this random string generation in a number of situations. Any script handler that is implemented in JavaScript can call the `generateRandomString` function.

17.4. Synchronizing User Passwords With LDAP Servers

Password synchronization ensures uniform password changes across the resources that store the password. After password synchronization, a user can authenticate with the same password on each resource. No centralized directory or authentication server is required for performing authentication. Password synchronization reduces the number of passwords users need to remember, so they can use fewer, stronger passwords.

OpenIDM can propagate passwords to the resources that store a user's password. In addition, you can download two plugins that intercept and synchronize passwords that are changed natively in OpenDJ and Active Directory.

When you use these plugins to synchronize passwords, set up password policy enforcement on the LDAP resource, rather than on OpenIDM. Alternatively, ensure that all password policies that are enforced are identical to prevent password updates on one resource from being rejected by OpenIDM or by another resource.

The password synchronization plugin intercepts password changes on the LDAP resource before the passwords are stored in encrypted form. The plugin then sends the intercepted password value to OpenIDM, using an HTTP POST request to patch the corresponding managed user object.

Note

The plugins do not use the LDAP connector to transmit passwords, but send a generic HTTP POST request with a `patch-by-query` action, similar to the following:

```
HTTP POST /managed/user?_action=patch-by-query&uid=bjensen&password=MyPassw0rd
```

If the OpenIDM instance is unavailable when a password is changed in either OpenDJ or Active Directory, the respective password plugin intercepts the change, encrypts the password, and stores the encrypted password in a JSON file. The plugin then checks whether the OpenIDM instance is available, at a predefined interval. When OpenIDM becomes available, the plugin performs a PATCH on the managed user record, to replace the password with the encrypted password stored in the JSON file.

To be able to synchronize passwords, both password synchronization plugins require that the corresponding managed user object exist in the OpenIDM repository.

The following sections describe how to use the password synchronization plugin for OpenDJ, and the corresponding plugin for Active Directory.

17.4.1. Synchronizing Passwords With OpenDJ

OpenIDM must be installed, and running before you continue with the procedures in this section.

Important

Password synchronization with OpenDJ requires keys to encrypt the password and certificates to secure communications with OpenIDM. In a production environment, you should use a certificate generated by a Certificate Authority. For evaluation or testing, you can use the self-signed certificate and key pair that are generated when you set up the OpenDJ server.

17.4.1.1. Establishing Secure Communication Between OpenIDM and OpenDJ

There are three possible modes of communication between the OpenDJ password synchronization plugin and OpenIDM:

- *SSL Authentication.* In this case, you must import the OpenIDM certificate into OpenDJ's truststore (either the self-signed certificate that is generated the first time OpenIDM is started, or a CA-signed certificate).

  For more information, see "To Import OpenIDM's Certificate into the OpenDJ Keystore".

- *Mutual SSL Authentication.* In this case, you must import the OpenIDM certificate into OpenDJ's truststore, as described in "To Import OpenIDM's Certificate into the OpenDJ Keystore", *and* import the OpenDJ certificate into OpenIDM's truststore, as described in "To Import OpenDJ's Certificate into the OpenIDM Truststore". You must also add the OpenDJ certificate DN as a value of the `allowedAuthenticationIdPatterns` property in your project's `conf/authentication.json` file. Mutual SSL authentication is the default configuration of the password synchronization plugin, and the one described in this procedure.

- *HTTP Basic Authentication.* In this case, the connection is secured using a username and password, rather than any exchange of certificates. OpenIDM supports basic authentication for testing purposes only. You should *not* use basic authentication in production. To configure the plugin for basic authentication, set the following properties in the plugin configuration:

- `ds-cfg-openidm-url`
- `ds-cfg-openidm-username`
- `ds-cfg-openidm-password`

For more information, see "Installing the OpenDJ Password Synchronization Plugin". Note that the password sync plugin also requires OpenIDM's certificate to encrypt the password such that it can be decrypted when it is replayed on OpenIDM. Therefore, even if you use HTTP basic authentication, you must import the OpenIDM certificate into OpenDJ's truststore, as described in "To Import OpenIDM's Certificate into the OpenDJ Keystore".

To Import OpenIDM's Certificate into the OpenDJ Keystore

Unless you use certificates signed by a well-known CA, you must export the certificate from OpenIDM's keystore into OpenDJ's keystore to secure communication from OpenIDM to OpenDJ.

OpenIDM generates a self-signed certificate the first time it starts up. This procedure uses the self-signed certificate to get the password synchronization plugin up and running. In a production environment, you should use a certificate that has been signed by a Certificate Authority.

OpenDJ does not enable a trust manager provider by default. For OpenDJ to trust the OpenIDM certificate, you must enable a trust manager provider and reference it from the password plugin configuration.

1. Export OpenIDM's generated self-signed certificate to a file, as follows:

   ```
   $ cd /path/to/openidm/security
   $ keytool \
    -export \
    -alias openidm-localhost \
    -file openidm-localhost.crt \
    -keystore keystore.jceks \
    -storetype jceks
   Enter keystore password: changeit
   Certificate stored in file <openidm-localhost.crt>
   ```

   The default OpenIDM keystore password is `changeit`.

2. Create and enable a trust manager provider in OpenDJ:

```
$ cd /path/to/opendj/bin
$ ./dsconfig create-trust-manager-provider \
 --hostname localhost \
 --port 4444 \
 --bindDN "cn=Directory Manager" \
 --bindPassword password \
 --provider-name PKCS12 \
 --type file-based \
 --set enabled:true \
 --set trust-store-type:PKCS12 \
 --set trust-store-file:config/keystore \
 --set trust-store-pin-file:config/keystore.pin \
 --trustAll \
 --no-prompt
```

3. Import the self-signed certificate into OpenDJ's keystore:

   ```
   $ cd /path/to/opendj/config
   $ keytool \
    -import \
    -alias openidm-localhost \
    -file /path/to/openidm/security/openidm-localhost.crt \
    -keystore keystore \
    -storepass:file keystore.pin \
    -storetype PKCS12 \
    -noprompt
   Certificate was added to keystore
   ```

To Import OpenDJ's Certificate into the OpenIDM Truststore

For mutual authentication, you must import OpenDJ's certificate into the OpenIDM truststore.

OpenDJ generates a self-signed certificate when you set up communication over LDAPS. This procedure uses the self-signed certificate to get the password synchronization plugin up and running. In a production environment, you should use a certificate that has been signed by a Certificate Authority.

1. Export OpenDJ's generated self-signed certificate to a file, as follows:

   ```
   $ cd /path/to/opendj/config
   $ keytool \
    -export \
    -alias server-cert \
    -file server-cert.crt \
    -keystore keystore \
    -storepass:file keystore.pin
   Certificate stored in file <server-cert.crt>
   ```

2. Import the OpenDJ self-signed certificate into OpenIDM's truststore:

```
$ cd /path/to/openidm/security
$ keytool \
 -importcert \
 -alias server-cert \
 -keystore truststore \
 -storepass changeit \
 -file /path/to/opendj/config/server-cert.crt
Owner: CN=localhost, O=OpenDJ RSA Self-Signed Certificate
Issuer: CN=localhost, O=OpenDJ RSA Self-Signed Certificate
Serial number: 41cefe38
Valid from: Thu Apr 14 10:17:39 SAST 2016 until: Wed Apr 09 10:17:39 SAST 2036
Certificate fingerprints:
  MD5:  0D:BC:44:B3:C4:98:90:45:97:4A:8D:92:84:2B:FC:60
  SHA1: 35:10:B8:34:DE:38:59:AA:D6:DD:B3:44:C2:14:90:BA:BE:5C:E9:8C
  SHA256: 43:66:F7:81:3C:0D:30:26:E2:E2:09:...9F:5E:27:DC:F8:2D:42:79:DC:80:69:73:44:12:87
  Signature algorithm name: SHA1withRSA
  Version: 3
Trust this certificate? [no]: yes
Certificate was added to keystore
```

3. Add the certificate DN as a value of the `allowedAuthenticationIdPatterns` property for the `CLIENT_CERT` authentication module, in your project's `conf/authentication.json` file.

   For example, if you are using the OpenDJ self-signed certificate, add the DN `"CN=localhost, O=OpenDJ RSA Self-Signed Certificate, OU=None, L=None, ST=None, C=None"`, as shown in the following excerpt:

```
$ more /path/to/openidm/project-dir/conf/authentication.json
...
{
     "name" : "CLIENT_CERT",
     "properties" : {
         "queryOnResource" : "security/truststore",
         "defaultUserRoles" : [
             "openidm-cert"
         ],
         "allowedAuthenticationIdPatterns" : [
             "CN=localhost, O=OpenDJ RSA Self-Signed Certificate, OU=None, L=None, ST=None, C=None"
         ]
     },
     "enabled" : true
}
     ...
```

17.4.1.2. Installing the OpenDJ Password Synchronization Plugin

The following steps install the password synchronization plugin on an OpenDJ directory server that is running on the same host as OpenIDM (localhost). If you are running OpenDJ on a different host, use the fully qualified domain name instead of `localhost`.

You must use the plugin version that corresponds to your OpenIDM and OpenDJ versions. For more information, see "Supported Connectors, Connector Servers, and Plugins" in the *Release Notes*. This procedure assumes that you are using OpenIDM 5 and OpenDJ 5.

1. Download the password synchronization plugin (`IDM-DS-account-change-notification-hander-5.zip`) from the ForgeRock BackStage site.

2. Extract the contents of the .zip file to the directory where OpenDJ is installed:

```
$ unzip ~/Downloads/IDM-DS-account-change-notification-hander-5.zip -d /path/to/opendj/
Archive:  IDM-DS-account-change-notification-hander-5.zip
   creating: opendj/
   ...
```

3. Restart OpenDJ to load the additional schema from the password synchronization plugin:

```
$ cd /path/to/opendj/bin
$ ./stop-ds --restart
Stopping Server..
.
...
[23/Nov/2016:13:19:11 +0200] category=EXTENSIONS severity=NOTICE
 msgID=org.opends.messages.extension.571 msg=Loaded extension from file
 '/path/to/opendj/lib/extensions/openidm-account-change-handler.jar' (build version, revision
 1)
...
[23/Nov/2016:13:19:43 +0200] category=CORE severity=NOTICE msgID=org.opends.messages.core
.139
... The Directory Server has started successfully
```

4. Configure the password synchronization plugin.

   The password plugin configuration is specified in the `openidm-accountchange-plugin-sample-config` file which should have been extracted to `path/to/opendj/config` when you extracted the plugin.

   Use a text editor to update the configuration, for example:

```
$ cd /path/to/opendj/config
$ more openidm-accountchange-plugin-sample-config
dn: cn=OpenIDM Notification Handler,cn=Account Status Notification Handlers,cn=config
objectClass: top
objectClass: ds-cfg-account-status-notification-handler
objectClass: ds-cfg-openidm-account-status-notification-handler
cn: OpenIDM Notification Handler
...
```

   At a minimum, you *must* set the value of the `ds-cfg-trust-manager-provider` property, as the default value references a trust manager provider that does not exist. In addition, you can configure the following elements of the plugin:

   `ds-cfg-enabled`

   Specifies whether the plugin is enabled.

   Default value: `true`

`ds-cfg-attribute`

The attribute in OpenIDM that stores user passwords. This property is used to construct the patch request on the OpenIDM managed user object.

Default value: `password`

`ds-cfg-query-id`

The query-id for the patch-by-query request. This query must be defined in the repository configuration.

Default value: `for-userName`

`ds-cfg-attribute-type`

Specifies zero or more attribute types that the plugin will send along with the password change. If no attribute types are specified, only the DN and the new password will be synchronized to OpenIDM.

Default values: `entryUUID` and `uid`

`ds-cfg-log-file`

The directory where plugin log files are written, and where modified passwords are written when the plugin cannot contact OpenIDM. The default location is a directory named `/path/to/opendj/logs/pwsync`. Passwords in this directory will be encrypted.

Default value: `logs/pwsync`

Note that this setting has no effect if `ds-cfg-update-interval` is set to `0 seconds`.

`ds-cfg-update-interval`

The interval, in seconds, at which password changes are propagated to OpenIDM. If this value is 0, updates are made synchronously in the foreground, and no encrypted passwords are stored in the `ds-cfg-log-file` directory.

Default value: `0 seconds`

`ds-cfg-openidm-url`

The endpoint at which the plugin should find OpenIDM managed user accounts.

Default value: `https://localhost:8444/openidm/managed/user`

For HTTP basic authentication, specify the `http` protocol in the URL, and a non-mutual authentication port, for example `http://localhost:8080/openidm/managed/user`.

`ds-cfg-ssl-cert-nickname`

The alias of the client certificate in the OpenDJ keystore. If you configured LDAPS when you set up of OpenDJ, the default client key alias is `server-cert`. Do not use this self-signed certificate in production.

Default value: `server-cert`

`ds-cfg-private-key-alias`

The alias of the private key that should be used by OpenIDM to decrypt the session key.

Default value: `openidm-localhost`

`ds-cfg-certificate-subject-dn`

The certificate subject DN of the OpenIDM private key. The default configuration assumes that you are using the self-signed certificate that is generated when OpenIDM first starts.

Default value: `CN=localhost, O=OpenIDM Self-Signed Certificate, OU=None, L=None, ST=None, C=None`

`ds-cfg-key-manager-provider`

The OpenDJ key manager provider. The key manager provider specified here must be enabled. If you do not specify a keystore, the default configuration references a PKCS#12 keystore file that contains a self-signed certificate.

Default value: `cn=Default Key Manager,cn=Key Manager Providers,cn=config`

`ds-cfg-trust-manager-provider`

The OpenDJ trust manager provider. The trust manager provider specified here must be enabled. Before you use the plugin, you must create a trust manager provider and set this value accordingly.

Default value: `cn=PKCS12,cn=Trust Manager Providers,cn=config`

`ds-cfg-openidm-username`

An OpenIDM administrative username that the plugin will use to make REST calls to OpenIDM.

Default value: `openidm-admin`

For SSL authentication and HTTP basic authentication, the user specified here must have the rights to patch managed user objects.

This property is commented out by default, because the default configuration assumes mutual SSL authentication. If you use HTTP or SSL authentication, you must uncomment this property.

`ds-cfg-openidm-password`

The password of the OpenIDM administrative user specified by the previous property.

Default value: `openidm-admin`

This property is commented out by default, because the default configuration assumes mutual SSL authentication. If you use HTTP or SSL authentication, you must uncomment this property.

If you are using mutual authentication, the default configuration might be suitable for your deployment. Otherwise, update the plugin configuration as required.

5. Add the plugin configuration to OpenDJ's configuration:

```
$ cd /path/to/opendj/bin
$ ./ldapmodify \
 --port 1389 \
 --hostname `hostname` \
 --bindDN "cn=Directory Manager" \
 --bindPassword "password" \
 --filename ../config/openidm-accountchange-plugin-sample-config
Processing ADD request for cn=OpenIDM Notification Handler,cn=Account Status
Notification Handlers,cn=config
ADD operation successful for DN cn=OpenIDM Notification Handler,cn=Account
Status Notification Handlers,cn=config
```

6. Restart OpenDJ for the new configuration to take effect:

```
$ ./stop-ds --restart
Stopping Server..
.
...
[23/Nov/2016:13:25:50 +0200] category=EXTENSIONS severity=NOTICE
 msgID=org.opends.messages.extension.571 msg=Loaded extension from file
 '/path/to/opendj/lib/extensions/opendj-openidm-account-change-notification-handler-4.0.0-sources.jar'
 (build 4.0.0, revision
 1)
...
[23/Nov/2016:13:26:27 +0200] category=CORE severity=NOTICE msgID=org.opends.messages.core.139
 msg=The Directory Server has sent an alert notification generated by
 class org.opends.server.core.DirectoryServer (alert type org.opends.server.DirectoryServerStarted,
 alert ID org.opends.messages.core-135): The Directory Server has started successfully
```

7. Adjust your OpenDJ password policy configuration to use the password synchronization plugin.

   The following command adjusts the default password policy:

```
$ cd /path/to/opendj/bin
$ ./dsconfig \
 set-password-policy-prop \
 --port 4444 \
 --hostname `hostname` \
 --bindDN "cn=Directory Manager" \
 --bindPassword password \
 --policy-name "Default Password Policy" \
 --set account-status-notification-handler:"OpenIDM Notification Handler" \
 --trustStorePath ../config/admin-truststore \
 --no-prompt
```

Password synchronization should now be configured and working.

You can test that the setup has been successful as follows:

1. Change a user password in OpenDJ.

   The new password should be synchronized to the corresponding OpenIDM managed user account.

2. Make sure that the `PASSTHROUGH_AUTHENTICATION` module is disabled (to ensure that the user is not authenticating with her OpenDJ credentials) and that the `MANAGED_USER` module is enabled (so that the user can authenticate with her managed user credentials).

3. You should now be able to log into the Self Service UI (https://localhost:8443/#login/) as that user ID, with the new password.

17.4.1.3. Uninstalling the OpenDJ Password Synchronization Plugin

To uninstall the plugin, change the OpenDJ configuration as follows:

1. Reset your OpenDJ password policy configuration so that it no longer uses the password synchronization plugin.

   The following command resets the default password policy:

```
$ cd /path/to/opendj/bin
$ ./dsconfig \
 set-password-policy-prop \
 --port 4444 \
 --hostname `hostname` \
 --bindDN "cn=Directory Manager" \
 --bindPassword password \
 --policy-name "Default Password Policy" \
 --reset account-status-notification-handler \
 --trustStorePath ../config/admin-truststore \
 --no-prompt
```

2. Delete the OpenIDM Notification Handler from the OpenDJ configuration:

```
$ ./dsconfig \
 delete-account-status-notification-handler \
 --port 4444 \
 --hostname `hostname` \
 --bindDN "cn=Directory Manager" \
 --bindPassword password \
 --handler-name "OpenIDM Notification Handler" \
 --trustStorePath ../config/admin-truststore \
 --no-prompt
The Account Status Notification Handler was deleted successfully
```

3. Remove the password synchronization plugin from the OpenDJ extensions:

   ```
   $ cd /path/to/opendj
   $ rm lib/extensions/opendj-openidm-account-change-notification-handler*
   ```

4. Restart OpenDJ for the new configuration to take effect:

   ```
   $ cd /path/to/opendj/bin
   $ ./stop-ds --restart
   ```

17.4.2. Synchronizing Passwords With Active Directory

Use the Active Directory password synchronization plugin to synchronize passwords between OpenIDM and Active Directory (on systems running at least Microsoft Windows Server 2003).

Install the plugin on Active Directory domain controllers (DCs) to intercept password changes, and send the password values to OpenIDM over an encrypted channel. You must have Administrator privileges to install the plugin. In a clustered Active Directory environment, you must install the plugin on all DCs.

17.4.2.1. Installing the Active Directory Password Synchronization Plugin

The following steps install the password synchronization on an Active directory server:

1. Download the Active Directory password synchronization plugin from the ForgeRock BackStage site.

2. Install the plugin using one of the following methods:

   - Double-click the setup file to launch the installation wizard.

   - Alternatively, from a command line, start the installation wizard with the `IDM-setup-5.exe` command. To save the settings in a configuration file, use the `/saveinf` switch as follows.

     ```
     C:\Path\To > IDM-setup-5.exe /saveinf=C:\temp\adsync.inf
     ```

   - If you have a configuration file with installation parameters, you can install the password plugin in silent mode as follows:

     ```
     C:\Path\To > IDM-setup-5.exe /verysilent /loadinf=C:\temp\adsync.inf
     ```

3. Provide the following information during the installation. You must accept the license agreement shown to proceed with the installation.

OpenIDM Connection information

- *OpenIDM URL.* Enter the URL where OpenIDM is deployed, including the query that targets each user account. For example:

```
https://localhost:8444/openidm/managed/user?_action=patch&_queryId=for-userName&uid=
${samaccountname}
```

- *OpenIDM User Password attribute.* The password attribute for the `managed/user` object, such as `password`.

  If the `password` attribute does not exist in the `managed/user` object on OpenIDM, the password sync service will return an error when it attempts to replay a password update that has been made in Active Directory. If your managed user objects do not include passwords, you can add an `onCreate` script to the Active Directory > Managed Users mapping that sets an empty password when managed user accounts are created. The following excerpt of a `sync.json` file shows such a script in the mapping:

```
"mappings" : [
 {
   "name" : "systemAdAccounts_managedUser",
   "source" : "system/ad/account",
   "target" : "managed/user",
   "properties" : [
     {
       "source" : "sAMAccountName",
       "target" : "userName"
     }
   ],
   "onCreate" : {
     "type" : "text/javascript",
     "source" : "target.password=''"
   },
...
```

  The onCreate script creates an empty password in the `managed/user` object, so that the password attribute exists and can be patched.

OpenIDM Authentication Parameters

Provide the following information:

- *User name.* Enter name of an administrative user that can authenticate to OpenIDM, for example, `openidm-admin`.
- *Password.* Enter the password of the user that authenticates to OpenIDM, for example, `openidm-admin`.
- *Select authentication type.* Select the type of authentication that Active Directory will use to authenticate to OpenIDM.

For plain HTTP authentication, select `OpenIDM Header`. For SSL mutual authentication, select `Certificate`.

Certificate authentication settings

If you selected `Certificate` as the authentication type on the previous screen, specify the details of the certificate that will be used for authentication.

- *Select Certificate file.* Browse to select the certificate file that Active Directory will use to authenticate to OpenIDM. The certificate file must be configured with an appropriate encoding, cryptographic hash function, and digital signature. The plugin can read a public or a private key from a PKCS12 archive file.

  For production purposes, you should use a certificate that has been issued by a Certificate Authority. For testing purposes, you can generate a self-signed certificate. Whichever certificate you use, you must import that certificate into OpenIDM's trust store.

  To generate a self-signed certificate for Active Directory, follow these steps:

  1. On the Active Directory host, generate a private key, which will be used to generate a self-signed certificate with the alias `ad-pwd-plugin-localhost`:

     ```
     > keytool.exe ^
      -genkey ^
      -alias ad-pwd-plugin-localhost ^
      -keyalg rsa ^
      -dname "CN=localhost, O=AD-pwd-plugin Self-Signed Certificate" ^
      -keystore keystore.jceks ^
      -storetype JCEKS
     Enter keystore password: changeit
     Re-enter new password: changeit
     Enter key password for <ad-pwd-plugin-localhost>
           <RETURN if same as keystore password>
     ```

  2. Now use the private key, stored in the `keystore.jceks` file, to generate the self-signed certificate:

     ```
     > keytool.exe ^
      -selfcert ^
      -alias ad-pwd-plugin-localhost ^
      -validity 365 ^
      -keystore keystore.jceks ^
      -storetype JCEKS ^
      -storepass changeit
     ```

  3. Export the certificate. In this case, the **keytool** command exports the certificate in a PKCS12 archive file format, used to store a private key with a certificate:

```
> keytool.exe ^
 -importkeystore ^
 -srckeystore keystore.jceks ^
 -srcstoretype jceks ^
 -srcstorepass changeit ^
 -srckeypass changeit ^
 -srcalias ad-pwd-plugin-localhost ^
 -destkeystore ad-pwd-plugin-localhost.p12 ^
 -deststoretype PKCS12 ^
 -deststorepass changeit ^
 -destkeypass changeit ^
 -destalias ad-pwd-plugin-localhost ^
 -noprompt
```

4. The PKCS12 archive file is named `ad-pwd-plugin-localhost.p12`. Import the contents of the keystore contained in this file to the system that hosts OpenIDM. To do so, import the PKCS12 file into the OpenIDM keystore file, named `truststore`, in the `/path/to/openidm/security` directory.

   On the machine that is running OpenIDM, enter the following command:

   ```
   $ keytool \
    -importkeystore \
    -srckeystore /path/to/ad-pwd-plugin-localhost.p12 \
    -srcstoretype PKCS12 \
    -destkeystore truststore \
    -deststoretype JKS
   ```

- *Password to open the archive file with the private key and certificate.* Specify the keystore password (`changeit`, in the previous example).

Password Encryption settings

Provide the details of the certificate that will be used to encrypt password values.

- *Select certificate file.* Browse to select the certificate that will be used for password encryption. The certificate must be in PKCS12 format.

  For evaluation purposes, you can use a self-signed certificate, as described earlier. For production purposes, you should use a certificate that has been issued by a Certificate Authority.

  Whichever certificate you use, the certificate must be imported into OpenIDM's keystore, so that OpenIDM can locate the key with which to decrypt the data. To import the certificate into OpenIDM's keystore, `keystore.jceks`, run the following command on the OpenIDM host (UNIX):

  ```
  $ keytool \
   -importkeystore \
   -srckeystore /path/to/ad-pwd-plugin-localhost.p12 \
   -srcstoretype PKCS12 \
   -destkeystore /path/to/openidm/security/keystore.jceks \
   -deststoretype jceks
  ```

- *Private key alias.* Specify the alias for the certificate, such as `ad-pwd-plugin-localhost`.
- *Password to open certificate file.* Specify the password to access the PFX keystore file, such as `changeit`, from the previous example.
- *Select encryption standard.* Specify the encryption standard that will be used when encrypting the password value (AES-128, AES-192, or AES-256).

Data storage

Provide the details for the storage of encrypted passwords in the event that OpenIDM is not available when a password modification is made.

- Select a secure directory in which the JSON files that contain encrypted passwords are queued. The server should prevent access to this folder, except access by the `Password Sync service`. The path name cannot include spaces.
- *Directory poll interval (seconds).* Enter the number of seconds between calls to check whether OpenIDM is available, for example, `60`, to poll OpenIDM every minute.

Log storage

Provide the details of the messages that should be logged by the plugin.

- Select the location to which messages should be logged. The path name cannot include spaces.
- *Select logging level.* Select the severity of messages that should be logged, either `error`, `info`, `warning`, `fatal`, or `debug`.

Select Destination Location

Setup installs the plugin in the location you select, by default `C:\Program Files\OpenIDM Password Sync`.

4. After running the installation wizard, restart the computer.
5. If you selected to authenticate over plain HTTP in the previous step, your setup is now complete.

   If you selected to authenticate with mutual authentication, complete this step.

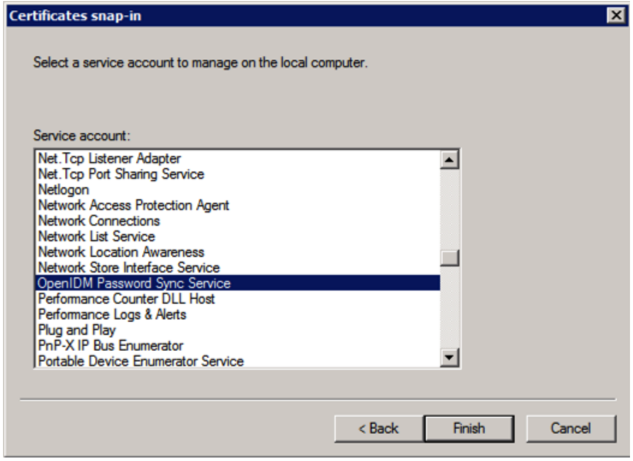
   - The Password Sync Service uses Windows certificate stores to verify OpenIDM's identity. The certificate that OpenIDM uses must therefore be added to the list of trusted certificates on the Windows machine.

     For production purposes, you should use a certificate that has been issued by a certificate authority. For test purposes, you can use the self-signed certificate that is generated by OpenIDM on first startup.

To add the OpenIDM certificate to the list of trusted certificates, use the Microsoft Management Console.

1. Select Start and type `mmc` in the Search field.
2. In the Console window, select File > Add/Remove Snap-in.
3. From the left hand column, select Certificates and click Add.
4. Select My user account, and click Finish.
5. Repeat the previous two steps for Service account and Computer account.

   For Service account, select Local computer, then select OpenIDM Password Sync Service.

   Certificates snap-in

   Select a service account to manage on the local computer.

   Service account:

   Net.Tcp Listener Adapter
   Net.Tcp Port Sharing Service
   Netlogon
   Network Access Protection Agent
   Network Connections
   Network List Service
   Network Location Awareness
   Network Store Interface Service
   OpenIDM Password Sync Service
   Performance Counter DLL Host
   Performance Logs & Alerts
   Plug and Play
   PnP-X IP Bus Enumerator
   Portable Device Enumerator Service

   For Computer account, select Local computer.

6. Click Finish when you have added the three certificate snap-ins.
7. Still in the Microsoft Management Console, expand Certificates - Current User > Personal and select Certificates.
8. Select Action > All Tasks > Import to open the Certificate Import Wizard.
9. Browse for the OpenIDM certificate (`openidm-localhost.crt` by default, if you use OpenIDM's self-signed certificate).
10. Enter the Password for the certificate (`changeit` by default, if you use OpenIDM's self-signed certificate).
11. Accept the default for the Certificate Store.

12. Click Finish to complete the import.
13. Repeat the previous six steps to import the certificate for:

    `Certificates - Current User > Trusted Root Certification Authorities`
    `Certificates - Service > OpenIDM Password Sync\Personal`
    `Certificates - Service > OpenIDM Password Sync\Trusted Root Certification Authorities`
    `Certificates > Local Computer > Personal`
    `Certificates > Local Computer > Trusted Root Certification Authorities`

Password synchronization should now be configured and working. To test that the setup has been successful, change a user password in Active Directory. That password should be synchronized to the corresponding OpenIDM managed user account, and you should be able to query the user's own entry in OpenIDM using the new password.

17.4.2.2. Changing the Password Synchronization Plugin Configuration After Installation

If you need to change any settings after installation, access the settings using the Registry Editor under HKEY_LOCAL_MACHINE > SOFTWARE > ForgeRock > OpenIDM > PasswordSync.

For information about creating registry keys, see Configure a Registry Item in the Windows documentation.

You can change the following registry keys to reconfigure the plugin:

Keys to set the method of authentication

- `authType` sets the authentication type.

  For plain HTTP or SSL authentication using OpenIDM headers, set `authType` to `idm`.

  For SSL mutual authentication using a certificate, set `authType` to `cert`.

  By default, the plugin does not validate the OpenIDM certificate. To configure this validation, set the following registry key: `netSslVerifyPeer = True`.

- `authToken0` sets the username or certificate path for authentication.

  For example, for plain HTTP or SSL authentication, set `authToken0` to `openidm-admin`.

  For SSL mutual authentication, set `authToken0` to the certificate path, for example `path/to/certificate/cert.p12`. Only PKCS12 format certificates are supported.

- `authToken1` sets the password for authentication.

  For example, for plain HTTP or SSL authentication, set `authToken1` to `openidm-admin`.

  For SSL mutual authentication, set `authToken1` to the password to the keystore.

Keys to set encryption of captured passwords

- `certFile` sets the path to the keystore used for encrypting captured passwords, for example `path/to/keystore.p12`. Only PKCS12 keystores are supported.
- `certPassword` sets the password to the keystore.
- `keyAlias` specifies the alias that is used to encrypt passwords.
- `keyType` sets the cypher algorithm, for example `aes128`

Keys to set encryption of sensitive registry values

For security reasons, you should encrypt the values of the `authToken1` and `certPassword` keys. These values are encrypted automatically when the plugin is installed, but when you change the settings, you can encrypt the values manually by setting the `encKey` registry key.

> **Note**
>
> If you do not want to encrypt the values of the `authToken1` and `certPassword` keys, you *must* remove the `encKey` from the registry, otherwise the plugin will use the value stored in that key to decrypt those values (even if they include an empty string).

To encrypt the values of the `authToken1` and `certPassword` keys:

1. Optionally, generate a new encryption key by running the following command:

   ```
   idmsync.exe --key
   ```

2. Encrypt the values of the sensitive registry keys as follows:

   ```
   idmsync.exe --encrypt "key-value" "authToken1Value"
   idmsync.exe --encrypt "key-value" "certPasswordValue"
   ```

3. Replace the existing values of the `encyKey`, `authToken1` and `certPassword` keys with the values you generated in the previous step.

   If you do not want to generate a new encryption key, skip the first step and use the existing encryption key from the registry.

Keys to set the OpenIDM connection information

The password synchronization plugin assumes that the Active Directory user attribute is `sAMAccountName`. The default attribute will work in most deployments. If you cannot use the `sAMAccountName` attribute to identify the Active Directory user, set the following registry keys on your Active Directory server, specifying an alternative attribute. These examples use the `employeeId` attribute instead of `sAMAccountName`:

- `userAttribute = employeeId`
- `userSearchFilter = (&(objectClass=user)(sAMAccountName=%s))`

- `idmURL = https://localhost:8444/openidm/managed/user?_action=patch&_queryId=for-userName&uid=${employeeId}`

Keys to set the behavior when OpenIDM is unavailable

When OpenIDM is unavailable, or when an update fails, the password synchronization plugin stores the user password change a JSON file on the Active Directory system and attempts to resend the password change at regular intervals.

After installation, you can change the behaviour by setting the following registry keys:

Also the netTimeout in milliseconds can be set.

- `dataPath` - the location where the plugin stores the unsent changes. When any unsent changes have been delivered successfully, files in this path are deleted. The plugin creates one file for each user. This means that if a user changes his password three times in a row, you will see only one file containing the last change.
- `pollEach` - the interval (in seconds) at which the plugin attempts to resend the changes.
- `netTimeout` - the length of time (in milliseconds) after which the plugin stops attempting a connection.

Keys to set the logging configuration

- `logPath` sets the path to the log file.
- `logSize` - the maximum log size (in Bytes) before the log is rotated. When the log file reaches this size, it is renamed `idm.log.0` and a new `idm.log` file is created.
- `logLevel` sets the logging level, `debug`, `info`, `warning`, `error`, or `fatal`.

Key to configure support for older OpenIDM versions

If the `idm2only` key is set to `true`, the plugin uses an old version of the patch request. This key *must not* exist in the registry for OpenIDM versions 3.0 and later.

If you change any of the registry keys associated with the password synchronization plugin, run the **idmsync.exe --validate** command to make sure that all of the keys have appropriate values.

The password synchronization plugin is installed and run as a service named OpenIDM Password Sync Service. You can use the Windows Service Manager to start and stop the service. To start or stop the plugin manually, run the **idmsync.exe --start** or **idmsync.exe --stop** command.

Chapter 18
Managing Authentication, Authorization and Role-Based Access Control

OpenIDM provides a flexible authentication and authorization mechanism, based on REST interface URLs and on managed roles. This chapter describes how to configure the supported authentication modules, and how roles are used to support authentication, authorization, and access control.

18.1. The Authentication Model

You *must* authenticate before you can access the OpenIDM REST interface. User self-registration requires anonymous access. For this purpose, OpenIDM includes an `anonymous` user, with the password `anonymous`. For more information, see "Internal Users".

OpenIDM supports an enhanced authentication mechanism over the REST interface, that is compatible with the AJAX framework. Although OpenIDM understands the authorization header of the HTTP basic authorization contract, it deliberately does not utilize the full contract. In other words, it does not cause the browser built in mechanism to prompt for username and password. However, OpenIDM does understand utilities such as `curl` that can send the username and password in the Authorization header.

In general, the HTTP basic authentication mechanism does not work well with client side web applications, and applications that need to render their own login screens. Because the browser stores and sends the username and password with each request, HTTP basic authentication has significant security vulnerabilities. OpenIDM therefore supports sending the username and password via the authorization header, and returns a token for subsequent access.

This document uses the OpenIDM authentication headers in all REST examples, for example:

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

OpenIDM supports RFC 5987-encoded characters in the authentication headers (`X-OpenIDM-Username`, `X-OpenIDM-Password`, and `X-OpenIDM-Reauth-Password`). This support enables you to use non-ASCII characters in these header values. The RFC 5987-encoding is automatically detected and decoded when present. As per the RFC 5987 specification, the following character sets are supported:

- UTF-8
- ISO-8859-1

The following command shows a request for a user (openidm-admin) whose password is Passw£rd123. The Unicode £ sign (U+00A3) is encoded into the octet sequence C2 A3 using UTF-8 character encoding, then percent-encoded.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: UTF-8''Passw%C2%A3rd123" \
 --request GET \
 "http://localhost:8080/openidm/managed/user?_queryId=query-all-ids"
```

For more information, see RFC 5987.

For more information about the OpenIDM authentication mechanism, see "Use Message Level Security".

18.1.1. Authenticating OpenIDM Users

OpenIDM stores two types of users in its repository - internal users and managed users. The way in which both of these user types are authenticated is defined in your project's conf/authentication.json file.

18.1.1.1. Internal Users

OpenIDM creates two internal users by default: anonymous and openidm-admin. These internal user accounts are separated from other user accounts to protect them from any reconciliation or synchronization processes.

OpenIDM stores internal users and their role membership in a table in the repository. The two default internal users have the following functions:

anonymous

> This user enables anonymous access to OpenIDM, for users who do not have their own accounts. The anonymous user has limited rights within OpenIDM. By default, the anonymous user has the openidm-reg role, and can be used to allow self-registration. For more information about self-registration, see "The End User and Commons User Self-Service".

openidm-admin

> This user serves as the top-level administrator. After installation, the openidm-admin user has full access, and provides a fallback mechanism in the event that other users are locked out of their accounts. Do not use openidm-admin for regular tasks. Under normal circumstances, the openidm-admin account does not represent a regular user, so audit log records for this account do not represent the actions of any real person.
>
> The default password for the openidm-admin user (also openidm-admin) is not encrypted, and is not secure. In production environments, you must change this password to a more secure one, as described in the following section. The new password will be encoded using a salted hash algorithm, when it is changed.

18.1.1.1.1. Managing Internal Users Over REST

Like any other user in the repository, you can manage internal users over the REST interface.

To list the internal users over REST, query the `repo` endpoint as follows:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET  \
 "http://localhost:8080/openidm/repo/internal/user?_queryId=query-all-ids"
{
  "result": [
    {
      "_id": "openidm-admin",
      "_rev": "1"
    },
    {
      "_id": "anonymous",
      "_rev": "1"
    }
  ],
  "resultCount": 2,
  "pagedResultsCookie": null,
  "totalPagedResultsPolicy": "NONE",
  "totalPagedResults": -1,
  "remainingPagedResults": -1
}
```

To query the details of an internal user, include the user's ID in the request, for example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET  \
 "http://localhost:8080/openidm/repo/internal/user/openidm-admin"
{
  "_id": "openidm-admin",
  "_rev": "1",
  "roles": [
    {
      "_ref": "repo/internal/role/openidm-admin"
    },
    {
      "_ref": "repo/internal/role/openidm-authorized"
    }
  ],
  "userName": "openidm-admin",
  "password": "openidm-admin"
}
```

To change the password of the default administrative user, send a PUT request to the user object. The following example changes the password of the `openidm-admin` user to `Passw0rd`:

```
$ curl \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request PUT \
 --data '{
     "_id": "openidm-admin",
     "roles": [
         {
             "_ref": "repo/internal/role/openidm-admin"
         },
         {
             "_ref": "repo/internal/role/openidm-authorized"
         }
     ],
     "userName": "openidm-admin",
     "password": "Passw0rd"
 }' \
 "http://localhost:8080/openidm/repo/internal/user/openidm-admin"
{
  "_id": "openidm-admin",
  "_rev": "2",
  "roles": [
    {
      "_ref": "repo/internal/role/openidm-admin"
    },
    {
      "_ref": "repo/internal/role/openidm-authorized"
    }
  ],
  "userName": "openidm-admin",
  "password": {
    "$crypto": {
      "value": {
        "algorithm": "SHA-256",
        "data": "spKRPPYpDFZZWuJsOQa03vT2Gf+pFYUW8Zj6eCXuvMj19wZasYmdI2sCOrmmxiUQ"
      },
      "type": "salted-hash"
    }
  }
}
```

18.1.1.2. Managed Users

External users that are managed by OpenIDM are known as managed users.

The table in which managed users are stored depends on the type of repository. For JDBC repositories, OpenIDM stores managed users in the managed objects table, named `managedobjects`, and indexes those objects in a table named `managedobjectproperties`.

For an OrientDB repository, managed objects are stored in the table `managed_user`.

OpenIDM provides RESTful access to managed users, at the context path `/openidm/managed/user`. For more information, see "Managing Users Over REST".

18.1.1.3. Authenticating Internal and Managed Users

By default, the attribute names that are used to authenticate managed and internal users are `username` and `password`, respectively. However, you can explicitly define the properties that constitute usernames, passwords or roles with the propertyMapping object in the conf/authentication.json file. The following excerpt of the authentication.json file shows the default property mapping object:

```
...
    "propertyMapping" : {
        "authenticationId" : "username",
        "userCredential" : "password",
        "userRoles" : "roles"
    },
 ...
```

If you change the attribute names that are used for authentication, you must adjust the following authentication queries (defined in the repository configuration file, openidm/conf/repo.*repo-type*.json).

Two queries are defined by default.

credential-internaluser-query

This query uses the username attribute for login, for internal users. For example, the following credential-internaluser-query is defined in the default repository configuration file for a MySQL repository.

```
"credential-internaluser-query" : "SELECT objectid, pwd, roles FROM
        ${_dbSchema}.${_table} WHERE objectid = ${username}",
```

credential-query

This query uses the username attribute for login, for managed users. For example, the following credential-query is defined in the default repository configuration file for a MySQL repository.

```
"credential-query" : "SELECT * FROM ${_dbSchema}.${_table} WHERE
        objectid = ${username} and accountStatus = 'active'",
```

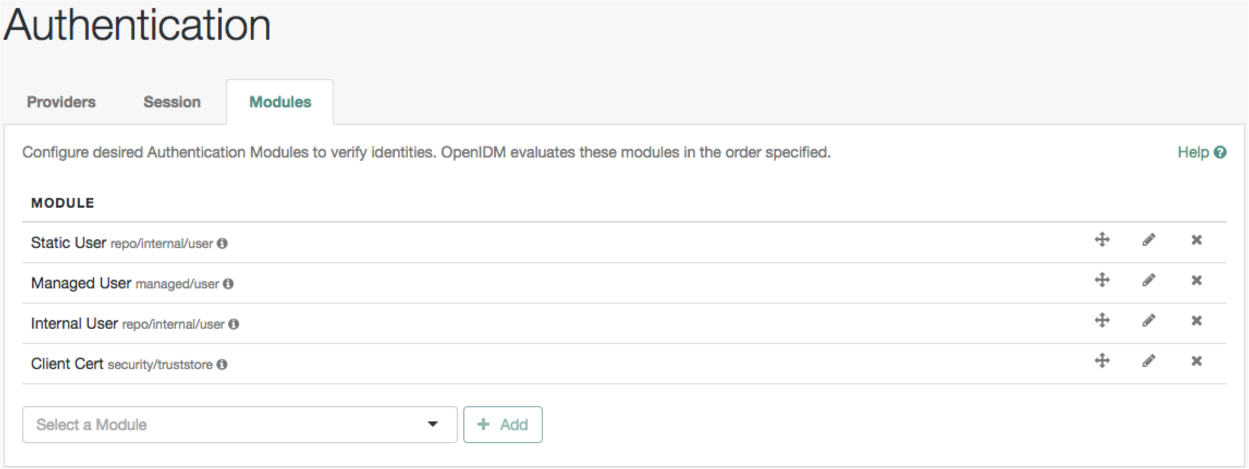
The query that is used for a particular resource is specified by the queryId property in the authentication.json file. The following sample excerpt of that file shows that the credential-query is used when validating managed user credentials.

```
{
    "queryId" : "credential-query",
    "queryOnResource" : "managed/user",
...
}
```

18.1.2. Supported Authentication and Session Modules

The authentication configuration is defined in conf/authentication.json. This file configures the methods by which a user request is authenticated. It includes both session and authentication module configuration.

You can review and configure supported local modules in the Admin UI. To do so, log into https://localhost:8443/admin, and select Configure > Authentication. Choose Local when asked to select an authentication provider, and select the Session and then the Modules tab.



Authentication

Configure desired Authentication Modules to verify identities. OpenIDM evaluates these modules in the order specified.

MODULE
Static User repo/internal/user
Managed User managed/user
Internal User repo/internal/user
Client Cert security/truststore

Whenever you modify an authentication module in the Admin UI, that may affect your current session. OpenIDM prompts you with the following message:

```
Your current session may be invalid. Click here
   to logout and re-authenticate.
```

When you select the *Click here* link, OpenIDM logs you out of any current session and returns you to the login screen.

18.1.2.1. Supported Session Module

At this time, OpenIDM includes one supported session module. The JSON Web Token session module configuration specifies keystore information, and details about the session lifespan. The default `JWT_SESSION` configuration is as follows:

```
"sessionModule" : {
    "name" : "JWT_SESSION",
        "properties" : {
            "keyAlias" : "&{openidm.https.keystore.cert.alias}",
            "privateKeyPassword" : "&{openidm.keystore.password}",
            "keystoreType" : "&{openidm.keystore.type}",
            "keystoreFile" : "&{openidm.keystore.location}",
            "keystorePassword" : "&{openidm.keystore.password}",
            "sessionOnly" : true
            "isHttpOnly" : true
            "maxTokenLifeMinutes" : "120",
            "tokenIdleTimeMinutes" : "30"
        }
},
```

Note

If you're working with the *OPENAM_SESSION* module, change the token lifetime properties as shown here, to match the session token lifetime associated with OpenAM.

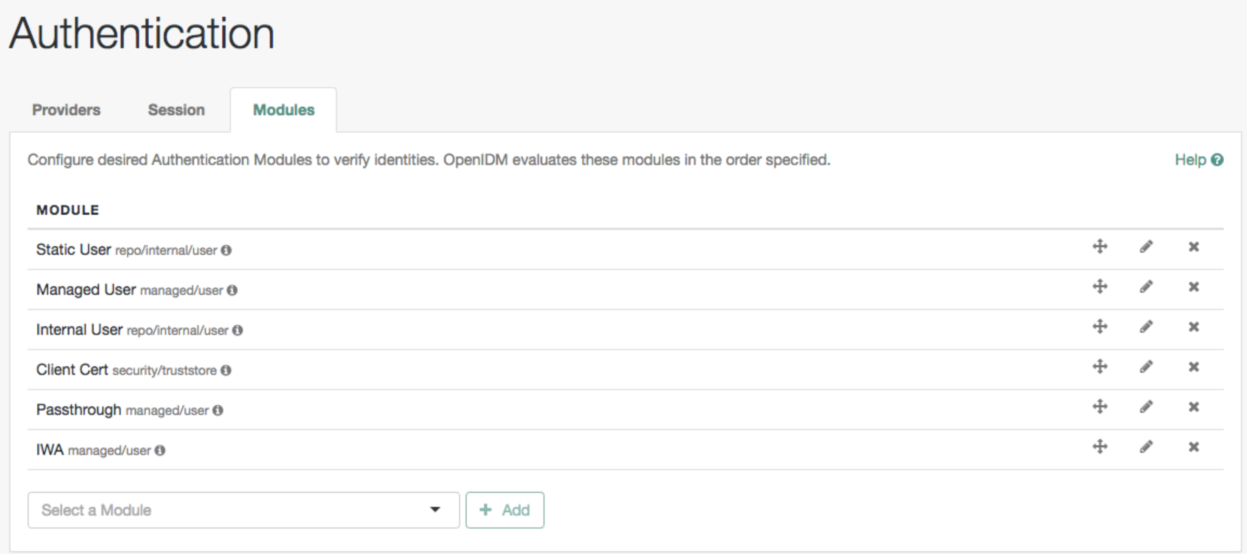
```
    "maxTokenLifeSeconds"  : "5",
    "tokenIdleTimeSeconds" : "5"
```

For more information about the `JWT_SESSION` module, see the following Javadoc page: *Class JwtSessionModule*.

18.1.2.2. Supported Authentication Modules

OpenIDM evaluates modules in the order shown in the `authentication.json` file for your project. When OpenIDM finds a module to authenticate a user, it does not evaluate subsequent modules.

You can also configure the order of authentication modules in the Admin UI. After logging in, choose Configure > Authentication, and select the Modules tab. The following figure illustrates how you might include the IWA module in the Admin UI.



Authentication

Configure desired Authentication Modules to verify identities. OpenIDM evaluates these modules in the order specified.

MODULE
Static User repo/internal/user
Managed User managed/user
Internal User repo/internal/user
Client Cert security/truststore
Passthrough managed/user
IWA managed/user

You must prioritize the authentication modules that query OpenIDM resources. Prioritizing the modules that query external resources might lead to authentication problems for internal users such as `openidm-admin`.

STATIC_USER

> `STATIC_USER` authentication provides an anonymous authentication mechanism that bypasses any database lookups if the headers in a request indicate that the user is `anonymous`. The following sample REST call uses `STATIC_USER` authentication in the self-registration process:

```
$ curl \
 --header "X-OpenIDM-Password: anonymous" \
 --header "X-OpenIDM-Username: anonymous" \
 --header "Content-Type: application/json" \
 --data '{
       "userName":"steve",
       "givenName":"Steve",
       "sn":"Carter",
       "telephoneNumber":"0828290289",
       "mail":"scarter@example.com",
       "password":"Passw0rd"
       }' \
 --request POST \
 "http://localhost:8080/openidm/managed/user/?_action=create"
```

Note that this is not the same as an anonymous request that is issued without headers.

Authenticating with the STATIC_USER module avoids the performance cost of reading the database for self-registration, certain UI requests, and other actions that can be performed anonymously. Authenticating the anonymous user with the STATIC_USER module is identical to authenticating the anonymous user with the INTERNAL_USER module, except that the database is not accessed. So, STATIC_USER authentication provides an authentication mechanism for the anonymous user that avoids the database lookups incurred when using INTERNAL_USER.

A sample STATIC_USER authentication configuration follows:

```
{
    "name" : "STATIC_USER",
    "enabled" : true,
    "properties" : {
        "propertyMapping" : "{}",
        "queryOnResource" : "repo/internal/user",
        "username" : "anonymous",
        "password" : "anonymous",
        "defaultUserRoles" : [
            "openidm-reg"
        ],
        "augmentSecurityContext" : null
    }
}
```

TRUSTED_ATTRIBUTE

The TRUSTED_ATTRIBUTE authentication module allows you to configure OpenIDM to trust the HttpServletRequest attribute of your choice. You can configure it by adding the TRUSTED_ATTRIBUTE module to your authentication.json file, as shown in the following code block:

```
...
{
    "name" : "TRUSTED_ATTRIBUTE",
    "properties" : {
        "queryOnResource" : "managed/user",
        "propertyMapping" : {
            "authenticationId" : "username",
            "userRoles" : "authzRoles"
        },
        "defaultUserRoles" : [ ],
        "authenticationIdAttribute" : "X-ForgeRock-AuthenticationId",
        "augmentSecurityContext" : {
            "type" : "text/javascript",
            "file" : "auth/populateRolesFromRelationship.js"
        }
    },
    "enabled" : true
}
...
```

`TRUSTED_ATTRIBUTE` authentication queries the `managed/user` repository, and allows authentication when credentials match, based on the `username` and `authzRoles` assigned to that user, specifically the `X-ForgeRock-AuthenticationId` attribute.

To see how you can configure this with OpenIDM, see "*The Trusted Servlet Filter Sample*" in the *Samples Guide*.

MANAGED_USER

`MANAGED_USER` authentication queries the repository, specifically the `managed/user` objects, and allows authentication if the credentials match. The default configuration uses the `username` and `password` of the managed user to authenticate, as shown in the following sample configuration:

```
{
    "name" : "MANAGED_USER",
    "enabled" : true,
    "properties" : {
        "augmentSecurityContext": {
            "type" : "text/javascript",
            "source" : "require('auth/customAuthz').setProtectedAttributes(security)"
        },
        "queryId" : "credential-query",
        "queryOnResource" : "managed/user",
        "propertyMapping" : {
            "authenticationId" : "username",
            "userCredential" : "password",
            "userRoles" : "roles"
        },
        "defaultUserRoles" : [ ]
    }
},
```

The `augmentSecurityContext` property can be used to add custom properties to the security context of users who authenticate with this module. By default, this property adds a list of *protected properties* to the user's security context. These protected properties are defined in the managed object schema. For more information, see the isProtected property described in "Creating and Modifying Managed Object Types".

INTERNAL_USER

INTERNAL_USER authentication queries the repository, specifically the repo/internal/user objects, and allows authentication if the credentials match. The default configuration uses the username and password of the internal user to authenticate, as shown in the following sample configuration:

```
{
    "name" : "INTERNAL_USER",
    "enabled" : true,
    "properties" : {
        "queryId" : "credential-internaluser-query",
        "queryOnResource" : "repo/internal/user",
        "propertyMapping" : {
            "authenticationId" : "username",
            "userCredential" : "password",
            "userRoles" : "roles"
        },
        "defaultUserRoles" : [ ]
    }
},
```

CLIENT_CERT

The client certificate module, CLIENT_CERT, provides authentication by validating a client certificate, transmitted via an HTTP request. OpenIDM compares the subject DN of the request certificate with the subject DN of the truststore.

A sample CLIENT_CERT authentication configuration follows:

```
{
    "name" : "CLIENT_CERT",
    "enabled" : true,
    "properties" : {
        "queryOnResource" : "security/truststore",
        "defaultUserRoles" : [ "openidm-cert" ],
        "allowedAuthenticationIdPatterns" : [ ]
    }
},
```

For more information about certificate-based authentication, see "Configuring Client Certificate Authentication".

The modules that follow point to external systems. In the authentication.json file, you should generally include these modules after any modules that query internal OpenIDM resources.

PASSTHROUGH

PASSTHROUGH authentication queries an external system, such as an LDAP server, and allows authentication if the provided credentials match those in the external system. The following sample configuration shows pass-through authentication using the user objects in the system endpoint `system/ldap/account`. For more information on pass-through authentication, see "Configuring Pass-Through Authentication".

OPENAM_SESSION

The `OPENAM_SESSION` module enables you to protect an OpenIDM deployment with ForgeRock's *OpenAM* (OpenAM) product.

In general, when integrating with OpenAM, you'll want to use the `OPENID_CONNECT` module. For an example configuration, see the following "*Integrating OpenIDM With the ForgeRock Identity Platform*" in the *Samples Guide*.

For detailed options, see "OPENAM_SESSION Module Configuration Options".

The use case is when you need to integrate IDM endpoints behind the scenes within other applications, such as with a company intranet portal. In that configuration, users would log into OpenAM to access the portal; at that point, their sessions would use the OpenAM SSO cookie, also known as `iPlanetDirectoryPro`. For more information, see *Session Cookies* in the AM *Authentication and Single Sign-On Guide*.

> **Note**
>
> If you use the `OPENAM_SESSION` token, you'll need to set a `JWT_SESSION` maximum token lifetime of *5 seconds*, to match the corresponding token session lifetime in OpenAM. For more information on the `JWT_SESSION` module, see "Supported Session Module".
>
> Ensure that at least one user in any shared OpenDJ repository has an `openidm-admin` role.
>
> Set up logins with OpenAM, to work with the related login session cookie, known as `iPlanetDirectoryPro`.

IWA

The `IWA` module enables users to authenticate by using Integrated Windows Authentication (IWA), rather than by providing a username and password. For information about configuring the IWA module with OpenIDM, see "Configuring IWA Authentication".

SOCIAL_PROVIDERS

The `SOCIAL_PROVIDERS` module supports configuration of social ID providers that comply with OAuth 2.0 and OpenID Connect 1.0 standards. For information about configuring this module with social ID providers such as Google, LinkedIn, and Facebook, see "Configuring the Social Providers Authentication Module".

OPENID_CONNECT

The OPENID_CONNECT module supports the use of OpenID Connect 1.0, which is an authentication layer built on OAuth 2.0. For information about configuring the

OPENID_CONNECT module with OpenIDM, see "Configuring Authentication With OpenID Connect".

If you're configuring a social ID provider compliant with OpenID Connect 1.0 or OAuth 2.0 standards, use the SOCIAL_PROVIDERS module described in "Configuring the Social Providers Authentication Module".

The OPENID_CONNECT module also supports integration with OpenAM. For an example of this integration, follow the procedure described in "*Integrating OpenIDM With the ForgeRock Identity Platform*" in the *Samples Guide*.

OAUTH

The OAUTH module works only with the OAuth 2.0 standards. For information about configuring OAUTH with OpenIDM, see "Configuring Authentication With OAuth 2.0".

18.1.3. Configuring Pass-Through Authentication

With pass-through authentication, the credentials included with the REST request are validated against those stored in a remote system, such as an LDAP server.

The following excerpt of an `authentication.json` shows a pass-through authentication configuration for an LDAP system.

```
"authModules" : [
    {
       "name" : "PASSTHROUGH",
       "enabled" : true,
       "properties" : {
          "augmentSecurityContext": {
             "type" : "text/javascript",
             "file" : "auth/populateAsManagedUser.js"
          },
          "queryOnResource" : "system/ldap/account",
          "propertyMapping" : {
             "authenticationId" : "uid",
             "groupMembership" : "memberOf"
          },
          "groupRoleMapping" : {
             "openidm-admin" : ["cn=admins"]
          },
          "managedUserLink" : "systemLdapAccounts_managedUser",
          "defaultUserRoles" : [
             "openidm-authorized"
          ]
       },
    },
    ...
 ]
```

For more information on authentication module properties, see the following: "*Authentication and Session Module Configuration Details*".

The OpenIDM samples, described in "*Overview of the Samples*" in the *Samples Guide*, include several examples of pass-through authentication configuration. Samples 2, 2b, 2c, and 2d use an external LDAP system for authentication. Sample 3 authenticates against a SQL database. Sample 6 authenticates against an Active Directory server. The `scriptedrest2dj` sample uses a scripted REST connector to authenticate against an OpenDJ server.

18.1.4. Configuring Authentication With OpenID Connect

The `OPENID_CONNECT` authentication module complies with OpenID Connect 1.0 standards.

If you want to enable a social ID provider that fully complies with OpenID Connect 1.0 standards, OpenIDM includes an authentication module wrapper known as `SOCIAL_PROVIDERS`. It is a specialized facility for sharing a social ID provider configuration with the authentication service, which you can configure as if it were a separate authentication module. For more information, see "Configuring the Social Providers Authentication Module".

The following excerpt of an `authentication.json` shows the default configuration associated with the `OPENID_CONNECT` authentication module.

```
{
    "enabled" : true,
    "properties" : {
        "resolvers" : [
            {
                "well-known" : "https://openam.example.com/openam/oauth2/.well-known/openid-
configuration",
                "name" : "OIDC",
                "client_id" : "client_id",
                "client_secret" : {
                    "$crypto" : {
                        "type" : "x-simple-encryption",
                        "value" : {
                            "cipher" : "AES/CBC/PKCS5Padding",
                            "salt" : "<someSaltValue>",
                            "data" : "<someEncryptedValue>",
                            "iv" : "<someEncryptedValue>",
                            "key" : "<someEncryptedValue>",
                            "mac" : "<someHashedValue>"
                        }
                    }
                },
                "scope" : [
                    "openid"
                ],
                "authorization_endpoint" : "https://openam.example.com/openam/oauth2/authorize",
                "token_endpoint" : "https://openam.example.com/openam/oauth2/access_token",
                "icon" : "<button class=\"btn btn-lg btn-default btn-block btn-social-provider\"><img src=
\"images/forgerock_logo.png\">Sign in</button>"
            }
        ],
        "queryOnResource" : "managed/user",
        "defaultUserRoles" : [
            "openidm-authorized"
        ],
        "queryId" : "",
        "openIdConnectHeader" : "authToken",
        "propertyMapping" : {
            "authenticationId" : "userName",
            "userCredential" : "",
            "userRoles" : "authzRoles"
        }
    },
    "name" : "OPENID_CONNECT"
}
```

If you include a `well-known` URL, OpenIDM uses information from that endpoint to pre-populate the `authorization_endpoint` and `token_endpoint`.

For more information on authentication module properties, see the following: "*Authentication and Session Module Configuration Details*".

18.1.5. Configuring Authentication With OAuth 2.0

The `OAUTH` authentication module complies with OAuth 2.0 standards.

If you want to enable a social ID provider that fully complies with OAuth 2.0 standards, OpenIDM includes an authentication module wrapper known as `SOCIAL_PROVIDERS`. It is a specialized facility for sharing a social ID provider configuration with the authentication service, which you can configure as if it were a separate authentication module. For more information, see "Configuring the Social Providers Authentication Module".

The following excerpt of an `authentication.json` shows the default configuration associated with the `OAUTH` authentication module.

```
{
    "enabled" : true,
    "properties" : {
        "resolvers" : [
            {
                "name" : "OAUTH",
                "client_id" : "client_id",
                "client_secret" : {
                    "$crypto" : {
                        "type" : "x-simple-encryption",
                        "value" : {
                            "cipher" : "AES/CBC/PKCS5Padding",
                            "salt" : "<someSaltedValue>",
                            "data" : "<someEncryptedValue>",
                            "iv" : "<someCipherValue>",
                            "key" : "openidm-sym-default",
                            "mac" : "<someCode>"
                        }
                    }
                },
                "scope" : [
                    "scope1",
                    "scope2"
                ],
                "authorization_endpoint" : "https://openam.example.com/openam/oauth2/authorize",
                "token_endpoint" : "https://openam.example.com/openam/oauth2/access_token",
                "userinfo_endpoint" : "https://openam.example.com/openam/oauth2/userinfo",
                "authenticationId" : "sub",
                "icon" : "<button class=\"btn btn-lg btn-default btn-block btn-social-provider\"><img src=
\"images/forgerock_logo.png\">Sign in</button>"
            }
        ],
        "queryOnResource" : "managed/user",
        "defaultUserRoles" : [
            "openidm-authorized"
        ],
        "queryId" : "",
        "authTokenHeader" : "authToken",
        "authResolverHeader" : "provider",
        "propertyMapping" : {
            "authenticationId" : "userName",
            "userCredential" : "",
            "userRoles" : "authzRoles"
        }
    },
    "name" : "OAUTH"
}
```

> **Note**
>
> For OAuth 2.0, the `userCredential` should be left blank.

For more information on authentication module properties, see the following: "*Authentication and Session Module Configuration Details*".

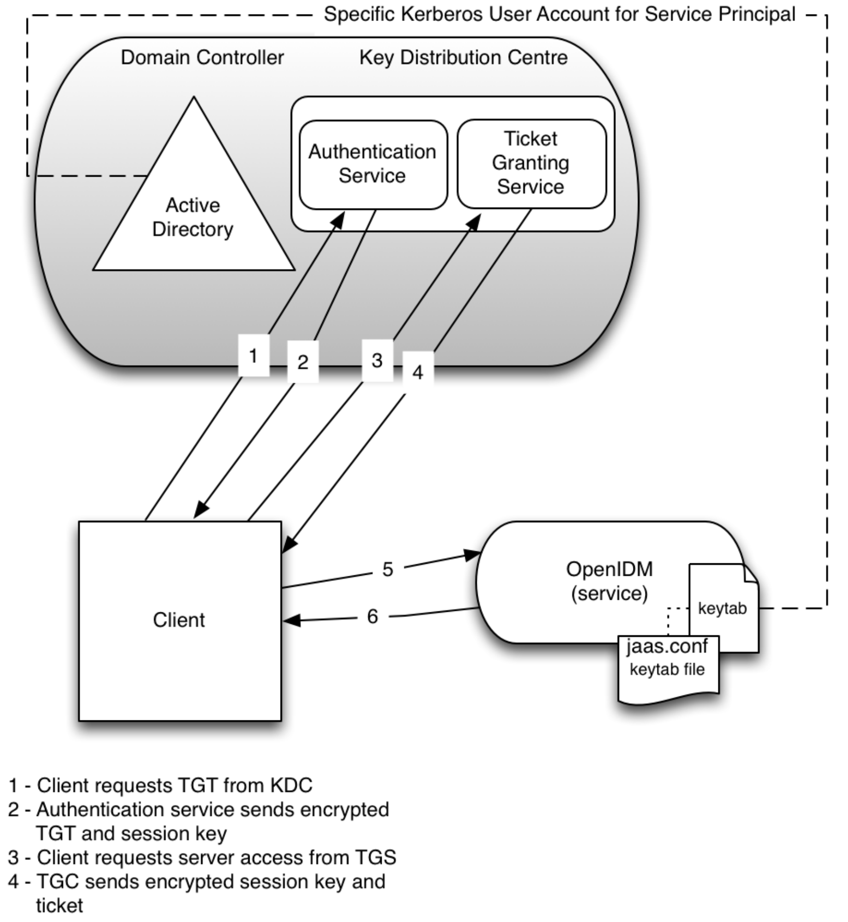
18.1.6. Configuring IWA Authentication

When OpenIDM is configured for IWA authentication, client browsers can authenticate to OpenIDM using a Kerberos ticket.

To enable Kerberos authentication, OpenIDM needs a specific Kerberos user account in Active Directory, and a keytab file that maps the service principal to this user account. When this is set up, the client presents OpenIDM with a Kerberos ticket. If OpenIDM can validate that ticket, the client is granted an encrypted session key for the OpenIDM service. That client can then access OpenIDM without providing a username or password, for the duration of the session.

The complete Kerberos authentication process is shown in the following diagram:

Client Authentication to OpenIDM Using a Kerberos Ticket


Specific Kerberos User Account for Service Principal
Domain Controller
Key Distribution Centre
Active Directory
Authentication Service
Ticket Granting Service
1
2
3
4
Client
5
6
OpenIDM (service)
keytab
jaas.conf keytab file
1 - Client requests TGT from KDC
2 - Authentication service sends encrypted TGT and session key
3 - Client requests server access from TGS
4 - TGC sends encrypted session key and ticket

This section assumes that you have an active Kerberos server acting as a Key Distribution Center (KDC). If you are running Active Directory in your deployment, that service includes a Kerberos KDC by default.

The steps required to set up IWA with OpenIDM are described in the following sections:

1. "Creating a Specific Kerberos User Account for OpenIDM"
2. "Creating a Keytab File"
3. "Configuring OpenIDM for IWA"

18.1.6.1. Creating a Specific Kerberos User Account for OpenIDM

To authenticate OpenIDM to the Kerberos KDC you must create a specific user entry in Active Directory whose credentials will be used for this authentication. This Kerberos user account must not be used for anything else.

The Kerberos user account is used to generate the Kerberos keytab. If you change the password of this Kerberos user after you have set up IWA authentication, you must update the keytab accordingly.

Create a new user in Active Directory as follows:

1. Select New > User and provide a login name for the user that reflects its purpose, for example, openidm@example.com.

2. Enter a password for the user. Check the *Password never expires* option and leave all other options unchecked.

   If the password of this user account expires, and is reset, you must update the keytab with the new password. It is therefore easier to create an account with a password that does not expire.

3. Click Finish to create the user.

18.1.6.2. Creating a Keytab File

A Kerberos keytab file (`krb5.keytab`) enables OpenIDM to validate the Kerberos tickets that it receives from client browsers. You must create a Kerberos keytab file for the host on which OpenIDM is running.

This section describes how to use the **ktpass** command, included in the Windows Server toolkit, to create the keytab file. Run the **ktpass** command on the Active Directory domain controller. Pay close attention to the use of capitalization in this example because the keytab file is case-sensitive. Note that you must disable UAC or run the **ktpass** command as a user with administration privileges.

The following command creates a keytab file (named `openidm.HTTP.keytab`) for the OpenIDM service located at `openidm.example.com`.

```
C:\Users\Administrator>ktpass ^
 -princ HTTP/openidm.example.com@EXAMPLE.COM ^
 -mapUser EXAMPLE\openidm ^
 -mapOp set ^
 -pass Passw0rd1 ^
 -crypto ALL
 -pType KRB5_NT_PRINCIPAL ^
 -kvno 0 ^
 -out openidm.HTTP.keytab

Targeting domain controller: host.example.com
Using legacy password setting method
Successfully mapped HTTP/openidm.example.com to openidm.
Key created.
Output keytab to openidm.HTTP.keytab:
Keytab version: 0x502
keysize 79 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
 vno 0 etype 0x1 (DES-CBC-CRC) keylength 8 (0x73a28fd307ad4f83)
keysize 79 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
 vno 0 etype 0x3 (DES-CBC-MD5) keylength 8 (0x73a28fd307ad4f83)
keysize 87 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
 vno 0 etype 0x17 (RC4-HMAC) keylength 16 (0xa87f3a337d73085c45f9416be5787d86)
keysize 103 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
 vno 0 etype 0x12 (AES256-SHA1) keylength 32 (0x6df9c282abe3be787553f23a3d1fcefc
  6fc4a29c3165a38bae36a8493e866d60)
keysize 87 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
 vno 0 etype 0x11 (AES128-SHA1) keylength 16 (0xf616977f071542cd8ef3ff4e2ebcc09c)
```

The **ktpass** command takes the following options:

- `-princ` specifies the service principal name in the format *`service/host-name@realm`*

  In this example (`HTTP/openidm.example.com@EXAMPLE.COM`), the client browser constructs an SPN based on the following:

  - The service name (HTTP).

    The service name for SPNEGO web authentication *must* be HTTP.

  - The FQDN of the host on which OpenIDM runs (`openidm.example.com`).

    This example assumes that users will access OpenIDM at the URL `https://openidm.example.com:8443`.

  - The Kerberos realm name (`EXAMPLE.COM`).

    The realm name must be in upper case. A Kerberos realm defines the area of authority of the Kerberos authentication server.

- `-mapUser` specifies the name of the Kerberos user account to which the principal should be mapped (the account that you created in "Creating a Specific Kerberos User Account for OpenIDM"). The username must be specified in down-level logon name format (DOMAIN\UserName). In our example, the Kerberos user name is `EXAMPLE\openidm`.

- `-mapOp` specifies how the Kerberos user account is linked. Use `set` to set the first user name to be linked. The default (`add`) adds the value of the specified local user name if a value already exists.

- `-pass` specifies a password for the principal user name. Use "*" to prompt for a password.

- `-crypto` Specifies the cryptographic type of the keys that are generated in the keytab file. Use `ALL` to specify all crypto types.

  This procedure assumes a 128-bit cryptosystem, with a default RC4-HMAC-NT cryptography algorithm. You can use the **ktpass** command to view the crypto algorithm, as follows:

  ```
  C:\Users\Administrator> ktpass -in .\openidm.HTTP.keytab
  Existing keytab:
  Keytab version: 0x502
  keysize 79 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
   vno 0 etype 0x1 (DES-CBC-CRC) keylength 8 (0x73a28fd307ad4f83)
  keysize 79 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
   vno 0 etype 0x3 (DES-CBC-MD5) keylength 8 (0x73a28fd307ad4f83)
  keysize 87 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
   vno 0 etype 0x17 (RC4-HMAC) keylength 16 (0xa87f3a337d73085c45f9416be5787d86)
  keysize 103 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
   vno 0 etype 0x12 (AES256-SHA1) keylength 32 (0x6df9c282abe3be787553f23a3d1fcefc6
   fc4a29c3165a38bae36a8493e866d60)
  keysize 87 HTTP/openidm.example.com@EXAMPLE.COM ptype 1 (KRB5_NT_PRINCIPAL)
   vno 0 etype 0x11 (AES128-SHA1) keylength 16 (0xf616977f071542cd8ef3ff4e2ebcc09c)
  ```

- `-ptype` Specifies the principal type. Use `KRB5_NT_PRINCIPAL`.

- `-kvno` specifies the key version number. Set the key version number to 0.

- `-out` specifies the name of the keytab file that will be generated, for example, `openidm.HTTP.keytab`.

  Note that the keys that are stored in the keytab file are similar to user passwords. You must therefore protect the Kerberos keytab file in the same way that you would protect a file containing passwords.

For more information about the **ktpass** command, see the ktpass reference in the Windows server documentation.

18.1.6.3. Configuring OpenIDM for IWA

To configure the IWA authentication module, you must do the following:

1. Add the `IWA` authentication module to your project's `conf/authentication.json` file.

2. Modify your project's `conf/system.properties` file to include a pointer to your login configuration for JAAS.

This section assumes that the connection from OpenIDM to the Active Directory Server is through an LDAP connector, and that the mapping from managed users to the users in Active Directory (in your project's `conf/sync.json` file) identifies the Active Directory target as `system/ad/account`. If you have named the target differently, modify the `"queryOnResource" : "system/ad/account"` property accordingly.

Add the IWA authentication module towards the end of your `conf/authentication.json` file. For example:

```
"authModules" : [
    ...
    {
        "name" : "IWA",
        "properties": {
            "servicePrincipal" : "HTTP/openidm.example.com@EXAMPLE.COM",
            "keytabFileName" : "openidm.HTTP.keytab",
            "kerberosRealm" : "EXAMPLE.COM",
            "kerberosServerName" : "kdc.example.com",
            "queryOnResource" : "system/ad/account",
            "maxTokenSize": 48000,
            "propertyMapping" : {
                "authenticationId" : "sAMAccountName",
                "groupMembership" : "memberOf"
            },
            "groupRoleMapping" : {
                "openidm-admin": [ ]
            },
            "groupComparisonMethod": "ldap",
            "defaultUserRoles" : [
                "openidm-authorized"
            ],
            "augmentSecurityContext" : {
                "type" : "text/javascript",
                "file" : "auth/populateAsManagedUser.js"
            }
        },
        "enabled" : true
    }
```

The IWA authentication module includes the following configurable properties:

`servicePrincipal`

The Kerberos principal for authentication, in the following format:

```
HTTP/host.domain@DC-DOMAIN-NAME
```

host and *domain* correspond to the host and domain names of the OpenIDM server. *DC-DOMAIN-NAME* is the domain name of the Windows Kerberos domain controller server. The *DC-DOMAIN-NAME* can differ from the domain name for the OpenIDM server.

`keytabFileName`

The full path to the keytab file for the Service Principal.

`kerberosRealm`

The Kerberos Key Distribution Center realm. For the Windows Kerberos service, this is the domain controller server domain name.

`kerberosServerName`

The fully qualified domain name of the Kerberos Key Distribution Center server, such as that of the domain controller server.

`queryOnResource`

The IDM resource to check for the authenticating user; for example, `system/ad/account`.

`maxTokenSize`

During the Kerberos authentication process, the Windows server builds a token to represent the user for authorization. This property sets the maximum size of the token, to prevent DoS attacks, if the SPENGO token in the request being made is amended with extra data. The default maximum token size is `48000` bytes.

`groupRoleMapping`

Enables you to grant different roles to users who are authenticated through the `IWA` module.

You can use the `IWA` module in conjunction with the `PASSTHROUGH` authentication module. In this case, a failure in the `IWA` module allows users to revert to forms-based authentication.

To add the `PASSTHROUGH` module, follow "Configuring Pass-Through Authentication".

When you have included the `IWA` module in your `conf/authentication.json` file, edit the `conf/system.properties` file to include a pointer to your login configuration file for JAAS. For example:

```
java.security.auth.login.config=&{launcher.project.location}/conf/gssapi_jaas.conf
```

Your `gssapi_jaas.conf` file must include the following information related to the LDAP connector:

```
org.identityconnectors.ldap.LdapConnector {
    com.sun.security.auth.module.Krb5LoginModule required
    client=TRUE
    principal="openidm.example.com@EXAMPLE.COM"
    useKeyTab=true
    keyTab="C:\\Users\\Administrator\\openidm\\security\\openidm.HTTP.keytab";
};
```

The value of the `principal` property must reflect the username. The value of the `keyTab` property must match what you have configured in your `authentication.json` file.

18.1.7. Configuring Client Certificate Authentication

The `CLIENT_CERT` module enables you to authenticate users by validating their certificates against the truststore.

The following procedure shows how to configure the client certificate authentication module:

With this configuration, users are authenticated if their public certificates are found in the truststore and have been issued by a CA whose root certificate is in the truststore. When users are authenticated, they receive the roles listed in the `defaultUserRoles` attribute. There is no further role retrieval and population.

To validate users against the truststore, follow these steps:

1. Import each potential user's public certificate into the OpenIDM truststore, as follows:

   ```
   $ keytool \
    -import \
    -alias username \
    -file certificate.pem \
    -keystore path/to/openidm/truststore \
    -storetype JKS
   ```

   In this example, *username* is the username with which the user logs in to OpenIDM and *certificate.pem* is that user's public certificate.

   Each time you add a new user, you must import that user's certificate into the truststore in the same way.

2. Configure the client certificate module, either in the Admin UI, or in your project's `conf/authentication.json` file as follows:

   ```
   {
       "name" : "CLIENT_CERT",
       "enabled" : true,
       "properties" : {
           "queryOnResource" : "security/truststore",
           "defaultUserRoles" : [
               "openidm-cert"
           ],
           "allowedAuthenticationIdPatterns" : [
               "(.*)"
           ]
       }
   }
   ```

   Verify the following properties:

   `enabled`

   Make sure that this property is set to `true`.

   `queryOnResource`

   Specifies the resource against which users are validated, in this case `security/truststore`.

   `defaultUserRoles`

   When a user authenticates with this module, the roles listed here are added to the user's security context. By default, the `openidm-cert` role is added.

`allowedAuthenticationIdPattern`

This property contains a regular expression (regex) that defines which user distinguished names (DNs) are allowed to authenticate with a certificate.

By default users can authenticate only if their certificates have been issued by a Certification Authority (CA) that is listed in the truststore. The default truststore includes several trusted root CA certificates and any user certificate issued by those CAs will be trusted. Change the value of this property to restrict certificates to those issued to users in your domain, or use some other regular expression to limit who will be trusted. The value in the example configuration (`"(.*)"`) means that all certificates will be trusted. If you leave this property empty, no certificates will be trusted.

> **Note**
>
> If you are using a private CA to issue user certificates, you should delete the default truststore file and create a new one that only contains your CA's public certificate and the public certificates of the users that you want to trust. For more information, see "Remove Unused CA Digital Certificates".

The following procedure enables you to test this authentication module with a generated self-signed certificate.

In this procedure, you will verify the certificate over port 8444 as defined in your project's `conf/boot/boot.properties` file:

```
openidm.auth.clientauthonlyports=8444
```

Demonstrating the `CLIENT_CERT` Module

1. Generate the self-signed certificate with the following command:

```
$ openssl \
 req \
 -x509 \
 -newkey rsa:1024 \
 -keyout key.pem \
 -out cert.pem \
 -days 3650 \
 -nodes
```

2. Respond to the questions when prompted.

```
Country Name (2 letter code) [XX]:
State or Province Name (full name) []:
Locality Name (eg, city) [Default City]:
Name (eg, company) [Default Company Ltd]:ForgeRock
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:localhost
Email Address []:
```

In this case, the `Name` corresponds to the `O` (for organization) of ForgeRock, and the `Common Name` corresponds to the `cn` of `localhost`. You'll use this information in a couple of steps.

3. Import the certificate `cert.pem` file into the OpenIDM truststore:

```
$ keytool \
 -importcert \
 -keystore \
 /path/to/openidm/security/truststore \
 -storetype JKS \
 -storepass changeit \
 -file cert.pem \
 -trustcacerts \
 -noprompt \
 -alias \
 client-cert-example
 Certificate was added to keystore
```

4. Open the `authentication.json` file in the *`project-dir`*`/conf` directory. Scroll to the code block with `CLIENT_CERT` and include the information from when you generated the self-signed certificate:

```
...
{
   "name" : "CLIENT_CERT",
   "properties" : {
      "queryOnResource" : "security/truststore",
      "defaultUserRoles" : [
         "openidm-cert"
      ],
      "allowedAuthenticationIdPatterns" : [
         "cn=localhost, O=ForgeRock"
      ]
   },
   "enabled" : true
}
...
```

5. Start OpenIDM:

```
$ cd /path/to/openidm
$ ./startup.sh -p project-dir
```

6. Send an HTTP request with your certificate file `cert.pem`:

```
$ curl \
--cert-type PEM
 \
--insecure
 \
--key key.pem
 \
--key-type PEM
 \
--tlsv1.2
 \
--cert /path/to/./cert.pem
 \
--header "X-OpenIDM-Username: jdoe"
 \
--header "X-OpenIDM-Password: Passw0rd"
 \
--request GET \
"https://localhost:8444/openidm/info/login"
      {
  "_id" : "login",
  "authenticationId" : "jdoe",
  "authorization" : {
    "component" : "managed/user",
    "authLogin" : false,
    "roles" : [ "openidm-authorized" ],
    "ipAddress" : "127.0.0.1",
    "protectedAttributeList" : [ "password" ],
    "id" : "d537507e-f05c-495a-b7fd-3bf9f19ffbed",
    "moduleId" : "MANAGED_USER"
  }
}
```

18.1.8. Interactions Between Modules in the Stack

OpenIDM supports integration with ForgeRock's Access Management product, also known as OpenAM. When you set up OpenIDM modules as described in *Integrating OpenIDM with the ForgeRock Identity Platform*, you'll set up interactions as described in OpenID Connect Authorization Code Flow in the *OpenID Connect 1.0 Guide*.

This integration between OpenIDM and OpenAM involves several different factors, described in the following sections:

18.1.8.1. Standards-Based Integration

When you integrate OpenIDM with OpenAM, you're setting up OpenIDM as a Relying Party, fully compliant with the OpenID Connect Discovery specification. The following list depicts each relevant standard associated with the OpenIDM implementation of the OPENID_CONNECT module, as described in "Configuring Authentication With OpenID Connect".

- *Relying Party*

As the Relying Party, OpenIDM registers OAuth 2.0 client profiles with OpenAM.

- *OpenID Provider*

  OpenAM acts as the OpenID Provider with configuration information.

- *Scope*

  Per the Scope Values section of the OpenID Connect specification, `openid` is a required scope for OpenID Connect requests.

- *Well-known endpoint*

  Per *Obtaining OpenID Provider Configuration Information*, "OpenID Providers supporting Discovery MUST make a JSON document available at the path formed by concatenating the string /.well-known/openid-configuration to the Issuer." ForgeRock complies with this by concatenating the noted string to the end of the related OAuth 2.0 URL; a typical well-known endpoint URL might be: `http://openam.example.com:8080/openam/oauth2/.well-known/openid-configuration`.

- *client_id*

  A client identifier valid at the authorization server; in this case, OpenIDM is the client and OpenAM is the authorization server.

- *client_secret*

  The client secret, in this case, is the password associated with the `client_id`. Clients authenticate with OpenAM (as an authorization server) by including the client credentials in the request body after receiving a `client_secret` value.

- *authorization_endpoint*

  As noted in *RFC 6749*, the authorization endpoint is used to interact with the resource owner and obtain an authorization grant. For the OpenAM implementation of the authorization endpoint, see OAuth 2.0 Client and Resource Server Endpoints in the *OAuth 2.0 Guide*.

- *token_endpoint*

  Also known as the Access Token Endpoint, this OpenAM endpoint receives an authorization code and returns an access token to OpenIDM. If you're using the OpenAM top-level realm, the endpoint will resemble `http://openam.example.com:8080/openam/oauth2/access_token`.

  If you're using an OpenAM sub-realm (as described in Setting Up Realms in the *Setup and Maintenance Guide*), include the name of the realm in the URL, such as: `http://openam.example.com:8080/openam/oauth2/access_token?realm=/someRealm`.

- *end_session_endpoint*

  The End Session Endpoint allows a relying party (OpenIDM) to request logging out an end-user at the OpenID Connect Party (OpenAM), at an endpoint such as: `http://openam.example.com:8080/openam`

`/oauth2/connect/endSession`. For more information, see the *OpenID Connect 1.0 Guide* *Reference* chapter.

18.1.8.2. Using the OpenIDM Session Module

When integrating OpenIDM and OpenAM, both servers have a session module. When you log into OpenIDM in the integrated configuration, the OpenIDM JWT_SESSION module is used solely as a client-side cache.

Without the OpenIDM JWT_SESSION module, the OPENID_CONNECT authentication module would have to call the OpenAM session module for *every* request. In contrast, with the OpenIDM JWT_SESSION module, it valdiates the OPENID_CONNECT token only after the JWT_SESSION module times out.

While fewer calls ot the OpenAM session module improves performance, that can lead to a problem; if a user logs out of OpenAM (or if that user's OpenAM session has timed out), that user's OpenIDM session may still be active. To minimize that issue, you can reduce the timeout associated with that user's OpenIDM JWT_SESSION, as shown in "Supported Session Module".

18.1.8.3. REST Calls and Integration

When you run a REST call on the integrated OpenIDM/OpenAM system, the application should have an OIDC token obtained from OpenAM. Caching is the responsibility of the REST application.

18.1.8.4. Mapping Admin Users

When you integrate OpenIDM with OpenAM, you're integrating their administrative accounts, and potentially more. For an example of how this is done, review the `amSessionCheck.js` file, as described in "Understanding the Integrated OpenAM Administrative User" in the *Samples Guide*.

18.2. Roles and Authentication

OpenIDM includes a number of default roles, and supports the configuration of managed roles, enabling you to customize the roles mechanism as needed.

The following roles are configured by default:

openidm-reg

Role assigned to users who access OpenIDM with the default anonymous account.

The `openidm-reg` role is excluded from the reauthorization required policy definition by default.

openidm-admin

OpenIDM administrator role, excluded from the reauthorization required policy definition by default.

openidm-authorized

Default role for any user who has authenticated with a user name and password.

openidm-cert

Default role for any user authenticated with mutual SSL authentication.

This role applies only for mutual authentication. Furthermore, the shared secret (certificate) must be adequately protected. The `openidm-cert` role is excluded from the reauthorization required policy definition by default.

openidm-tasks-manager

Role for users who can be assigned to workflow tasks.

When a user authenticates, OpenIDM calculates that user's roles as follows:

- If the authentication module with which the user authenticates includes a `defaultUserRoles` property, OpenIDM assigns those roles to the user on authentication. The `defaultUserRoles` property is specified as an array.
- The `userRoles` property is a mapping that specifies the attribute or list of attributes in the user entry that contains that specific user's authorization roles. For example, the following excerpt indicates that the `userRoles` should be taken from the user's `authzRoles` property on authentication:

  ```
  "userRoles" : "authzRoles"
  ```
- If the authentication module includes a `groupRoleMapping`, `groupMembership`, or `groupComparison` property, OpenIDM can assign additional roles to the user, depending on the user's group membership.

The roles calculated in sequence are cumulative.

For users who have authenticated with mutual SSL authentication, the module is `CLIENT_CERT` and the default role for such users is `openidm-cert`.

```
{   "name" : "CLIENT_CERT",
    "properties" : {
        "queryOnResource": "security/truststore",
        "defaultUserRoles": [  "openidm-cert" ],
        "allowedAuthenticationPatterns" : [ ]
    },
    "enabled" : "true"
}
```

Access control for such users is configured in the `access.js` file. For more information, see "Authorization".

18.3. Authorization

OpenIDM provides role-based authorization that restricts direct HTTP access to REST interface URLs. The default authorization configuration grants access rights to users, based on the following *internal* roles:

openidm-reg
openidm-authorized
openidm-admin
openidm-cert
openidm-tasks-manager

Note that this access control applies to direct HTTP calls only. Access for internal calls (for example, calls from scripts) is not affected by this mechanism.

Authorization roles are referenced in a user's `authzRoles` property, and are implemented using the relationships mechanism, described in "Managing Relationships Between Objects".

By default, all managed users have the `openidm-authorized` role. The following request shows the authorization roles for user `psmith`:

```
$ curl \
 --cacert self-signed.crt \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "https://localhost:8443/openidm/managed/user/psmith?_fields=authzRoles"
{
  "_id": "psmith",
  "_rev": "1",
  "authzRoles": [
    {
      "_ref": "repo/internal/role/openidm-authorized",
      "_refProperties": {
        "_id": "8e7b2c97-dfa8-4eec-a95b-b40b710d443d",
        "_rev": "1"
      }
    }
  ]
}
```

The authorization implementation is configured in two script files:

- `openidm/bin/defaults/script/router-authz.js`
- `*project-dir*/script/access.js`

OpenIDM calls the `router-authz.js` script for each request, through an `onRequest` hook that is defined in the `router.json` file. `router-authz.js` calls your project's access configuration script (`access.js`) to determine the allowed HTTP requests. If access is denied, according to the configuration defined in `access.js`, the `router-authz.js` script throws an exception, and OpenIDM denies the request.

18.3.1. Understanding the Router Authorization Script (`router-authz.js`)

This file provides the functions that enforce access rules. For example, the following function controls whether users with a certain role can start a specified process.

```
...
function isAllowedToStartProcess() {
var processDefinitionId = request.content._processDefinitionId;
return isProcessOnUsersList(processDefinitionId);
}
...
```

There are certain authorization-related functions in `router-authz.js` that should *not* be altered, as indicated in the comments in the file.

18.3.2. Understanding the Access Configuration Script (`access.js`)

This file defines the access configuration for HTTP requests and references the methods defined in `router-authz.js`. Each entry in the configuration contains a pattern to match against the incoming request ID, and the associated roles, methods, and actions that are allowed for requests on that pattern.

The options shown in the default version of the file do not include all of the actions available at each endpoint.

The following sample configuration entry indicates the configurable parameters and their purpose.

```
{
    "pattern"   : "*",
    "roles"     : "openidm-admin",
    "methods"   : "*", // default to all methods allowed
    "actions"   : "*", // default to all actions allowed
    "customAuthz" : "disallowQueryExpression()",
    "excludePatterns": "system/*"
},
```

As shown, this entry affects users with the `openidm-admin` role. Such users have HTTP access to all but `system` endpoints. The parameters are as follows:

`pattern`

The REST endpoint to which access is being controlled. `"*"` indicates access to all endpoints. `"managed/user/*"` would indicate access to all managed user objects.

`roles`

A list of the roles to which this access configuration applies.

The `roles` referenced here align with the details that are read from an object's security context (`security.authorization.roles`). The `authzRoles` relationship property of a managed user produces this security context value during authentication.

`methods`

A comma-separated list of the methods to which access is being granted. The method can be one or more of `create, read, update, delete, patch, action, query`. A value of `"*"` indicates that all methods are allowed. A value of `""` indicates that no methods are allowed.

`actions`

A comma-separated list of the allowed actions. The possible values depend on the service (URL) that is being exposed. The following list indicates the possible actions for each service.

`openidm/info/*` - `(no action parameter applies)`
`openidm/authentication` - `reauthenticate`
`openidm/config/ui/*` - `(no action parameter applies)`
`openidm/endpoint/getprocessforuser` - `create, complete`
`openidm/endpoint/gettasksview` - `create, complete`
`openidm/external/email` - `send`
`openidm/external/rest` - `(no action parameter applies)`
`openidm/managed` - `patch, triggerSyncCheck`
`openidm/managed/user` - `validateObject, validateProperty`
`openidm/policy` - `validateObject, validateProperty`
`openidm/recon` - `recon, reconById, cancel`
`openidm/repo` - `updateDbCredentials`
`openidm/script/*` - `eval`
`openidm/security/keystore` - `generateCert, generateCSR`
`openidm/security/truststore` - `generateCert, generateCSR`
`openidm/sync` - `notifyCreate, notifyUpdate, notifyDelete, recon, performAction`
`openidm/system` - `test, testConfig, availableConnectors, createCoreConfig, createFullConfig, liveSync, authenticate`
`openidm/system/<name>` - `script, test, liveSync`
`openidm/system/<name>/{id}` - `authenticate, liveSync`
`openidm/taskscanner` - `execute, cancel`
`openidm/workflow/processdefinition` - `create, complete`
`openidm/workflow/processinstance` - `create, complete`
`openidm/workflow/taskinstance` - `claim, create, complete`

A value of `"*"` indicates that all actions exposed for that service are allowed. A value of `""` indicates that no actions are allowed.

`customAuthz`

An optional parameter that enables you to specify a custom function for additional authorization checks. Custom functions are defined in `router-authz.js`.

`excludePatterns`

An optional parameter that enables you to specify particular endpoints to which access should not be granted.

18.3.3. Granting Internal Authorization Roles

Internal authorization roles can be granted to users through the Admin UI or over the REST interface, in much the same way as managed roles are granted. For more information about granting managed roles, see "Granting a Role to a User". To grant an internal role to a user through the Admin UI:

1. Select Manage > User and click the user to whom you want to grant the role.
2. Select the Authorization Roles tab and click Add Authorization Roles.
3. Select Internal Role as the Type, click to select from the list of defined Internal Roles, then click Add.

To grant an internal role over REST, add a reference to the internal role to the user's `authzRoles` property. The following command adds the `openidm-admin` role to user bjensen:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PATCH \
 --data '[
    {
       "operation": "add",
       "field": "/authzRoles/-",
       "value": {"_ref" : "repo/internal/role/openidm-admin"}
    }
 ]' \
 "http://localhost:8080/openidm/managed/user/bjensen"
{
  "_id": "bjensen",
  "_rev": "4",
  "mail": "bjensen@example.com",
  "givenName": "Barbara",
  "sn": "Jensen",
  "description": "Created By XML1",
  "userName": "bjensen@example.com",
  "telephoneNumber": "1234567",
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

Note

Because internal roles are not managed objects, you cannot manipulate them in the same way as managed roles. Therefore you cannot add a user to an internal role, as you would to a managed role.

18.3.4. Extending the Authorization Mechanism

You can extend the default authorization mechanism by defining additional functions in `router-authz.js` and by creating new access control configuration definitions in `access.js`.

18.3.5. Managing User Access to Workflows

The Self-Service UI is integrated with the embedded Activiti worfklow engine, enabling users to interact with workflows. Available workflows are displayed under the Processes item on the Dashboard. In order for a workflow to be displayed here, the workflow definition file must be present in the `openidm/workflow` directory.

A sample workflow integration with the Self-Service UI is provided in `openidm/samples/workflow`, and documented in "Sample Workflow - Provisioning User Accounts" in the *Samples Guide*. Follow the steps in that sample for an understanding of how the workflow integration works.

General access to workflow-related endpoints is based on the access rules defined in the `script/access.js` file. The configuration defined in the `conf/process-access.json` file determines who can invoke workflows. By default all users with the role `openidm-authorized` or `openidm-admin` can invoke any available workflow. The default `process-access.json` file is as follows:

```
{
    "workflowAccess" : [
        {
            "propertiesCheck" : {
                "property" : "_id",
                "matches" : ".*",
                "requiresRole" : "openidm-authorized"
            }
        },
        {
            "propertiesCheck" : {
                "property" : "_id",
                "matches" : ".*",
                "requiresRole" : "openidm-admin"
            }
        }
    ]
}
```

`"property"`

Specifies the property used to identify the process definition. By default, process definitions are identified by their `_id`.

`"matches"`

A regular expression match is performed on the process definitions, according to the specified property. The default (`"matches" : ".*"`) implies that all process definition IDs match.

`"requiresRole"`

Specifies the OpenIDM role that is required for users to have access to the matched process definition IDs. In the default file, users with the role `openidm-authorized` or `openidm-admin` have access.

To extend the process action definition file, identify the processes to which users should have access, and specify the qualifying user roles. For example, if you want to allow access to users with a role of `ldap`, add the following code block to the `process-access.json` file:

```
{
    "propertiesCheck" : {
        "property" : "_id",
        "matches" : ".*",
        "requiresRole" : "ldap"
    }
}
```

18.3.5.1. Adding Another Role to a Workflow

Sometimes, you'll want to configure multiple roles with access to the same workflow process. For example, if you want users with a role of doctor and nurse to both have access to certain workflows, you could set up the following code block within the `process-access.json` file:

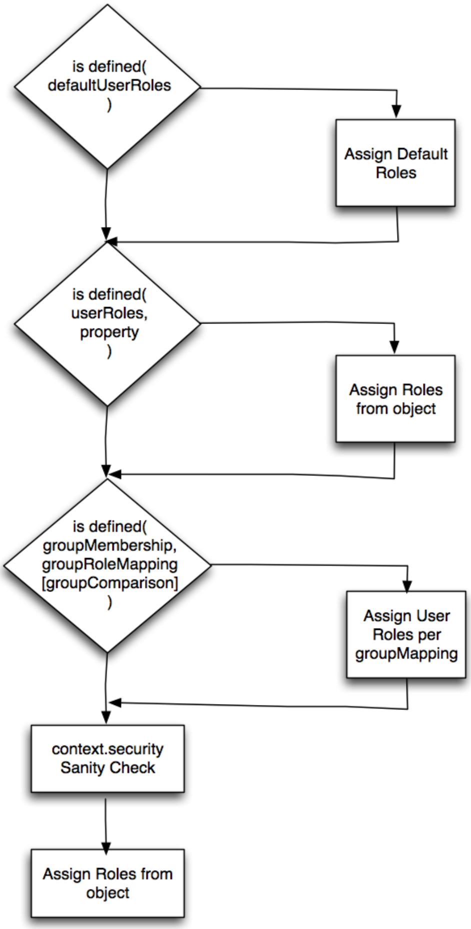
```
{
     "propertiesCheck" : {
          "property" : "_id",
          "matches" : ".*",
          "requiresRole" : "doctor"
     }
},
{
     "propertiesCheck" : {
          "property" : "_id",
          "matches" : ".*",
          "requiresRole" : "nurse"
     }
}
```

You could add more `requiresRole` code blocks, such as:

```
{
     "propertiesCheck" : {
          "property" : "_id",
          "matches" : ".*",
          "requiresRole" : "medic"
     }
}
```

18.4. Building Role-Based Access Control (RBAC)

Internal roles can be granted in a number of ways. The roles granted to specific users are cumulative, and are calculated based on the process depicted here:


is defined(defaultUserRoles)
Assign Default Roles
is defined(userRoles, property)
Assign Roles from object
is defined(groupMembership, groupRoleMapping [groupComparison])
Assign User Roles per groupMapping
context.security Sanity Check
Assign Roles from object

RBAC incorporates authentication and authorization options from roles configured for clients, for managed / internal users, as well as for group memberships.

The properties listed in this section are described in "Configuring Pass-Through Authentication".

Roles and authentication options can be configured for users in three stages:

Client Controlled

The `defaultUserRoles` may be added to authentication modules configured in the applicable `authentication.json` file. Default roles are listed in "Roles and Authentication".

If you see the following entry in `authentication.json`, the cited authentication property applies to all authenticated users:

```
"defaultUserRoles" : [ ]
```

Managed / Internal

Accumulated roles for users are collected in the `userRoles` property.

For a definition of managed and internal users, see "Authenticating OpenIDM Users".

Group roles

OpenIDM also uses group roles as input. Options include `groupMembership`, `groupRoleMapping`, and `groupComparison`

context.security

Once OpenIDM assigns roles and authentication modules to a user, OpenIDM then evaluates the result based on the `context.security` map, based on the scripts in the `policy.js` file. For more information, see "Roles, Authentication, and the Security Context".

18.4.1. Roles, Authentication, and the Security Context

The Security Context (`context.security`), consists of a principal (defined by the `authenticationId` property) and an access control element (defined by the `authorization` property).

If authentication is successful, the authentication framework sets the principal. OpenIDM stores that principal as the `authenticationId`. For more information, see the authentication components defined in "Supported Authentication Modules".

The `authorization` property includes an `id`, an array of `roles` (see "Roles and Authentication"), and a `component`, that specifies the resource against which authorization is validated. For more information, see "Configuring Pass-Through Authentication".

Chapter 19

Securing & Hardening Servers

OpenIDM provides a security management service that manages keystore and truststore files. The security service is accessible over the REST interface, enabling you to read and import SSL certificates, and to generate certificate signing requests.

This chapter describes the security management service and its REST interface.

In addition, the chapter outlines the specific security procedures that you should follow before deploying OpenIDM in a production environment.

> **Note**
>
> In a production environment, avoid the use of communication over insecure HTTP, self-signed certificates, and certificates associated with insecure ciphers.

19.1. Accessing the Security Management Service

By default, OpenIDM stores keystore and truststore files in the `/path/to/openidm/security` directory. These files can be managed by using the **keytool** command, or over the REST interface, at the URL `http://localhost:8080/openidm/security`. For information about using the **keytool** command, see http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html.

The keystore and truststore that OpenIDM uses are configured in the project's `conf/boot/boot.properties` file. The default configuration is as follows:

```
openidm.keystore.type=JCEKS
openidm.truststore.type=JKS
openidm.keystore.provider=SunJCE
openidm.truststore.provider=SUN
openidm.keystore.location=security/keystore.jceks
openidm.truststore.location=security/truststore
```

To use a different keystore and truststore, edit these properties. For an example, see how OpenIDM can be configured to use an HSM provider, in "Configuring OpenIDM to Support an HSM Provider".

You *must* specify a keystore for OpenIDM to be able to start up. If you do not specify a truststore, OpenIDM uses the keystore configuration as the truststore.

The following sections describe how to manage certificates and keys over REST and provides some comparable examples using the **keytool** command.

19.1.1. Displaying the Contents of the Keystore

OpenIDM generates a number of encryption keys the first time the server starts up. After startup, display the contents of the keystore over REST, as follows:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/security/keystore"
    {
  "_id": "keystore",
  "type": "JCEKS",
  "provider": {
    "Alg.Alias.Cipher.2.16.840.1.101.3.4.1.26": "AES_192/GCM/NoPadding",
    ...
  },
  "aliases": [
    "openidm-sym-default",
    "openidm-jwtsessionhmac-key",
    "openidm-localhost",
    "openidm-selfservice-key"
  ]
}
```

By default, OpenIDM includes the following aliases:

`openidm-sym-default`

The default symmetric key that is used, for example, to encrypt the configuration.

`openidm-jwtsessionhmac-key`

Used by the JWT session module to encrypt JWT session cookies.

`openidm-selfservice-key`

Used by the Self-Service UI to encrypt managed user passwords and other sensitive data.

`openidm-localhost`

The default alias that is used by the Jetty web server to service SSL requests. This alias references a private key and a self-signed certificate. You can use the self-signed certificate for testing purposes. When you deploy OpenIDM in a production environment, replace the self-signed certificate with a certificate that has been signed by a certificate authority.

19.1.2. Importing a Signed Certificate into the Keystore

If you have an existing CA-signed certificate, you can import it into OpenIDM's keystore by running a RESTful `PUT` command on the keystore alias. Include the signed certificate, private key, CA root certificate, and any intermediate certificates in the JSON payload.

The following command imports a CA-signed certificate, with the alias *example-com* into the keystore. Replace that alias with the alias of your certificate.

Important

This example includes line breaks in the certificate for legibility. If you are copying this command to test it, replace the line breaks with `/n` characters so that each certificate appears on a single line in the JSON payload, for example:

```
--data '{
    "alias": "example-com",
    "cert": [
         "-----BEGIN CERTIFICATE-----\nMIIGcDC...mfCH5cCrid0=\n-----END CERTIFICATE-----",
         "-----BEGIN CERTIFICATE----\nMIIGOTCA...XB0EU0bg==\n-----END CERTIFICATE-----"
    ],
     "privateKey": "-----BEGIN RSA PRIVATE KEY-----\nz5...tQ==\n-----END RSA PRIVATE KEY-----"
}'
```

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PUT \
 --data '{
    "alias": "example-com",
    "cert": "-----BEGIN CERTIFICATE-----\n
MIIGcDCCBVigAwIBAgIDC23tMA0GCSqGSIb3DQEBBQUAMIGMMQswCQYDVQQGEwJJ\n
TDEWMBQGA1UEChMNU3RhcnRDb20gTHRkLjErMCkGA1UECxMiU2VjdXJlIERpZ2l0\n
YWwgQ2VydGlmaWNhdGUgU2lnbmluZzE4MDYGA1UEAxMvU3RhcnRDb20gQ2xhc3Mg\n
MSBQcmltYXJ5IEludGVybWVkaWF0ZSBTZXJ2ZXIgQ0EwHhcNMTMwODA3MTMyODAz\n
WhcNMTQwODA4MDY0NTM5WjB2MRkwFwYDVQQNExBwZ3BDaGU4cEJPZnptVE9KMQsw\n
CQYDVQQGEwJHQjEjMCEGA1UEAxMadGVzdC1jb25uZWN0LmZvcmdlcm9jay5jb20x\n
JzAlBgkqhkiG9w0BCQEWGHBvc3RtYXN0ZXJAZm9yZ2Vyb2NrLmNvbTCCASIwDQYJ\n
KoZIhvcNAQEBBQADggEPADCCAQoCggEBAJRWGbnMGs+uGKU6ZrlTaaFdPczLqZnv\n
D37T0FOc/X3XXHxSVH94FDk7N4ansP2o6BsDWttIkM2AXkX3efMRaNpgxg7l4+DL\n
opV6H1RkrRba2Lom6Hp2pgkqvOBfd1ZMOmLbjUHt0jhypnIzu7TVwtTH7Ywsrx9F\n
uR9d4veYdW70IeQ64EhUG3RJBGG++AYJZCOjgEfbCwAYe/NoX/YVu+aMreHMR/+0\n
CV0YXKvHZgytcwZIc5WkQYaSWQA9lDWZzt5XjCErCATfiGEQ0k02QgpEfNTXxwQs\n
kfxh//O/qbfOWmloGwVU/2NY+5z3ZW8/eCksmiL1gGAYQAd+9+WI7BsCAwEAAaOC\n
Au4wggLqMAkGA1UdEwQCMAAwCwYDVR0PBAQDAgOoMBMGA1UdJQQMMAoGCCsGAQUF\n
BwMBMB0GA1UdDgQWBBR2zHzb71ZOHSwDZk28L9It3PvOtzAfBgNVHSMEGDAWgBTr\n
QjTQmLCrn/Qbawj3zGQu7w4sRTA0BgNVHREELTArghp0ZXN0LWNvbm5lY3QuZm9y\n
Z2Vyb2NrLmNvbYINZm9yZ2Vyb2NrLmNvbTCCAVYGA1UdIASCAU0wggFJMAgGBmeB\n
DAECATCCATsGCysGAQQBgbU3AQIDMIIBKjAuBggrBgEFBQcCARYiaHR0cDovL3d3\n
dy5zdGFydHNzbC5jb20vcG9saWN5LnBkZjCB9wYIKwYBBQUHAgIwgeowJxYgU3Rh\n
cnRDb20gQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkwAwIBARqBvlRoaXMgY2VydGlm\n
aWNhdGUgd2FzIGlzc3VlZCBhY2NvcmRpbmcgdG8gdGhlIENsYXNzIDEgVmFsaWRh\n
dGlvbiByZXF1aXJlbWVudHMgb2YgdGhlIFN0YXJ0Q29tIENBIHBvbGljeSwgcmVs\n
aWFuY2Ugb25seSBmb3IgdGhlIGludGVuZGVkIHB1cnBvc2UgaW4gY29tcGxpYW5j\n
ZSBvZiB0aGUgcmVseWluZyBwYXJ0eSBvYmxpZ2F0aW9ucy4wNQYDVR0fBC4wLDAq\n
oCigJoYkaHR0cDovL2NybC5zdGFydHNzbC5jb20vY3J0MS1jcmwuY3JsMIGOBggr\n
BgEFBQcBAQSBgTB/MDkGCCsGAQUFBzABhi1odHRwOi8vb2NzcC5zdGFydHNzbC5j\n
b20vc3ViL2NsYXNzMS9zZXJ2ZXIvY2EwQgYIKwYBBQUHMAKGNmh0dHA6Ly9haWEu\n
c3RhcnRzc2wuY29tL2NlcnRzL3N1Yi5jbGFzczEuc2VydmVyLmNhLmNydDAjBgNV\n
HRIEHDAahhhodHRwOi8vd3d3LnN0YXJ0c3NsLmNvbS8wDQYJKoZIhvcNAQEFBQAD\n
ggEBAKVOAHtXTrgISj7XvE4/lLxAfIP56nlhpoLu8CqVlLK6eK4zCQRyTiFYx3xq\n
VQMSNVgQIdimjEsMz8o5/fDrCrozsT6sqxIPFsdgdskPyz9YyC9Y/AVBuECxabQr\n
B//0STicfdPg8PuDYtI64/INA47d/gtb57RaTFYxKs6bU8vtObinDJCwT33x4tvt\n
ob18DwB3/PeTbWyVUIxB0nvfm89dys0SF2alaA/bLuy0B7rdlppd4dOMpmiD0tnI\n
DORtr5HOD1xGiixZWzA1V2pTmF/hJZbhmEgBUSIyPK5Z9pZPephMf+/KrovbQqKr\n
6SEjgs7dGwpo6fA2mfCH5cCrid0=\n
-----END CERTIFICATE-----",
    "privateKey": "-----BEGIN RSA PRIVATE KEY-----\n
zDot5q3vP9YjCihMZMkSa0zT2Zt+8S+mC0EVuYuTVhVpqrVNtkP1mlt+CYqmDffY\n
sGuD6SMrT6+SeAzX2uYFgY4+s8yaRWBcr0C5Z7yihilM6BK+IJ4is9kaW5VFr1Ph\n
wRKvSeFHBGh2wLNpjVSNPzLMDZBtkVi9Ny/xD5C3M1Gah0PGmnrPGCP8tr1Lshv4\n
PxYJwzHzouTdQDkLYlCjMN++NmIYfx7zrbEYV4VzXMxgNq7d3+d5dlVfE8xpAjSR\n
Lqlamib+doe1oWOQ2WiS6baBAH+Gw5rgqfwhJbCY/UlbCpuJ6kl7TLvTrFp8YpvB\n
Iv1GD0yuwSued3a+AxMFuIzTBYd2rC6rHq+eF4eHd/Q/Sbm9+9VuW/h8dW3LGvbE\n
5SUUhNw6uSkOZmZ0z/+FLbwoLPCASukY9biSd+12KJf4N42WZxID+9mJTp1j/Bv7\n
n29oGfZ3vav8PqG+F987hSyWEIdGTMfIxwaUrdYe1fmbUCxv0suMcYTRbAs9g3cm\n
eCNxbZBYC/fL+Nlj5NjZ+gxA/tEXV7wWynPZW3mZny6fQpDTDMslqsoFZR+rAUzH\n
ViePuLbCdxIC5heUyqvDBbeOzgQWOu6SZjX+mAQpo0DPKt1KDP4DKv9EW92sIwW3\n
AnFg98sje0DZ+zfsnevGioQMJrG0JSnqTYADxHaauu7NWndkfMZisfNIKA0u+ajU\n
AbP8xFXIP5JU8O4tWmlbxAbMOYfrZHabFNZx4DH1OVOJqdJIVx0KER0GSZd50D6W\n
QBzCfEbwMlJ17OB0AgWzNrbaak3MCmW1mh7OecjQwge1ajy7ho+JtQ==\n
-----END RSA PRIVATE KEY-----"
 }' \
 "http://localhost:8080/openidm/security/keystore/cert/example-com"

    {
  "_id": "example-com",
  "alias": "example-com",
  "cert": "-----BEGIN CERTIFICATE-----...-----END CERTIFICATE-----",
  "privateKey": "-----BEGIN RSA PRIVATE KEY-----...-----END RSA PRIVATE KEY-----"
}
```

If the import is successful, the command returns the certificate alias that has been added to the keystore, along with the certificates and keys.

19.1.3. Using an Alternative Certificate to Service SSL Requests

By default, OpenIDM uses the certificate with the alias `openidm-localhost` to service SSL requests. If you use a different certificate alias, you must import that certificate into your keystore *and* your truststore, and configure OpenIDM to use the new certificate.

1. Import the certificate into the keystore as described in "Importing a Signed Certificate into the Keystore".

2. Import the certificate into the trustore in a similar way. Substitute line breaks with `/n` characters so each certificate is on a single line:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request PUT \
 --data '{
    "alias": "example-com",
    "cert": "-----BEGIN CERTIFICATE-----\nMIIGc...rid0=\n-----END CERTIFICATE-----",
    "privateKey": "-----BEGIN RSA PRIVATE KEY-----\nzDt...tQ==\n-----END RSA PRIVATE KEY-----"
 }' \
 "http://localhost:8080/openidm/security/truststore/cert/example-com"

{
  "_id": "example-com",
  "alias": "example-com",
  "cert": "-----BEGIN CERTIFICATE-----\nMIIGc...CH5cCrid0=\n-----END CERTIFICATE-----",
  "privateKey": "-----BEGIN RSA PRIVATE KEY-----\nzDt...tQ==\n-----END RSA PRIVATE KEY-----"
}
```

3. Change the value of the `openidm.https.keystore.cert.alias` property in your project's `conf/boot/boot.properties` file to match the new alias, for example:

   ```
   openidm.https.keystore.cert.alias=example-com
   ```

4. Restart OpenIDM for the change to take effect.

19.1.4. Using Keytool to Import a Signed Certificate

This section shows how to import an existing CA-signed certificate into the OpenIDM keystore by using the **keytool** command, rather than the REST interface.

This procedure assumes that you have the following items, in .PEM format:

- A CA-signed certificate
- The private key associated with the Certificate Signing Request (CSR) that was used to request the signed certificate
- Optionally, any intermediary and root certificates from the Certificate Authority

  If there are multiple intermediary CA certificates, you can concatenate them with the root certificate into a single .PEM file.

1. Stop OpenIDM.
2. Back up your existing `openidm/security/keystore` and `openidm/security/truststore` files.
3. (Optional) Delete the default `openidm-localhost` certificate from the existing keystore and truststore files:

```
$ keytool \
 -delete \
 -alias openidm-localhost \
 -keystore security/keystore.jceks \
 -storetype jceks \
 -storepass changeit
```

```
$ keytool \
 -delete \
 -alias openidm-localhost \
 -keystore security/truststore \
 -storepass changeit
```

4. Generate a new PKCS12 keystore using the existing CA signed certificate, private key and CA certificate chain:

```
$ openssl pkcs12 \
 -export \
 -in cert.pem \
 -inkey key.pem \
 -certfile chain.pem \
 -name openidm-signed-cert \
 -out cert.pkcs12
Enter Export Password: changeit
Verifying - Enter Export Password: changeit
```

> **Important**
>
> When you generate the new PKCS12 keystore file, you are prompted to set an export password. This password *must* be the same as the existing OpenIDM keystore password. If you have not changed the OpenIDM keystore password, it is `changeit`. In a production environment, you *should* change this password.

5. Import the PKCS12 keystore that you generated in the previouse step into the OpenIDM keystore:

```
$ keytool \
 -importkeystore \
 -srckeystore cert.pkcs12 \
 -srcstoretype pkcs12 \
 -destkeystore /path/to/openidm/security/keystore.jceks \
 -storetype jceks
Enter destination keystore password: changeit
Enter source keystore password: changeit
Entry for alias openidm-signed-cert successfully imported.
Import command completed:  1 entries successfully imported, 0 entries failed or cancelled
```

6. Import the certificate into the OpenIDM truststore:

```
$ keytool \
 -import \
 -file cert.pem \
 -keystore /path/to/openidm/security/truststore \
 -alias openidm-signed-cert
Enter keystore password: changeit
Owner: EMAILADDRESS=admin@example.com, CN=example, OU=admin, O=www.example.com, ST=WA,
 C=US
...
Certificate fingerprints:
  MD5:  C2:06:DE:B0:AD:C7:28:14:1D:B6:BE:4A:CC:A1:CA:A0
  SHA1: F9:D7:6A:AE:47:99:61:0A:3C:90:4D:F0:73:DC:79:F4:30:B4:08:B1
  SHA256:
 6C:1A:0F:AF:16:89:8B:EE:1E:AE:A9:19:56:29:D8:6D:C1:4D:82:58:C0:43:66:08:C4:C9:16:1D:BA:C5:D6:5D
  Signature algorithm name: SHA1withRSA
  Version: 3
...
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

7. Edit your project's `conf/boot/boot.properties` file and set the `openidm.https.keystore.cert.alias` to the alias of the signed certificate, for example:

   ```
   openidm.https.keystore.cert.alias=openidm-signed-cert
   ```

8. Restart OpenIDM for the new certificate to be taken into account.

19.1.5. Generating a Certificate Signing Request Over REST

If you do not have an existing signed certificate, you can generate a certificate signing request (CSR) over REST, as described in this section. The details of the CSR are specified in JSON format, for example:

```
{
    "CN" : "www.example.com",
    "OU" : "HR",
    "L"  : "Cupertino",
    "C"  : "US"
}
```

For information about the complete contents of a CSR, see http://www.sslshopper.com/what-is-a-csr-certificate-signing-request.html.

To generate a CSR over REST, include the private key alias in the JSON payload. The following example uses the alias `example-com`.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{"CN" : "www.example.com",
 "OU" : "HR",
 "L"  : "Cupertino",
 "C"  : "US",
 "alias" : "example-com"}' \
 "http://localhost:8080/openidm/security/keystore?_action=generateCSR"
{
  "_id": "example-com",
  "csr": "-----BEGIN CERTIFICATE REQUEST-----\nMIIC...H6i14==\n-----END CERTIFICATE REQUEST-----\n",
  "publicKey": {
    "algorithm": "RSA",
    "format": "X.509",
    "encoded": "-----BEGIN PUBLIC KEY-----\nMIIBIjA...MQIDAQAB\n-----END PUBLIC KEY-----\n"
  }
}
```

This example request returns the CSR and the public key.

When the signed certificate is returned by the certificate authority and you import the certificate into the keystore, you must include the private key that you used to generate the request with the certificate chain.

Send the output from

```
"csr": "-----BEGIN CERTIFICATE REQUEST-----
     ...
     -----END CERTIFICATE REQUEST-----
```

to your certificate authority for signature.

When the signed certificate is returned, import it into the keystore, as described in "Importing a Signed Certificate into the Keystore".

19.1.6. Generating a Self-Signed Certificate Over REST

To generate a self-signed X.509 certificate, use the `generateCert` action on the `keystore` endpoint. This action must be performed as an authenticated administrative user. The generated certificate is returned in the response to the request, and stored in the OpenIDM keystore.

Specify the details of the certificate in the JSON payload. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '{
   "algorithm" : "RSA",
   "signatureAlgorithm" : "SHA512WithRSAEncryption",
   "keySize"  : 2048,
   "domainName"  : "www.example.com",
   "validFrom" : "2016-08-13T07:59:44.497+02:00",
   "validTo" : "2017-08-13T07:59:44.497+02:00",
   "alias" : "new-alias"
 }' \
 "http://localhost:8080/openidm/security/keystore?_action=generateCert"
{
  "_id": "new-alias",
  "type": "X.509",
  "cert": "-----BEGIN CERTIFICATE-----\nMIIDSDjCgAwIBAg...Ib==\n-----END CERTIFICATE-----\n",
  "publicKey": {
    "algorithm": "RSA",
    "format": "X.509",
    "encoded": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgk...AB\n-----END PUBLIC KEY-----\n"
  },
  "issuer": {
    "ST": "None",
    "C": "None",
    "OU": "None",
    "CN": "www.example.com",
    "L": "None",
    "O": "None"
  },
  "notBefore": 1471067984000,
  "notAfter": 1502603984000
}
```

The following certificate details can be specified:

`algorithm` (optiona)

The public key algorithm, for example, `RSA`. If no algorithm is specified, a default of `RSA` is used.

`signatureAlgorithm` (optional)

The signature type, for example, `SHA512WithRSAEncryption`. If no algorithm is specified, a default of `SHA512WithRSAEncryption` is used.

`keySize` (optional)

The size of the key (in bits) used in the cryptographic algorithm, for example `2048`. If no key size is specified, a default of `2048` is used.

`domainName`

The fully qualified domain name (FQDN) of your server, for example `www.example.com`.

`validFrom` and `validTo` (optional)

The validity period of the certificate, in UTC time format, for example `2016-08-13T07:59:44.497+02:00`. If no values are specified, the certificate is valid for one year, from the current date.

`alias`

The keystore alias or string that identifies the certificate, for example `openidm-localhost`.

19.1.7. Deleting Certificates Over REST

If you use an alternative certificate, as described in "Using an Alternative Certificate to Service SSL Requests", you might want to delete the default certificate from the keystore and the truststore. You can delete certificates by sending a REST delete request to the keystore or truststore endpoint.

The following example deletes the `openidm-localhost` certificate from the keystore:

```
$ curl \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "If-Match: *" \
 --request DELETE \
 "http://localhost:8080/openidm/security/keystore/cert/openidm-localhost"
{
    "_id":"openidm-localhost"
}
```

The following example deletes the `openidm-localhost` certificate from the truststore:

```
$ curl \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "If-Match: *" \
 --request DELETE \
 "http://localhost:8080/openidm/security/truststore/cert/openidm-localhost"
{
    "_id":"openidm-localhost"
}
```

You can use similar commands to delete custom certificates from the keystore and truststore, specifying the certificate alias in the request.

19.1.8. Rotating Encryption Keys

You can change the key that is used to encrypt managed data in an existing OpenIDM deployment. By default, OpenIDM encrypts managed object properties with the default symmetric key (`openidm-sym-default`). You can specify that a different key or cipher be used in the managed object schema (in your project's `conf/managed.json` file). You can also specify that different keys be used to encrypt different properties. For example, the following excerpt of a `managed.json` file specifies that user passwords must be encrypted with a key with alias `my-example-key`:

```
"password" : {
    "title" : "Password",
    "type" : "string",
    ...
    "encryption" : {
        "key" : "my-example-key"
    },
```

When you change the encryption key in the managed object schema, the affected properties are re-encrypted with the new key the next time the managed object is *updated*.

Important

If you add a new key to the keystore while OpenIDM is running, you *must* restart the server before you refer to that key in the configuration. OpenIDM does not reload the keystore during runtime.

One caveat with key rotation is that both the old key and the new key must remain in the keystore until every object that is encrypted with the old key has been updated to use the new key. You can force the key rotation on all managed objects by running the `triggerSyncCheck` action on the entire managed object data set. The `triggerSyncCheck` action examines the crypto blob of each object and updates the encrypted property with the correct key. For example, the following command forces all managed user objects to use the new key:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 "http://localhost:8080/openidm/managed/user/?_action=triggerSyncCheck"

{
    "status":"OK",
    "countTriggered":10
}
```

Note that the `triggerSyncCheck` action does not propagate the key rotation over all the nodes in an OpenIDM cluster. You must run the action on each successive node.

In a large managed object set, the `triggerSyncCheck` action can take a very long time to run on only a single node. You should therefore avoid using this action if your data set is large. An alternative to running `triggerSyncCheck` over the entire data set is to iterate over the managed data set and call `triggerSyncCheck` on each individual managed object. You can call this action manually or by using a script.

The following example shows the manual commands that must be run to launch the `triggerSyncCheck` action on all managed users. The first command uses a query filter to return all managed user IDs. The second command iterates over the returned IDs calling `triggerSyncCheck` on each ID:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 "http://localhost:8080/openidm/managed/user?_queryFilter=true&_fields=_id"

{
  "result": [
    {
      "_id": "bjensen",
      "_rev": "2"
    },
    {
      "_id": "scarter",
      "_rev": "2"
    },
    {
      "_id": "jdoe",
      "_rev": "2"
    }
,
...
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
"http://localhost:8080/openidm/managed/user/bjensen?_action=triggerSyncCheck"
{
  "userName": "bjensen",
  "givenName": "Babs",
  "sn": "Jensen",
  "telephoneNumber": "12345678",
  "active": "true",
  "mail": "bjensen@example.com",
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": [],
  "_id": "bjensen",
  "_rev": "2"
}
```

In large data sets, the most efficient way to achieve key rotation is to use the scheduler service to launch these commands. The following example uses the scheduler service for this purpose.

Using Scheduled Tasks to Rotate Keys

This example uses a script to generate multiple scheduled tasks. Each scheduled task iterates over a subset of the managed object set (defined by the `pageSize`). The generated scheduled task then calls another script that launches the `triggerSyncCheck` action on each managed object in that subset.

You can set up a similar schedule as follows:

1. Create a schedule configuration named `schedule-triggerSyncCheck.json` in your project's `conf` directory. That schedule should look as follows:

```
{
    "enabled" : true,
    "persisted" : true,
    "type" : "cron",
    "schedule" : "0 * * * * ? *",
    "concurrentExecution" : false,
    "invokeService" : "script",
    "invokeContext" : {
        "waitForCompletion" : false,
        "script": {
            "type": "text/javascript",
            "name": "sync/scheduleTriggerSyncCheck.js"
        },
        "input": {
            "pageSize": 2,
            "managedObjectPath" : "managed/user",
            "quartzSchedule" : "0 * * * * ? *"
        }
    }
}
```

You can change the following parameters of this schedule configuration to suit your deployment:

`pageSize`

The number of objects that each generated schedule will handle. This value should be high enough not to create too many schedules. The number of schedules that is generated is equal to the number of objects in the managed object store, divided by the page size.

For example, if there are 500 managed users and a page size of 100, five schedules will be generated (500/100).

`managedObjectPath`

The managed object set over which the scheduler iterates. For example, `managed/user` if you want to iterate over the managed user object set.

`quartzSchedule`

The schedule at which these tasks should run. For example, to run the task every minute, this value would be `` `0 * * * * ? *` ``.

2. The schedule calls a `scheduleTriggerSyncCheck.js` script, located in a directory named *`project-dir/`*`script/sync`. Create the `sync` directory, and add that script as follows:

```
var managedObjectPath = object.managedObjectPath;
var pageSize = object.pageSize;
var quartzSchedule = object.quartzSchedule;

var managedObjects = openidm.query(managedObjectPath, {
    "_queryFilter": "true",
    "_fields" : "_id"
});
```

```
var numberOfManagedObjects = managedObjects.result.length;

for (var i = 0; i < numberOfManagedObjects; i += pageSize) {
    var scheduleId = java.util.UUID.randomUUID().toString();
 var ids = managedObjects.result.slice(i, i + pageSize).map(function (obj) { return obj._id});
    var schedule = newSchedule(scheduleId, ids);
    openidm.create("/scheduler", scheduleId, schedule);
}

function newSchedule (scheduleId, ids) {
 var schedule = {
    "enabled" : true,
    "persisted" : true,
    "type" : "cron",
    "schedule" : quartzSchedule,
    "concurrentExecution" : false,
    "invokeService" : "script",
    "invokeContext" : {
        "waitForCompletion" : true,
        "script": {
            "type": "text/javascript",
            "name": "sync/triggerSyncCheck.js"
        },
        "input": {
            "ids" : ids,
            "managedObjectPath" : managedObjectPath,
            "scheduleId" : scheduleId
        }
    }
 };
 return schedule;
}
```

3. Each generated scheduled task calls a script named `triggerSyncCheck.js`. Create that script in your project's `script/sync` directory. The contents of the script are as follows:

   ```
   var ids = object.ids;
   var scheduleId = object.scheduleId;
   var managedObjectPath = object.managedObjectPath;

   for (var i = 0; i < ids.length; i++) {
       openidm.action(managedObjectPath + "/" + ids[i], "triggerSyncCheck", {}, {});
   }

   openidm.delete("scheduler/" + scheduleId, null);
   ```

4. When you have set up the schedule configuration and the two scripts, you can test this key rotation as follows:

   a. Edit your project's `conf/managed.json` file to return user passwords by default by setting `"scope" : "public"`.

```
"password" : {
    ...
    "encryption" : {
        "key" : "openidm-sym-default"
    },
    "scope" : "public",
....
```

Because passwords are not returned by default, you will not be able to see the new encryption on the password unless you change the property's `scope`.

b. Perform a GET request to return any managed user entry in your data set. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
"http://localhost:8080/openidm/managed/user/bjensen"
{
  "_id": "bjensen",
  "_rev": "3",
  "userName": "bjensen",
  "givenName": "Babs",
  "sn": "Jensen",
  "telephoneNumber": "12345678",
  "active": "true",
  "mail": "bjensen@example.com",
  "password": {
    "$crypto": {
      "type": "x-simple-encryption",
      "value": {
        "cipher": "AES/CBC/PKCS5Padding",
        "data": "YgTtyHGCPSrPBoJgWq1CYg==",
        "iv": "24Sgdl1+YL/2Qw0UQUjV8A==",
        "key": "openidm-sym-default"
      }
    }
  },
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

Notice that the user's password is encrypted with the default encryption key (`openidm-sym-default`).

c. Create a new encryption key in the OpenIDM keystore:

```
$ keytool \
 -genseckey \
 -alias my-new-key \
 -keyalg AES \
 -keysize 128 \
 -keystore /path/to/openidm/security/keystore.jceks \
 -storetype JCEKS
```

d. Shut down OpenIDM for keystore to be reloaded.

e. Change your project's `conf/managed.json` file to change the encryption key for managed user passwords:

```
"password" : {
    ...
    "encryption" : {
        "key" : "my-new-key"
    },
    "scope" : "public",
....
```

f. Restart OpenIDM and wait one minute for the first scheduled task to fire.

g. Perform a GET request again to return the entry of the managed user that you returned previously:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
"http://localhost:8080/openidm/managed/user/bjensen"
{
  "_id": "bjensen",
  "_rev": "4",
  "userName": "bjensen",
  "givenName": "Babs",
  "sn": "Jensen",
  "telephoneNumber": "12345678",
  "active": "true",
  "mail": "bjensen@example.com",
  "password": {
    "$crypto": {
      "type": "x-simple-encryption",
      "value": {
        "cipher": "AES/CBC/PKCS5Padding",
        "data": "qWD+lBC8iqXsPbpYkSaVdg==",
        "iv": "4HTKhHODo8x82tIky/PMIw==",
        "key": "my-new-key"
      }
    }
  },
  "accountStatus": "active",
  "effectiveRoles": [],
  "effectiveAssignments": []
}
```

Notice that the user password is now encrypted with `my-new-key`.

19.2. Security Precautions for a Production Environment

Out of the box, OpenIDM is set up for ease of development and deployment. When you deploy OpenIDM in production, there are specific precautions you should take to minimize security breaches. After following the guidance in this section, make sure that you test your installation to verify that it behaves as expected before putting it into production.

19.2.1. Use SSL and HTTPS

Disable plain HTTP access, as described in "Secure Jetty".

Use TLS/SSL to access OpenIDM, ideally with mutual authentication so that only trusted systems can invoke each other. TLS/SSL protects data on the network. Mutual authentication with strong certificates, imported into the trust and keystores of each application, provides a level of confidence for trusting application access.

Augment this protection with message level security where appropriate.

19.2.2. Restrict REST Access to the HTTPS Port

When possible, use a certificate to secure REST access, over HTTPS. For production, that certificate should be signed by a certificate authority.

OpenIDM generates a self-signed certificate when it first starts up. You can use this certificate to test secure REST access. To do so, create a self-signed certificate file, `self-signed.crt`, using the following procedure:

1. Extract the certificate that is generated when OpenIDM starts up.

   ```
   $ openssl s_client -showcerts -connect localhost:8443 </dev/null
   ```

   This command outputs the entire certificate to the terminal.

2. Using any text editor, create a file named `self-signed.crt`. Copy the portion of the certificate from `-----BEGIN CERTIFICATE-----` to `----END CERTIFICATE-----` and paste it into the `self-signed.crt` file, which should appear similar to the following:

```
$ more self-signed.crt
-----BEGIN CERTIFICATE-----
MIIB8zCCAVygAwIBAgIETkvDjjANBgkqhkiG9w0BAQUFADA+MSgwJgYDVQQKEx9P
cGVuSURNIFNlbGYtU2lnbmVkIENlcnRpZmljYXRlMRIwEAYDVQQDEwlsb2NhbGhv
c3QwHhcNMTEwODE3MTMzNTEwWhcNMjEwODE3MTMzNTEwWjA+MSgwJgYDVQQKEx9P
cGVuSURNIFNlbGYtU2lnbmVkIENlcnRpZmljYXRlMRIwEAYDVQQDEwlsb2NhbGhv
c3QwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAKwMkyvHS5yHAnI7+tXUIbfI
nQfhcTChpWNPTHc/cli/+Ta1InTpN8vRScPoBG0BjCaIKnVVl2zZ5ya74UKgwAVe
oJQ0xDZvIyeC9PlvGoqsdtH/Ihi+T+zzZ14oVxn74qWoxZcvkG6rWEOd42QzpVhg
wMBzX98slxkOZhG9IdRxAgMBAAEwDQYJKoZIhvcNAQEFBQADgYEASo4qMI0axEKZ
m0jU4yJejLBHydWoZVZ8fKcHVlD/rTirtVgWsVgvdr3yUr0Idk1rH1nEF47Tzn+V
UCq7qJZ75HnIIeVrZqmfTx8169paAKAaNF/KRhTE6ZII8+awst02L86shSSWqWz3
s5xPB2YTaZHWWdzrPVv90gL8JL/N7/
Q=
-----END CERTIFICATE-----
```

3. Test REST access on the HTTPS port, referencing the self-signed certificate in the command. For example:

```
$ curl \
 --header "X-OpenIDM-Username:openidm-admin" \
 --header "X-OpenIDM-Password:openidm-admin" \
 --cacert self-signed.crt \
 --request GET \
 "https://localhost:8443/openidm/managed/user/?_queryId=query-all-ids"
     {
    "result": [],
    "resultCount": 0,
    "pagedResultsCooke": null,
    "remainingPagedResuts": -1
}
```

19.2.3. Restrict the HTTP Payload Size

Restricting the size of HTTP payloads can protect the server against large payload HTTP DDoS attacks. IDM includes a servlet filter that limits the size of an incoming HTTP request payload, and returns a `413 Request Entity Too Large` response when the maximum payload size is exceeded.

By default, the maximum payload size is 5MB. You can configure the maximum size in your project's `conf/servletfilter-payload.json` file. That file has the following structure by default:

```
{
     "classPathURLs" : [ ],
     "systemProperties" : { },
     "requestAttributes" : { },
     "scriptExtensions" : { },
     "initParams" : {
         "maxRequestSizeInMegabytes" : "5"
     },
     "urlPatterns" : [
         "/*"
     ],
     "filterClass" : "org.forgerock.openidm.jetty.LargePayloadServletFilter"
 }
```

Change the value of the maxRequestSizeInMegabytes property to set a different maximum HTTP payload size. The remaining properties in this file are described in "Registering Additional Servlet Filters".

19.2.4. Encrypt Data Internally and Externally

Beyond relying on end-to-end availability of TLS/SSL to protect data, OpenIDM also supports explicit encryption of data that goes on the network. This can be important if the TLS/SSL termination happens prior to the final endpoint.

OpenIDM also supports encryption of data stored in the repository, using a symmetric key. This protects against some attacks on the data store. Explicit table mapping is supported for encrypted string values.

OpenIDM automatically encrypts sensitive data in configuration files, such as passwords. OpenIDM replaces clear text values when the system first reads the configuration file. Take care with configuration files having clear text values that OpenIDM has not yet read and updated.

19.2.5. Remove Unused CA Digital Certificates

The Java keystore and OpenIDM truststore files include several dozen root CA certificates. While the probability of a compromised root CA is low, best practices in security suggest that you should delete root CA certificates that are not used in your deployment.

To review the current list of root CA certificates in the OpenIDM truststore, run the following command:

```
$ keytool \
-storepass \
changeit \
-list
 \
-keystore
 \
/path/to/openidm/security/truststore
```

On UNIX/Linux systems, you can find additional lists of root CA certificates in files named cacerts. They include root CA certificates associated with your Java environment, such as Oracle JDK or OpenJDK. You should be able to find that file in the following location: ${JAVA_HOME}/jre/lib/security/cacerts.

Before doing anything with your Java environment keystore files, make sure the Java-related cacerts files are up to date. Install the latest supported version, as shown in "*Before You Install*" in the *Release Notes*.

You can remove root CA certificates with the **keytool** command. For example, the following command removes the hypothetical examplecomca2 certificate from the OpenIDM truststore file:

```
$ keytool \
-storepass \
changeit \
-delete
 \
-keystore
 \
/path/to/openidm/security/truststore
 \
-alias \
examplecomca2
```

Repeat the process for all root CA certificates that are not used in your deployment.

On Windows systems, you can manage certificates with the Microsoft Management Console (MMC) snap-in tool. For more information, see the following Microsoft Documentation on *Working With Certificates*. With this MMC snap-in, you can add and delete certificates.

19.2.6. Use Message Level Security

OpenIDM supports message level security, forcing authentication before granting access. Authentication works by means of a filter-based mechanism that lets you use either an HTTP Basic like mechanism or OpenIDM-specific headers, setting a cookie in the response that you can use for subsequent authentication. If you attempt to access OpenIDM URLs without the appropriate headers or session cookie, OpenIDM returns HTTP 401 Unauthorized, or HTTP 403 Forbidden, depending on the situation. If you use a session cookie, you must include an additional header that indicates the origin of the request.

19.2.6.1. Message Level Security with Logins

The following examples show successful authentication attempts.

```
$ curl \
 --dump-header /dev/stdout \
 --user openidm-admin:openidm-admin \
 "http://localhost:8080/openidm/managed/user?_queryId=query-all-ids"
HTTP/1.1 200 OK
Date: Wed, 07 Sep 2016 12:27:39 GMT
Cache-Control: no-cache
Content-Type: application/json; charset=UTF-8
Set-Cookie: session-jwt=eyJ0eXAiOiJKV1QiLCJjd...; Path=/
Vary: Accept-Encoding, User-Agent
Transfer-Encoding: chunked

{
  "result" : [ ],
  "resultCount" : 0,
  "pagedResultsCookie" : null,
  "totalPagedResultsPolicy" : "NONE",
  "totalPagedResults" : -1,
  "remainingPagedResults" : -1
}
```

```
$ curl \
 --dump-header /dev/stdout \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 "http://localhost:8080/openidm/managed/user?_queryId=query-all-ids"

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-cache
Set-Cookie: session-jwt=2l0zobpuk6st1b2m7gvhg5zas ...;Path=/
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Vary: Accept-Encoding, User-Agent
Content-Length: 82
Server: Jetty(8.y.z-SNAPSHOT)

{"result":[],"resultCount":"0","pagedResultsCookie":null,"remainingPagedResults":-1}


$ curl \
 --dump-header /dev/stdout \
 --header "Cookie: session-jwt=2l0zobpuk6st1b2m7gvhg5zas ..." \
 --header "X-Requested-With: OpenIDM Plugin" \
 "http://localhost:8080/openidm/managed/user?_queryId=query-all-ids"

Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: application/json; charset=UTF-8
Cache-Control: no-cache
Vary: Accept-Encoding, User-Agent
Content-Length: 82
Server: Jetty(8.y.z-SNAPSHOT)
```

Notice that the last example uses the cookie OpenIDM set in the response to the previous request, and includes the `X-Requested-With` header to indicate the origin of the request. The value of the header can be any string, but should be informative for logging purposes. If you do not include the `X-Requested-With` header, OpenIDM returns HTTP 403 Forbidden.

Note

The careful readers among you may notice that the expiration date of the JWT cookie, January 1, 1970, corresponds to the start of UNIX time. Since that time is in the past, browsers will not store that cookie after the browser is closed.

You can also request one-time authentication without a session.

```
$ curl \
 --dump-header /dev/stdout \
 --header "X-OpenIDM-NoSession: true" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 "http://localhost:8080/openidm/managed/user?_queryId=query-all-ids"

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Cache-Control: no-cache
Vary: Accept-Encoding, User-Agent
Content-Length: 82
Server: Jetty(8.y.z-SNAPSHOT)

{"result":[],"resultCount":"0","pagedResultsCookie":null,"remainingPagedResults":-1}
```

19.2.6.2. Sessions and the JWT Cookie

OpenIDM maintains sessions with a JWT session cookie, stored in a client browser. By default, it deletes the cookie when you log out. Alternatively, if you delete the cookie, that ends your session.

You can modify what happens to the session after a browser restart. Open the `authentication.json` file, and change the value of the `sessionOnly` property. For more information on `sessionOnly`, see "Session Module".

The JWT session cookie is based on the `JWT_SESSION` module, described in "Supported Authentication and Session Modules".

19.2.7. Replace Default Security Settings

The default security settings are adequate for evaluation purposes. In production environments, change at least the following settings:

- The password of the default administrative user (`openidm-admin`)
- The default keystore password

Change the Default Administrator Password

1. To change the password of the default administrative user, first retrieve the complete user object to make sure you have the currently assigned roles:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/repo/internal/user/openidm-admin"
{
  "_id": "openidm-admin",
  "_rev": "1",
  "password": "openidm-admin",
  "roles": [
    {
      "_ref": "repo/internal/role/openidm-admin"
    },
    {
      "_ref": "repo/internal/role/openidm-authorized"
    }
  ],
  "userName": "openidm-admin"
}
```

2. Update the password with a PUT request, including the `roles` property that you retrieved in the previous step:

   The following example changes the password of the `openidm-admin` user to `Passw0rd`:

```
$ curl \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request PUT \
 --data '{
     "password": "Passw0rd",
     "userName": "openidm-admin",
     "roles": [
       {
         "_ref": "repo/internal/role/openidm-admin"
       },
       {
         "_ref": "repo/internal/role/openidm-authorized"
       }
     ],
     "_id": "openidm-admin"
     }' \
 "http://localhost:8080/openidm/repo/internal/user/openidm-admin"
{
  "_id": "openidm-admin",
  "_rev": "2",
  "password": {
    "$crypto": {
      "value": {
        "algorithm": "SHA-256",
        "data": "gjTSqGjVyfTLiWRlEemKKArELUipXyaW416y14U9KbWOkvT6ReGu7PffiExIb26K"
      },
      "type": "salted-hash"
    }
  },
  "userName": "openidm-admin",
  "roles": [
    {
      "_ref": "repo/internal/role/openidm-admin"
    },
    {
      "_ref": "repo/internal/role/openidm-authorized"
    }
  ]
}
```

3. Test that the update has been successful by querying OpenIDM with the new credentials:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: Passw0rd" \
 --request GET \
 "http://localhost:8080/openidm/repo/internal/user/openidm-admin"
{
  "_id": "openidm-admin",
  "_rev": "2",
  ...
}
```

Tip

The administrative user can also reset their own password in the Self-Service UI as follows:

1. Log into the Self-Service UI (`https://localhost:8443/`) with the default username and password (`openidm-admin` and `openidm-admin`).
2. In the upper-right corner, select View Profile.
3. On the Password tab, enter and confirm the new password, then click Update.

Change the Default Keystore Password

The default keystore password is `changeit`. You should change this password in a production environment.

To change the default keystore password, follow these steps.

1. Shut down OpenIDM if it is running:

```
$ cd /path/to/openidm
$ ./shutdown.sh
```

2. Use the **keytool** command to change the keystore password. The following command changes the keystore password to `newPassword`:

```
$ keytool \
 -storepasswd \
 -keystore /path/to/openidm/security/keystore.jceks \
 -storetype jceks
Enter keystore password: changeit
New keystore password: newPassword
Re-enter new keystore password: newPassword
```

3. OpenIDM uses a number of encryption keys by default. The passwords of these keys must match the password of the keystore.

   To obtain a list of the keys in the keystore, run the following command:

```
$ keytool \
 -list \
 -keystore /path/to/openidm/security/keystore.jceks \
 -storetype jceks\
 -storepass newPassword
Keystore type: JCEKS
Keystore provider: SunJCE

Your keystore contains 4 entries

openidm-sym-default, 08 Sep 2016, SecretKeyEntry,
openidm-jwtsessionhmac-key, 08 Sep 2016, SecretKeyEntry,
openidm-localhost, 08 Sep 2016, PrivateKeyEntry,
Certificate fingerprint (SHA1): 9C:C7:4F:16:18:CB:78:AA:2E:DD:36:BD:92:FF:7E:28:51:99:81:0C
openidm-selfservice-key, 08 Sep 2016, SecretKeyEntry,
```

   Change the passwords of each of the four default encryption keys as follows:

```
$ keytool \
 -keypasswd \
 -alias openidm-localhost \
 -keystore /path/to/openidm/security/keystore.jceks \
 -storetype jceks
Enter keystore password:  newPassword
Enter key password for <openidm-localhost> changeit
New key password for <openidm-localhost>: newPassword
Re-enter new key password for <openidm-localhost>: newPassword
$ keytool \
 -keypasswd \
 -alias openidm-sym-default \
 -keystore /path/to/openidm/security/keystore.jceks \
 -storetype jceks
Enter keystore password:  newPassword
Enter key password for <openidm-sym-default> changeit
New key password for <openidm-sym-default>: newPassword
Re-enter new key password for <openidm-sym-default>: newPassword
$ keytool \
 -keypasswd \
 -alias openidm-selfservice-key \
 -keystore /path/to/openidm/security/keystore.jceks \
 -storetype jceks
Enter keystore password:  newPassword
Enter key password for <openidm-selfservice-key> changeit
New key password for <openidm-selfservice-key>: newPassword
```

```
Re-enter new key password for <openidm-selfservice-key>: newPassword
$ keytool \
 -keypasswd \
 -alias openidm-jwtsessionhmac-key \
 -keystore /path/to/openidm/security/keystore.jceks \
 -storetype jceks
Enter keystore password:  newPassword
Enter key password for <openidm-jwtsessionhmac-key> changeit
New key password for <openidm-jwtsessionhmac-key>: newPassword
Re-enter new key password for <openidm-jwtsessionhmac-key>: newPassword
```

4. Generate an obfuscated and an encrypted version of your new password, by using the crypto bundle provided with OpenIDM.

   The following example generates an obfuscated and encrypted version of the password `newPassword`:

   ```
   $ java -jar /path/to/openidm/bundle/openidm-util-5.0.0.jar
   This utility helps obfuscate passwords to prevent casual observation.
   It is not securely encrypted and needs further measures to prevent disclosure.
   Please enter the password:newPassword
   OBF:1uo91vn61ymf1sgo1v1p1ym71v2p1siu1ylz1vnw1unp
   CRYPT:dc5aa3ee8f58f7e2c04bbb0d09118199
   ```

5. Open your project's `conf/boot/boot.properties` file and comment out the default keystore password.

   Paste the obfuscated or the encrypted password as the value of the `openidm.keystore.password` property. For example, the following excerpt of a `boot.properties` file removes the default keystore password and sets the keystore password to the obfuscated value of `newPassword`:

   ```
   $ more conf/boot/boot.properties
   ...
   # Keystore password, adjust to match your keystore and protect this file
   # openidm.keystore.password=changeit
   openidm.truststore.password=changeit

   # Optionally use the crypto bundle to obfuscate the password and set one of these:
   openidm.keystore.password=OBF:1uo91vn61ymf1sgo1v1p1ym71v2p1siu1ylz1vnw1unp
   #openidm.keystore.password=CRYPT:...
   ```

   Set *either* the obfuscated or the encrypted password value here, not both.

6. Restart OpenIDM.

   ```
   $ ./startup.sh
   ```

Important

Repeat this procedure on each node if you run multiple nodes in a cluster to ensure that the new password is present on all nodes.

19.2.8. Secure Jetty

If you do not want to use regular HTTP on a production OpenIDM system, you need to make two changes.

First, edit the `openidm/conf/jetty.xml` configuration file. Comment out or delete the `<Call name="addConnector">` code block that includes the `openidm.port.http` property. Keep the `<Call name="addConnector">` code blocks that contain the `openidm.port.https` and `openidm.port.mutualauth` properties. You can set the value for these properties in the `conf/boot/boot.properties` file.

Second, edit the `openidm/conf/config.properties` configuration file. Set the `org.osgi.service.http.enabled` property to false, as shown in the following excerpt:

```
# Enable pax web http/https services to enable jetty
org.osgi.service.http.enabled=false
org.osgi.service.http.secure.enabled=true
```

19.2.9. Protect Sensitive REST Interface URLs

Anything attached to the router is accessible with the default policy, including the repository. If you do not need such access, deny it in the authorization policy to reduce the attack surface.

In addition, you can deny direct HTTP access to system objects in production, particularly access to `action`. As a rule of thumb, do not expose anything that is not used in production.

For an example that shows how to protect sensitive URLs, see "Understanding the Access Configuration Script (`access.js`)".

OpenIDM supports native query expressions on the repository, and you can enable these over HTTP. For example, the following query returns all managed users in an OrientDB repository:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 "http://localhost:8080/openidm/managed/user?_queryExpression=select+*+from+managed_user"
```

By default, direct HTTP access to native queries is disallowed, and should remain so in production systems.

For testing or development purposes, it can be helpful to enable native queries on the repository over HTTP. To do so, edit the access control configuration file (`access.js`). In that file, remove any instances of `"disallowQueryExpression()"` such as the following:

```
// openidm-admin can request nearly anything (except query expressions on repo endpoints)
{
    "pattern"   : "*",
    "roles"     : "openidm-admin",
    "methods"   : "*", // default to all methods allowed
    "actions"   : "*", // default to all actions allowed
 // "customAuthz" : "disallowQueryExpression()",
    "excludePatterns": "repo,repo/*"
},
// additional rules for openidm-admin that selectively enable certain parts of system/
{
    "pattern"   : "system/*",
    "roles"     : "openidm-admin",
    "methods"   : "create,read,update,delete,patch,query", // restrictions on 'action'
    "actions"   : ""
 // "customAuthz" : "disallowQueryExpression()"
},
```

19.2.10. Protect Sensitive Files & Directories

Protect OpenIDM files from access by unauthorized users.

In particular, prevent other users from reading files in at least the `openidm/conf/boot/` and `openidm/security/` directories.

The objective is to limit access to the user that is running the service. Depending on the operating system and configuration, that user might be `root`, `Administrator`, `openidm`, or something similar.

Protecting key files in Unix

1. For the target directory, and the files therein, make sure user and group ownership is limited to the user that is running the OpenIDM service.

2. Disable access of any sort for `other` users. One simple command for that purpose, from the `/path/to/openidm` directory, is:

   ```
   # chmod -R o-rwx .
   ```

Protecting key files in Windows

1. The OpenIDM process in Windows is normally run by the `Administrator` user.

2. If you are concerned about the security of the administrative account, you can `Deny` permissions on the noted directories to existing users, or alternatively the `Users` group.

19.2.11. Remove or Protect Development & Debug Tools

Before you deploy OpenIDM in production, remove or protect development and debug tools, including the Felix web console that is exposed under `/system/console`. Authentication for this console is not integrated with authentication for OpenIDM.

- To remove the Felix web console, remove the web console bundle and all of the plugin bundles related to the web console, as follows:

  ```
  $ cd /path/to/openidm/bundle
  $ rm org.apache.felix.webconsole*.jar
  $ rm openidm-felix-webconsole-5.0.0.jar
  ```

  Also remove the `felix.webconsole.json` from your project's `conf` directory.

  ```
  $ cd /path/to/project-dir
  $ rm conf/felix.webconsole.json
  ```

- Alternatively, protect access to the Felix web console by changing the credentials in your project's `conf/felix.webconsole.json` file. This file contains the username and password to access the console, by default:

  ```
  {
    "username" : "admin",
    "password" : "admin"
  }
  ```

19.2.12. Protect the Repository

You must use a supported JDBC repository in production (see "*Installing a Repository For Production*" in the *Installation Guide*.

Use a strong password for the JDBC connection and change at least the password of the database user (`openidm` by default). When you change the database username and/or password, you must update the database connection configuration file (`datasource.jdbc-default.json`) for your repository type.

You can use property substitution to set the database password and place an obfuscated version of the password in your project's `conf/boot/boot.properties` file or `conf/system.properties` file, or pass it in using the `OPENIDM_OPTS` environment variable.

The following excerpt of a MySQL connection configuration file sets the database password to the value of the `openidm.repo.password` property.

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

In your `boot.properties` file, you would include the obfuscated value of that password as follows:

1. Generate an obfuscated and an encrypted version of your password, by using the crypto bundle provided with OpenIDM:

   ```
   $ java -jar /path/to/openidm/bundle/openidm-util-5.0.0.jar
   This utility helps obfuscate passwords to prevent casual observation.
   It is not securely encrypted and needs further measures to prevent disclosure.
   Please enter the password: newPassword
   OBF:1uo91vn61ymf1sgo1v1p1ym71v2p1siu1ylz1vnw1unp
   CRYPT:dc5aa3ee8f58f7e2c04bbb0d09118199
   ```

2. Paste the obfuscated password as the value of the `openidm.repo.password` property, for example:

   ```
   $ more conf/boot/boot.properties
   openidm.repo.password=OBF:1uo91vn61ymf1sgo1v1p1ym71v2p1siu1ylz1vnw1unp
   ```

Alternatively, you can set the obfuscated password in the `OPENIDM_OPTS` environment variable and export that variable before startup. You must include the JVM memory options when you set the `OPENIDM_OPTS` variable. For example:

```
$ export OPENIDM_OPTS="-Xmx1024m -Xms1024m -Dopenidm.repo
.password=OBF:1uo91vn61ymf1sgo1v1p1ym71v2p1siu1ylz1vnw1unp"
$ ./startup.sh
Executing ./startup.sh...
Using OPENIDM_HOME:   /path/to/openidm
Using PROJECT_HOME:   /path/to/openidm
Using OPENIDM_OPTS:   -Xmx1024m -Xms1024m -Dopenidm.repo
.password=OBF:1uo91vn61ymf1sgo1v1p1ym71v2p1siu1ylz1vnw1unp
Using LOGGING_CONFIG: -Djava.util.logging.config.file=/path/to/openidm/conf/logging.properties
Using boot properties at /path/to/openidm/conf/boot/boot
.properties
-> OpenIDM version "5.0.0"
OpenIDM ready
```

Use a case sensitive database, particularly if you work with systems with different identifiers that match except for case. Otherwise correlation queries or correlation scripts can pick up identifiers that should not be considered the same.

19.2.13. Remove OrientDB Studio

OpenIDM ships with the OrientDB Studio web application. ForgeRock strongly recommends that you remove the web application before deploying in a production environment. To remove OrientDB studio, delete the following directory:

```
/path/to/openidm/db/util/orientdb
```

Verify that the application has been removed by trying to access `http://localhost:2480/`.

Note that an error will be logged on startup when you have removed OrientDB Studio. You can safely ignore this error.

19.2.14. Adjust Log Levels

Leave log levels at `INFO` in production to ensure that you capture enough information to help diagnose issues. For more information, see "*Configuring Server Logs*".

At start up and shut down, `INFO` can produce many messages. Yet, during stable operation, `INFO` generally results in log messages only when coarse-grain operations such as scheduled reconciliation start or stop.

19.2.15. Set Up Restart At System Boot

You can run OpenIDM in the background as a service (daemon), and add startup and shutdown scripts to manage the service at system boot and shutdown. For more information, see "*Starting and Stopping the Server*".

See your operating system documentation for details on adding a service such as OpenIDM to be started at boot and shut down at system shutdown.

19.2.16. Disable the API Explorer

As described in "API Explorer", OpenIDM includes an implementation of the *OpenAPI Initiative Specification*, also known as Swagger.

The API Explorer can help you identify endpoints, and run REST calls against those endpoints. To hide that information in production, disable the following property in the `conf/boot/boot.properties` file for your project:

```
openidm.apidescriptor.enabled=false
```

You can also remove this property from `boot.properties`, as it is `false` by default.

19.3. Configuring a Hardware Security Module (HSM) Device

You can configure an external PKCS #11 (HSM) device to manage the keys used to secure OpenIDM transactions.

> **Note**
>
> On Windows systems using the 64-bit JDK, the Sun PKCS #11 provider is available *only* from JDK version 1.8b49 onwards. If you want to use a PKCS #11 device on Windows, either use the 32-bit version of the JDK, or upgrade your 64-bit JDK to version 1.8b49 or higher.

19.3.1. Setting Up the HSM Configuration

This section assumes that you have access to an HSM device (or a software emulation of an HSM device, such as SoftHSM) and that the HSM provider has been configured and initialized.

The command-line examples in this section use SoftHSM for testing purposes. Before you start, set the correct environment variable for the SoftHSM configuration, for example:

```
$ export SOFTHSM2_CONF=/usr/local/Cellar/softhsm/2.0.0/etc/softhsm2.conf
```

Also initialize slot `0` on the provider, with a command similar to the following:

```
$ softhsm2-util --init-token --slot 0 --label "My token 1"
```

This token initialization requests two PINs—an SO PIN and a user PIN. You can use the SO PIN to reinitialize the token. The user PIN is provided to IDM so that it can interact with the token. Remember the values of these PINs because you will use them later in this section.

The PKCS#11 standard uses a configuration file to interact with the HSM device. The following example shows a basic configuration file for SoftHSM:

```
name = softHSM
library = /usr/local/Cellar/softhsm/2.0.0/lib/softhsm/libsofthsm2.so
slot = 1
attributes(generate, *, *) = {
   CKA_TOKEN = true
}
attributes(generate, CKO_CERTIFICATE, *) = {
   CKA_PRIVATE = false
}
attributes(generate, CKO_PUBLIC_KEY, *) = {
   CKA_PRIVATE = false
}
attributes(*, CKO_SECRET_KEY, *) = {
   CKA_PRIVATE = false
   CKA_EXTRACTABLE = true
}
```

Your HSM configuration file *must* include at least the following settings:

`name`

A suffix to identify the HSM provider. This example uses the `softHSM` provider.

`library`

The path to the PKCS #11 library.

`slot`

The slot number to use, specified as a string. Make sure that the slot you specify here has been initialized on the HSM device.

The `attributes` properties specify additional PKCS #11 attributes that are set by the HSM. For a complete list of these attributes, see the PKCS #11 Reference.

Important

If you are using the JWT Session Module, you *must* set `CKA_EXTRACTABLE = true` for secret keys in your HSM configuration file. For example:

```
attributes(*, CKO_SECRET_KEY, *) = {
   CKA_PRIVATE = false
   CKA_EXTRACTABLE = true
}
```

The HSM provider must allow secret keys to be extractable because the authentication service serializes the JWT Session Module key and passes it to the authentication framework as a base 64-encoded string.

The section that follows assumes that your HSM configuration file is located at `/path/to/hsm/hsm.conf`.

19.3.2. Populating the Default Encryption Keys

When OpenIDM first starts up, it generates a number of encryption keys required to encrypt specific data. If you are using an HSM provider, you must generate these keys manually, as follows:

1. The `openidm-sym-default` key is the default symmetric key required to encrypt the configuration. The following command generates that key in the HSM provider. The `-providerArg` must point to the HSM configuration file described in "Setting Up the HSM Configuration".

   ```
   $ keytool -genseckey \
    -alias openidm-sym-default \
    -keyalg HmacSHA256 \
    -keysize 2048 \
    -keystore NONE \
    -storetype PKCS11 \
    -providerClass sun.security.pkcs11.SunPKCS11 \
    -providerArg /path/to/hsm/hsm.conf
   Enter keystore password:
   ```

   Enter the password of your HSM device. If you are using SoftHSM, enter your user PIN as the keystore password. The remaining sample steps use *user PIN* as the password.

2. The `openidm-selfservice-key` is used by the Self-Service UI to encrypt managed user passwords and other sensitive data. Generate that key with a command similar to the following:

   ```
   $ keytool -genseckey \
    -alias openidm-selfservice-key \
    -keyalg HmacSHA256 \
    -keysize 2048 \
    -keystore NONE \
    -storetype PKCS11 \
    -providerClass sun.security.pkcs11.SunPKCS11 \
    -providerArg /path/to/hsm/hsm.conf
   Enter keystore password: user PIN
   ```

   Enter the password of your HSM device. If you are using SoftHSM, enter your user PIN as the keystore password.

3. The `openidm-jwtsessionhmac-key` is used by the JWT session module to encrypt JWT session cookies. For more information about the JWT session module, see "Supported Session Module". Generate the JWT session module key with a command similar to the following:

```
$ keytool -genseckey \
 -alias openidm-jwtsessionhmac-key \
 -keyalg HmacSHA256 \
 -keysize 2048 \
 -keystore NONE \
 -storetype PKCS11 \
 -providerClass sun.security.pkcs11.SunPKCS11 \
 -providerArg /path/to/hsm/hsm.conf
Enter keystore password: user PIN
```

4. The `openidm-localhost` certificate is used by OpenIDM to support SSL/TLS. Generate that certificate with a command similar to the following:

```
$ keytool -genkey \
 -alias openidm-localhost \
 -keyalg RSA \
 -keysize 2048 \
 -keystore NONE \
 -storetype PKCS11 \
 -providerClass sun.security.pkcs11.SunPKCS11 \
 -providerArg /path/to/hsm/hsm.conf
Enter keystore password: user PIN
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:  OpenIDM Self-Signed Certificate
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=localhost, OU=Unknown, O=OpenIDM Self-Signed Certificate, L=Unknown, ST=Unknown, C=Unknown
 correct?
  [no]:  yes
```

5. If you are *not* using the HSM provider for the truststore, you must add the certificate generated in the previous step to the default OpenIDM truststore.

   If you *are* using the HSM provider for the truststore, you can skip this step.

   To add the `openidm-localhost` certificate to the OpenIDM truststore, export the certificate from the HSM provider, then import it into the truststore, as follows:

```
$ keytool -export \
 -alias openidm-localhost \
 -file exportedCert \
 -keystore NONE \
 -storetype PKCS11 \
 -providerClass sun.security.pkcs11.SunPKCS11 \
 -providerArg /path/to/hsm/hsm.conf
Enter keystore password: user PIN
Certificate stored in file exportedCert
$ keytool -import \
 -alias openidm-localhost \
  -file exportedCert \
  -keystore /path/to/openidm/security/truststore

Enter keystore password: changeit
Owner: CN=localhost, OU=Unknown, O=OpenIDM Self-Signed Certificate, L=...
Issuer: CN=localhost, OU=Unknown, O=OpenIDM Self-Signed Certificate, L=...
Serial number: 5d2554bd
Valid from: Fri Aug 19 13:11:54 SAST 2016 until: Thu Nov 17 13:11:54 SAST 2016
Certificate fingerprints:
  MD5:  F1:9B:72:7F:7B:79:58:29:75:85:82:EC:79:D8:F9:8D
  SHA1: F0:E6:51:75:AA:CB:14:3D:C5:E2:EB:E5:7C:87:C9:15:43:19:AF:36
  SHA256: 27:A5:B7:0E:94:9A:32:48:0C:22:0F:BB:7E:3C:22:2A:64:B5:45:24:14:70:...
  Signature algorithm name: SHA256withRSA
  Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 7B 5A 26 53 61 44 C2 5A   76 E4 38 A8 52 6F F2 89  .Z&SaD.Zv.8.Ro..
0010: 20 04 52 EE                                         .R.
]
]
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

The default truststore password is *changeit*.

19.3.3. Configuring OpenIDM to Support an HSM Provider

To enable support for an HSM provider in OpenIDM, edit your project's `conf/boot/boot.properties` file. That file already contains a commented out configuration property that points to the HSM configuration:

```
# PKCS#11 configuration file
#openidm.security.pkcs11.config=
```

Uncomment that line and specify the path to your HSM configuration file, described in "Setting Up the HSM Configuration". For example:

```
# PKCS#11 configuration file
openidm.security.pkcs11.config=/path/to/hsm/hsm.conf
```

To configure an HSM provider for the OpenIDM keystore, add the following lines to the same section of the `boot.properties` file:

```
openidm.keystore.type=PKCS11
openidm.keystore.provider=SunPKCS11-softHSM
openidm.keystore.location=none
openidm.keystore.password=HSM SO Pin
```

If you are using the HSM provider for the OpenIDM truststore, add following lines to the same section of the `boot.properties` file:

```
openidm.truststore.type=PKCS11
openidm.truststore.provider=SunPKCS11-softHSM
openidm.truststore.location=none
openidm.truststore.password=HSM SO Pin
```

Remove or comment out the other keystore and (optionally) truststore parameters in the `boot.properties` file:

```
# openidm.keystore.type=JCEKS
# openidm.truststore.type=JKS
# openidm.keystore.provider=SunJCE
# openidm.truststore.provider=SUN
# openidm.keystore.location=security/keystore.jceks
# openidm.truststore.location=security/truststore

# Keystore password, adjust to match your keystore and protect this file
# openidm.keystore.password=changeit
# openidm.truststore.password=changeit
```

You should now be able to start OpenIDM with the keys in the HSM provider.

Chapter 20

Integrating Business Processes and Workflows

Key to any identity management solution is the ability to provide workflow-driven provisioning activities, whether for self-service actions such as requests for entitlements, roles or resources, running sunrise or sunset processes, handling approvals with escalations, or performing maintenance.

OpenIDM provides an embedded workflow and business process engine based on Activiti and the Business Process Model and Notation (BPMN) 2.0 standard.

More information about Activiti and the Activiti project can be found at http://www.activiti.org.

This chapter describes how to configure the Activiti engine, and how to manage workflow tasks and processes over the REST interface. You can also manage workflows in the Admin UI by selecting Manage > Workflow and then selecting Tasks or Processes.

20.1. BPMN 2.0 and the Activiti Tools

Business Process Model and Notation 2.0 is the result of consensus among Business Process Management (BPM) system vendors. The Object Management Group (OMG) has developed and maintained the BPMN standard since 2004.

The first version of the BPMN specification focused only on graphical notation, and quickly became popular with the business analyst audience. BPMN 1.x defines how constructs such as human tasks, executable scripts, and automated decisions are visualized in a vendor-neutral, standard way. The second version of BPMN extends that focus to include execution semantics, and a common exchange format. Thus, BPMN 2.0 process definition models can be exchanged not only between different graphical editors, but can also be executed as is on any BPMN 2.0-compliant engine, such as the engine embedded in OpenIDM.

Using BPMN 2.0, you can add artifacts describing workflow and business process behavior to OpenIDM for provisioning and other purposes. For example, you can craft the actual artifacts defining business processes and workflow in a text editor, or using a special Eclipse plugin. The Eclipse plugin provides visual design capabilities, simplifying packaging and deployment of the artifact to OpenIDM. For instructions on installing Activiti Eclipse BPMN 2.0 Designer, see the corresponding Alfresco documentation.

Also, read the Activiti *User Guide* section covering *BPMN 2.0 Constructs*, which describes in detail the graphical notations and XML representations for events, flows, gateways, tasks, and process constructs.

With the latest version of Activiti, JavaScript tasks can be added to workflow definitions. However, OpenIDM functions cannot be called from a JavaScript task in a workflow. Therefore, you can use JavaScript for non-OpenIDM workflow tasks, but you must use the `activiti:expression` construct to call OpenIDM functions.

20.2. Enabling Workflows

OpenIDM embeds an Activiti Process Engine that is started in the OSGi container. When you have started OpenIDM, run the **scr list** command in the OSGi console to check that the workflow bundle is enabled:

```
OpenIDM ready
-> scr list
  BundleId Component Name Default State
     Component Id State      PIDs (Factory PID)
  ...
  [  52]   org.forgerock.openidm.workflow  enabled
     [  18] [active    ] org.forgerock.openidm.workflow
  ...
```

Workflows are not active by default. To activate the workflow bundle, OpenIDM requires two configuration files:

- `workflow.json` specifies the configuration of the Activiti engine, including the data source that the Activiti engine will use.
- `datasource.jdbc-default.json` the default data source for Activiti.

To enable workflows, log in to the Admin UI and select Configure > System Preferences > Workflow, then select the Enable switch and click Save. Enabling workflows through the UI creates the default workflow configuration files in your project's `conf/` directory. To change the default Activiti configuration, see "Configuring the Activiti Engine". To change the data store that Activiti uses, see "Configuring the Activiti Data Source".

20.2.1. Configuring the Activiti Engine

The default `workflow.json` file that is created by the UI has the following structure:

```
{
    "useDataSource" : "default",
    "workflowDirectory" : "&{idm.instance.dir}/workflow"
}
```

`useDataSource`

Specifies the datasource configuration file that points to the repository where Activiti should store its data.

By default, this is the `datasource.jdbc-default.json` file. For information about changing the data store that Activiti uses, see "Configuring the Activiti Data Source".

`workflowDirectory`

Specifies the location in which OpenIDM expects to find workflow processes. By default, OpenIDM looks for workflow processes in the `project-dir/workflow` directory.

In addition to these default properties, you can configure the following elements of the Activiti engine:

```
{
    "mail" : {
        "host" : "yourserver.smtp.com",
        "port" : 587,
        "username" : "yourusername",
        "password" : "yourpassword",
        "starttls" : true
    },
    "history" : "audit"
}
```

`mail`

Specifies the details of the mail server that Activiti will use to send email notifications. By default, Activiti uses the mail server `localhost:25`. To specify a different mail server, enter the details of the mail server here.

`history`

Specifies the history level that should be used for the Activiti engine.

The Activiti history level determines how much historical information is retained when workflows are executed. The history level can be one of the following:

- `none`. No history archiving is done. This level results in the best performance for workflow execution, but no historical information is available.
- `activity`. Archives all process instances and activity instances. No details are archived.
- `audit`. This is the default level. All process instances, activity instances and submitted form properties are archived so that all user interaction through forms is traceable and can be audited.
- `full`. This is the highest level of history archiving and has the greatest performance impact. This history level stores all the information that is stored for the `audit` level, as well as any process variable updates.

20.2.2. Configuring the Activiti Data Source

By default, the Activiti engine runs with an embedded H2 database. The connection details to that H2 database are specified in the `datasource.jdbc-default.json` file.The default `datasource.jdbc-default.json` file that is created by the UI has the following structure:

```
{
    "driverClass" : "org.h2.Driver",
    "jdbcUrl" : "jdbc:h2:file:&{idm.install.dir}/db/activiti/database;MVCC=FALSE;DB_CLOSE_DELAY=0",
    "databaseName" : "activiti",
    "username" : "sa",
    "password" : {encrypted password},
    "connectionTimeout" : 30000,
    "connectionPool" : {
        "type" : "hikari",
        "minimumIdle" : 1,
        "maximumPoolSize" : 5
    }
}
```

Warning

- The embedded H2 database is only for evaluation/demo purposes and is *not* supported for production use.
- Do not configure the H2 WebConsole to listen to external connections via the JDBC URL parameter `-webAllowOthers`. Doing so can expose the system to remote attacks.
- If you require remote connections to the H2 WebConsole for development purposes, set the admin username and password combination within the JDBC URL.

If you are using a JDBC repository for OpenIDM data, you will already have a `datasource.jdbc-default.json` file in your project's `conf/` directory. In this case, when you enable workflows, OpenIDM uses the existing JDBC repository and creates the required Activiti tables in that JDBC repository.

To specify that Activiti should use a data source that is separate to your existing OpenIDM JDBC repository, create a new datasource configuration file in your project's `conf/` directory (for example `datasource.jdbc-activiti.json`) with the connection details to the separate data source. Then reference that file in the `useDataSource` property of the `workflow.json` file (for example, `"useDataSource" : "activiti"`.

For more information about the fields in this file, see "Understanding the JDBC Connection Configuration File".

20.3. Testing the Workflow Integration

OpenIDM reads workflow definitions from the `/path/to/openidm/workflow` directory.

A sample workflow (`example.bpmn20.xml`) is provided in the `samples/example-configurations/workflow` directory. To test the workflow integration, create a `workflow` directory under `/path/to/openidm` and copy the sample workflow to that directory:

```
$ cd /path/to/openidm/
$ mkdir workflow
$ cp samples/example-configurations/workflow/example.bpmn20.xml workflow/
```

Verify the workflow integration by using the REST API. The following REST call lists the defined workflows:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processdefinition?_queryId=query-all-ids"
```

The sample workflow definition that you copied in the previous step is named `osgiProcess`. The result of the preceding REST call therefore includes output similar to the following:

```
{
   "result": [
     {
       "_id": "osgiProcess:1:4",
       "_rev": "1",
       "candidateStarterGroupIdExpressions": [],
       "candidateStarterUserIdExpressions": [],
       "category": "Examples",
       "deploymentId": "1",
       "description": null,
       "eventSupport": {},
       "executionListeners": {},
       "graphicalNotationDefined": false,
       "hasStartFormKey": false,
       "historyLevel": null,
       "ioSpecification": null,
       "key": "osgiProcess",
       "laneSets": [],
       "name": "Osgi process",
       "participantProcess": null,
       "processDiagramResourceName": "OSGI-INF/activiti/example.osgiProcess.png",
       "properties": {},
       "resourceName": "OSGI-INF/activiti/example.bpmn20.xml",
       "revisionNext": 2,
       "startFormHandler": null,
       "suspended": false,
       "suspensionState": 1,
       "taskDefinitions": null,
       "tenantId": "",
       "variables": null,
       "version": 1
     }
   ],
   ...
}
```

The `osgiProcess` workflow calls OpenIDM, queries the available workflow definitions from Activiti, then prints the list of workflow definitions to the OpenIDM logs. Invoke the `osgiProcess` workflow with the following REST call:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --header "Content-Type: application/json" \
  --request POST \
  --data '{"_key":"osgiProcess"}' \
  "http://localhost:8080/openidm/workflow/processinstance?_action=create"
{
    "_id": "5",
    "processInstanceId": "5",
    "processDefinitionId": "osgiProcess:1:4",
    "businessKey": null,
    "status": "ended"
}
```

The workflow prints the list of workflow definitions to the OSGi console. With the default sample, you should see something like this:

```
script task using resolver: [
   pagedResultsCookie:null,
   remainingPagedResults:-1,
   result:[
     [
         tenantId:,
         candidateStarterGroupIdExpressions:[],
         candidateStarterUserIdExpressions:[],
         participantProcess:null,
         processDiagramResourceName:null,
         historyLevel:null,
         hasStartFormKey:false,
         laneSets:[],
         version:1, _id:osgiProcess:1:3,
         description:null,
         name:Osgi process,
         executionListeners:[:],
         key:osgiProcess,
         resourceName:OSGI-INF/activiti/example.bpmn20.xml,
         ioSpecification:null,
         taskDefinitions:null,
         suspensionState:1,
         deploymentId:1,
         properties:[:],
         startFormHandler:null,
         suspended:false,
         variables:null,
         _rev:1,
         revisionNext:2,
         category:Examples,
         eventSupport:[:],
         graphicalNotationDefined:false
     ]
   ]
]
script task using expression resolver: [
   pagedResultsCookie:null,
   remainingPagedResults:-1,
   result:[
     [
        tenantId:,
        candidateStarterGroupIdExpressions:[],
        ...
    ]
]
```

20.4. Defining Activiti Workflows

The following section outlines the process to follow when you create an Activiti workflow for OpenIDM.

1. Define your workflow in a text file, either using an editor, such as Activiti Eclipse BPMN 2.0 Designer, or a simple text editor.

2. Save the workflow definition with a `bpmn20.xml` extension.

   Note that each workflow definition references a script, in the `<scriptTask>` element. The `scriptFormat` of these scripts is always `groovy`. Currently only Groovy script is supported for workflow scripts.

3. Package the workflow definition file as a `.bar` file (Business Archive File). If you are using Eclipse to define the workflow, a `.bar` file is created when you select "Create deployment artifacts". A `.bar` file is essentially the same as a `.zip` file, but with the `.bar` extension.

4. Copy the `.bar` file to the `openidm/workflow` directory.

5. Invoke the workflow using a script (in `openidm/script/`) or directly using the REST interface. For more information, see "Invoking Activiti Workflows".

   You can also schedule the workflow to be invoked repeatedly, or at a future time. For more information, see "*Scheduling Tasks and Events*".

20.5. Invoking Activiti Workflows

You can invoke workflows and business processes from any trigger point within OpenIDM, including reacting to situations discovered during reconciliation. Workflows can be invoked from script files, using the `openidm.create()` function, or directly from the REST interface.

The following sample script extract shows how to invoke a workflow from a script file:

```
/*
 * Calling 'myWorkflow' workflow
 */

var params = {
  "_key": "myWorkflow"
};

openidm.create('workflow/processinstance', null, params);
```

The `null` in this example indicates that you do not want to specify an ID as part of the create call. For more information, see "openidm.create(resourceName, newResourceId, content, params, fields)".

You can invoke the same workflow from the REST interface with the following REST call:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --header "Content-Type: application/json" \
  --request POST \
  --data '{"_key":"myWorkflow"}' \
  "http://localhost:8080/openidm/workflow/processinstance?_action=create"
```

There are two ways in which you can specify the workflow definition that is used when a new workflow instance is started.

- `_key` specifies the `id` attribute of the workflow process definition, for example:

  ```
  <process id="sendNotificationProcess" name="Send Notification Process">
  ```

  If there is more than one workflow definition with the same `_key` parameter, the latest deployed version of the workflow definition is invoked.

- `_processDefinitionId` specifies the ID that is generated by the Activiti Process Engine when a workflow definition is deployed, for example:

  ```
  "sendNotificationProcess:1:104";
  ```

  To obtain the `processDefinitionId`, query the available workflows, for example:

  ```
  {
    "result": [
       {
         "name": "Process Start Auto Generated Task Auto Generated",
         "_id": "ProcessSAGTAG:1:728"
       },
       {
         "name": "Process Start Auto Generated Task Empty",
         "_id": "ProcessSAGTE:1:725"
       },
       ...
  ```

  If you specify a `_key` and a `_processDefinitionId`, the `_processDefinitionId` is used because it is more precise.

Use the optional `_businessKey` parameter to add specific business logic information to the workflow when it is invoked. For example, the following workflow invocation assigns the workflow a business key of `"newOrder"`. This business key can later be used to query "newOrder" processes.

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request POST \
  --data '{"_key":"myWorkflow", "_businessKey":"newOrder"}' \
  "http://localhost:8080/openidm/workflow/processinstance?_action=create"
```

Access to workflows is based on OpenIDM roles, and is configured in your project's conf/process-access.json file. For more information, see "Managing User Access to Workflows".

20.6. Querying Activiti Workflows

The Activiti implementation supports filtered queries that enable you to query the running process instances and tasks, based on specific query parameters. To perform a filtered query send a GET request to the workflow/processinstance context path, including the query in the URL.

For example, the following query returns all process instances with the business key "newOrder", as invoked in the previous example.

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processinstance?_queryId=filtered-
query&processInstanceBusinessKey=newOrder"
```

Any Activiti properties can be queried using the same notation, for example, processDefinitionId=managedUserApproval:1:6405. The query syntax applies to all queries with _queryId=filtered-query. The following query returns all process instances that were started by the user openidm-admin:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processinstance?_queryId=filtered-query&startUserId=openidm-
admin"
```

You can also query process instances based on the value of any process instance variable, by prefixing the variable name with var-. For example:

```
var-processvariablename=processvariablevalue
```

20.7. Using Custom Templates for Activiti Workflows

The embedded Activiti engine is integrated with the default user interface. For simple workflows, you can use the standard Activiti form properties, and have the UI render the corresponding generic forms automatically. If you require a more complex form template, (including input validation, rich input field types, complex CSS, and so forth) you must define a custom form template.

There are two ways in which you can define custom form templates for your workflows:

- Create an HTML template, and refer to that template in the workflow definition.

  This is the recommended method of creating custom form templates. To refer to the HTML template in the workflow definition, use the activiti:formKey attribute, for example activiti:formKey="nUCStartForm.xhtml".

The HTML file must be deployed as part of the workflow definition. Create a .zip file that contains the HTML template and the workflow definition file. Rename the .zip file with a .bar extension.

- Use an embedded template within the workflow definition.

  This method is not ideal, because the HTML code must be escaped, and is difficult to read, edit, or maintain, as a result. Also, sections of HTML code will most likely need to be duplicated if your workflow includes multiple task stages. However, you might want to use this method if your form is small, not too complex and you do not want to bother with creating a separate HTML file and .bar deployment.

20.8. Managing Workflows Over the REST Interface

In addition to the queries described previously, the following examples show the context paths that are exposed for managing workflows over the REST interface. The example output is based on the sample workflow that is provided in `openidm/samples/sync-asynchronous`. For a complete reference of all the context paths related to workflows, see "Managing Workflows Over REST".

openidm/workflow/processdefinition

- List the available workflow definitions:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processdefinition?_queryId=query-all-ids"
{
  "result" : [ {
    "_id" : "managedUserApproval:1:4",
    "_rev" : "1",
    "candidateStarterGroupIdExpressions" : [ ],
    "candidateStarterUserIdExpressions" : [ ],
    "category" : "Examples",
    "deploymentId" : "1",
    "description" : null,
    "eventSupport" : { },
    "executionListeners" : { },
    "graphicalNotationDefined" : false,
    "hasStartFormKey" : false,
    "historyLevel" : null,
    "ioSpecification" : null,
    "key" : "managedUserApproval",
    "laneSets" : [ ],
    "name" : "Managed User Approval Workflow",
    "participantProcess" : null,
    "processDiagramResourceName" : "OSGI-INF/activiti/managedUserApproval.managedUserApproval.png",
    "properties" : { },
    "resourceName" : "OSGI-INF/activiti/managedUserApproval.bpmn20.xml",
    "revisionNext" : 2,
    "startFormHandler" : null,
    "suspended" : false,
    "suspensionState" : 1,
    "taskDefinitions" : null,
    "tenantId" : "",
    "variables" : null,
    "version" : 1
  } ],
  "resultCount" : 1,
  "pagedResultsCookie" : null,
  "totalPagedResultsPolicy" : "NONE",
  "totalPagedResults" : -1,
  "remainingPagedResults" : -1
}
```

- List the workflow definitions, based on certain filter criteria:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processdefinition?_queryId=filtered-query&category=Examples"
{
  "result" : [ {
     "_id" : "managedUserApproval:1:4",
     ...
     "category" : "Examples",
     ...
     "name" : "Managed User Approval Workflow",
     ...
   } ],
   ...
}
```

openidm/workflow/processdefinition/{id}

- Obtain detailed information for a process definition, based on the ID. You can determine the ID by querying all the available process definitions, as described in the first example in this section.

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processdefinition/managedUserApproval:1:4"
{
  "_id" : "managedUserApproval:1:4",
  "_rev" : "2",
  "candidateStarterGroupIdExpressions" : [ ],
  "candidateStarterUserIdExpressions" : [ ],
  "category" : "Examples",
  "deploymentId" : "1",
  "description" : null,
  "eventSupport" : { },
  "executionListeners" : {
    "end" : [ { } ]
  },
  "graphicalNotationDefined" : true,
  "hasStartFormKey" : false,
  "historyLevel" : null,
  "ioSpecification" : null,
  "key" : "managedUserApproval",
  "laneSets" : [ ],
  "name" : "Managed User Approval Workflow",
  "participantProcess" : null,
  "processDiagramResourceName" : "OSGI-INF/activiti/managedUserApproval.managedUserApproval.png",
  "properties" : {
     "documentation" : null
  },
  "resourceName" : "OSGI-INF/activiti/managedUserApproval.bpmn20.xml",
  "revisionNext" : 3,
  "startFormHandler" : {
    "deploymentId" : "1",
    "formKey" : null,
    "formPropertyHandlers" : [ ]
  },
   "suspended" : false,
   "suspensionState" : 1,
    "taskDefinitions" : {
      "evaluateRequest" : {
        "assigneeExpression" : {
          "expressionText" : "openidm-admin"
        },
        "candidateGroupIdExpressions" : [ ],
        "candidateUserIdExpressions" : [ ],
        "categoryExpression" : null,
        "descriptionExpression" : null,
        "dueDateExpression" : null,
        "key" : "evaluateRequest",
        "nameExpression" : {
          "expressionText" : "Evaluate request"
        },
        "ownerExpression" : null,
        "priorityExpression" : null,
        "taskFormHandler" : {
          "deploymentId" : "1",
          "formKey" : null,
          "formPropertyHandlers" : [ {
            "defaultExpression" : null,
            "id" : "requesterName",
            "name" : "Requester's name",
            "readable" : true,
            "required" : false,
            "type" : null,
            "variableExpression" : {
              "expressionText" : "${sourceId}"
            },
        "variableName" : null,
        "writable" : false
      }, {
        "defaultExpression" : null,
        "id" : "requestApproved",
        "name" : "Do you approve the request?",
        "readable" : true,
        "required" : true,
        "type" : {
          "name" : "enum",
          "values" : {
            "true" : "Yes",
            "false" : "No"
        }
      },
      "variableExpression" : null,
      "variableName" : null,
      "writable" : true
      } ]
    },
    "taskListeners" : {
      "assignment" : [ { } ],
      "create" : [ { } ],
      "complete" : [ {
        "className" : "org.activiti.engine.impl.bpmn.listener.ScriptTaskListener",
        "multiInstanceActivityBehavior" : null
      } ]
    }
   }
  },
  "tenantId" : "",
  "variables" : { },
  "version" : 1,
    "formProperties" : [ ]
}
```

- Delete a workflow process definition, based on its ID. Note that you cannot delete a process definition if there are currently running instances of that process definition.

  OpenIDM picks up workflow definitions from the files located in the `/path/to/openidm/workflow` directory. If you delete the workflow definition (`.xml` file) from this directory, the OSGI bundle is deleted. However, deleting this file does not remove the workflow definition from the Activiti engine. You must therefore delete the definition over REST, as shown in the following example.

  Note that, although there is only one representation of a workflow definition in the file system, there might be several versions of the same definition in Activiti. If you want to delete redundant process definitions, delete the definition over REST, *making sure that you do not delete the latest version*.

  ```
  $ curl \
    --header "X-OpenIDM-Username: openidm-admin" \
    --header "X-OpenIDM-Password: openidm-admin" \
    --header "If-Match: *" \
    --request DELETE \
    "http://localhost:8080/openidm/workflow/processdefinition/managedUserApproval:1:3"
  ```

  The delete request returns the contents of the deleted workflow definition.

openidm/workflow/processinstance

- Start a workflow process instance. For example:

```
$ curl \
  --header "Content-Type: application/json" \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --data '{"_key":"managedUserApproval"}' \
  --request POST \
  "http://localhost:8080/openidm/workflow/processinstance?_action=create"
{
  "_id" : "42",
  "processInstanceId" : "42",
  "processDefinitionId" : "managedUserApproval:1:4",
  "businessKey" : null,
  "status" : "suspended"
}
```

- Obtain the list of running workflows (process instances). The query returns a list of IDs. For example:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processinstance?_queryId=query-all-ids"
{
  "result" : [ {
    "_id" : "42",
    "businessKey" : null,
    "deleteReason" : null,
    "durationInMillis" : null,
    "endActivityId" : null,
    "endTime" : null,
    "processDefinitionId" : "managedUserApproval:1:4",
    "processInstanceId" : "42",
    "processVariables" : { },
    "queryVariables" : null,
    "startActivityId" : "start",
    "startTime" : "2018-01-09T14:15:36.550Z",
    "startUserId" : "openidm-admin",
    "superProcessInstanceId" : null,
    "tenantId" : "",
    "processDefinitionResourceName" : "Managed User Approval Workflow"
  } ],
  "resultCount" : 1,
  "pagedResultsCookie" : null,
  "totalPagedResultsPolicy" : "NONE",
  "totalPagedResults" : -1,
  "remainingPagedResults" : -1
}
```

- Obtain the list of running workflows based on specific filter criteria.

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processinstance?_queryId=filtered-
query&businessKey=myBusinessKey"
```

openidm/workflow/processinstance/{id}

- Obtain the details of the specified process instance. For example:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processinstance/42"
{
  "_id" : "42",
  "businessKey" : null,
  "deleteReason" : null,
  "durationInMillis" : null,
  "endActivityId" : null,
  "endTime" : null,
  "processDefinitionId" : "managedUserApproval:1:4",
  "processInstanceId" : "42",
  "processVariables" : {
      ...
  },
  "queryVariables" : [ {
      ...
  } ],
 "startActivityId" : "start",
 "startTime" : "2018-01-09T14:15:36.550Z",
 "startUserId" : "openidm-admin",
 "superProcessInstanceId" : null,
 "tenantId" : "",
 "openidmObjectId" : "openidm-admin",
 "processDefinitionResourceName" : "Managed User Approval Workflow",
 "tasks" : [ {
      ...
 } ]
}
```

- Stop the specified process instance. For example:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request DELETE \
  "http://localhost:8080/openidm/workflow/processinstance/42"
{
  "_id" : "42",
  "businessKey" : null,
  "deleteReason" : null,
  "durationInMillis" : null,
  "endActivityId" : null,
  "endTime" : null,
  "processDefinitionId" : "managedUserApproval:1:4",
  "processInstanceId" : "42",
  "processVariables" : { },
  "queryVariables" : null,
  "startActivityId" : "start",
  "startTime" : "2018-01-09T14:15:36.550Z",
  "startUserId" : "openidm-admin",
  "superProcessInstanceId" : null,
  "tenantId" : ""
}
```

The delete request returns the contents of the deleted process instance.

openidm/workflow/processinstance/history

- List the running and completed workflows (process instances).

  The following query returns two process instances - one that has completed (`"endActivityId": "end"`) and one that is still running (`"endActivityId": null`):

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processinstance/history?_queryId=query-all-ids"
{
  "result" : [ {
    "_id" : "35",
    "businessKey" : null,
    "deleteReason" : "Deleted by Openidm",
    "durationInMillis" : 310686,
    "endActivityId" : null,
    "endTime" : "2018-01-09T14:20:28.342Z",
    "processDefinitionId" : "managedUserApproval:1:4",
    "processInstanceId" : "35",
    "processVariables" : { },
    "queryVariables" : null,
    "startActivityId" : "start",
    "startTime" : "2018-01-09T14:15:17.656Z",
    "startUserId" : "openidm-admin",
    "superProcessInstanceId" : null,
    "tenantId" : "",
    "processDefinitionResourceName" : "Managed User Approval Workflow"
  }, {
    "_id" : "42",
    "businessKey" : null,
    "deleteReason" : null,
    "durationInMillis" : null,
    "endActivityId" : null,
    "endTime" : null,
    "processDefinitionId" : "managedUserApproval:1:4",
    "processInstanceId" : "42",
    "processVariables" : { },
    "queryVariables" : null,
    "startActivityId" : "start",
    "startTime" : "2018-01-09T14:15:36.550Z",
    "startUserId" : "openidm-admin",
    "superProcessInstanceId" : null,
    "tenantId" : "",
    "processDefinitionResourceName" : "Managed User Approval Workflow"
  } ],
    "resultCount" : 2,
    "pagedResultsCookie" : null,
    "totalPagedResultsPolicy" : "NONE",
    "totalPagedResults" : -1,
    "remainingPagedResults" : -1
}
```

- Obtain the list of running and completed workflows, based on specific filter criteria.

  The following command returns the running and completed workflows that were launched by `user.0`.

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processinstance/history?_queryId=filtered-
query&startUserId=user.0"
{
  "result" : [ {
    "_id" : "79",
    "businessKey" : null,
    "deleteReason" : null,
    "durationInMillis" : null,
    "endActivityId" : null,
    "endTime" : null,
    "processDefinitionId" : "managedUserApproval:1:4",
    "processInstanceId" : "79",
    "processVariables" : { },
    "queryVariables" : null,
    "startActivityId" : "start",
    "startTime" : "2018-01-09T14:35:02.514Z",
    "startUserId" : "user.0",
    "superProcessInstanceId" : null,
    "tenantId" : "",
    "processDefinitionResourceName" : "Managed User Approval Workflow"
  } ],
  "resultCount" : 1,
  "pagedResultsCookie" : null,
  "totalPagedResultsPolicy" : "NONE",
  "totalPagedResults" : -1,
  "remainingPagedResults" : -1
}
```

For large result sets, you can use the `_sortKeys` parameter with a `filtered-query` to order search results by one or more fields. You can prefix a `-` character to the field name to specify that results should be returned in descending order, rather than ascending order.

The following query orders results according to their `startTime`. The `-` character in this case indicates that results should be sorted in reverse order, that is, with the most recent results returned first.

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processinstance/history?_queryId=filtered-query&_sortKeys=-
startTime"
{
  "result" : [ {
    "_id" : "79",
    "businessKey" : null,
    "deleteReason" : null,
    "durationInMillis" : null,
    "endActivityId" : null,
    "endTime" : null,
    "processDefinitionId" : "managedUserApproval:1:4",
    "processInstanceId" : "79",
    "processVariables" : { },
    "queryVariables" : null,
    "startActivityId" : "start",
    "startTime" : "2018-01-09T14:35:02.514Z",
    "startUserId" : "user.0",
    "superProcessInstanceId" : null,
    "tenantId" : "",
    "processDefinitionResourceName" : "Managed User Approval Workflow"
  }, {
    "_id" : "42",
    "businessKey" : null,
    "deleteReason" : "Deleted by Openidm",
    "durationInMillis" : 347254,
    "endActivityId" : null,
    "endTime" : "2018-01-09T14:21:23.804Z",
    "processDefinitionId" : "managedUserApproval:1:4",
    "processInstanceId" : "42",
    "processVariables" : { },
    "queryVariables" : null,
    "startActivityId" : "start",
    "startTime" : "2018-01-09T14:15:36.550Z",
    "startUserId" : "openidm-admin",
    "superProcessInstanceId" : null,
    "tenantId" : "",
    "processDefinitionResourceName" : "Managed User Approval Workflow"
  }, {
    "_id" : "6",
    "businessKey" : "sourceId: bjensen, targetId: undefined, reconId: 89945d65-9a5f-4dc5-b201-
ef6e167c2921-703",
    "deleteReason" : "Deleted by Openidm",
    "durationInMillis" : 695804,
    "endActivityId" : null,
    "endTime" : "2018-01-09T14:20:35.272Z",
    "processDefinitionId" : "managedUserApproval:1:4",
    "processInstanceId" : "6",
    "processVariables" : { },
    "queryVariables" : null,
    "startActivityId" : "start",
    "startTime" : "2018-01-09T14:08:59.468Z",
    "startUserId" : "openidm-admin",
    "superProcessInstanceId" : null,
 "tenantId" : "",
    "processDefinitionResourceName" : "Managed User Approval Workflow"
  } ],
    "resultCount" : 3,
    "pagedResultsCookie" : null,
    "totalPagedResultsPolicy" : "NONE",
    "totalPagedResults" : -1,
    "remainingPagedResults" : -1
}
```

Caution

The Activiti engine treats certain property values as *strings*, regardless of their actual data type. This might result in results being returned in an order that is different to what you might expect. For example, if you wanted to sort the following results by their `_id` field, `"88", "45", "101"`, you would expect them to be returned in the order `"45", "88", "101"`. Because Activiti treats IDs as strings, rather than numbers, they would be returned in the order `"101", "45", "88"`.

openidm/workflow/processdefinition/{id}/taskdefinition

- Query the list of tasks defined for a specific process definition. For example:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processdefinition/managedUserApproval:1:4/taskdefinition?
_queryId=query-all-ids"
{
  "result" : [ {
    "_id" : "evaluateRequest",
    "assignee" : {
      "expressionText" : "openidm-admin"
    },
    "categoryExpression" : null,
    "descriptionExpression" : null,
    "dueDate" : null,
    "formProperties" : {
      "deploymentId" : "1",
      "formKey" : null,
      "formPropertyHandlers" : [ {
        "_id" : "requesterName",
        "defaultExpression" : null,
        "name" : "Requester's name",
        "readable" : true,
        "required" : false,
        "type" : null,
        "variableExpression" : {
          "expressionText" : "${sourceId}"
        },
      "variableName" : null,
      "writable" : false
    }, {
      "_id" : "requestApproved",
      "defaultExpression" : null,
      "name" : "Do you approve the request?",
      "readable" : true,
      "required" : true,
      "type" : {
        "name" : "enum",
        "values" : {
          "true" : "Yes",
          "false" : "No"
        }
      },
      "variableExpression" : null,
      "variableName" : null,
      "writable" : true
      } ]
    },
    "name" : {
      "expressionText" : "Evaluate request"
    },
    "ownerExpression" : null,
    "priority" : null,
    "taskCandidateGroup" : [ ],
    "taskCandidateUser" : [ ],
    "taskListeners" : {
      "assignment" : [ { } ],
      "create" : [ { } ],
      "complete" : [ {
        "className" : "org.activiti.engine.impl.bpmn.listener.ScriptTaskListener",
        "multiInstanceActivityBehavior" : null
      } ]
    },
    "formResourceKey" : null
  } ],
  "resultCount" : 1,
  "pagedResultsCookie" : null,
  "totalPagedResultsPolicy" : "NONE",
  "totalPagedResults" : -1,
  "remainingPagedResults" : -1
}
```

- Query a task definition based on the process definition ID and the task name (`taskDefinitionKey`). For example:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/processdefinition/managedUserApproval:1:4/taskdefinition/evaluateRequest"
{
  "_id" : "evaluateRequest",
  "assignee" : {
    "expressionText" : "openidm-admin"
  },
  "categoryExpression" : null,
  "descriptionExpression" : null,
  "dueDate" : null,
  "formProperties" : {
    "deploymentId" : "1",
    "formKey" : null,
    "formPropertyHandlers" : [ {
      "_id" : "requesterName",
      "defaultExpression" : null,
      "name" : "Requester's name",
      "readable" : true,
      "required" : false,
      "type" : null,
      "variableExpression" : {
        "expressionText" : "${sourceId}"
      },
      "variableName" : null,
      "writable" : false
    }, {
      "_id" : "requestApproved",
      "defaultExpression" : null,
   "name" : "Do you approve the request?",
   "readable" : true,
   "required" : true,
   "type" : {
    "name" : "enum",
    "values" : {
     "true" : "Yes",
     "false" : "No"
    }
   },
   "variableExpression" : null,
   "variableName" : null,
   "writable" : true
   } ]
  },
  "name" : {
   "expressionText" : "Evaluate request"
  },
  "ownerExpression" : null,
  "priority" : null,
  "taskCandidateGroup" : [ ],
  "taskCandidateUser" : [ ],
  "taskListeners" : {
  "assignment" : [ { } ],
  "create" : [ { } ],
  "complete" : [ {
   "className" : "org.activiti.engine.impl.bpmn.listener.ScriptTaskListener",
   "multiInstanceActivityBehavior" : null
  } ]
 }
}
```

openidm/workflow/taskinstance

- Query all running task instances. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/workflow/taskinstance?_queryId=query-all-ids"
{
 "result" : [ {
  "_id" : "85",
  "_rev" : "1",
  "activityInstanceVariables" : { },
  "cachedElContext" : null,
  "category" : null,
  "createTime" : "2018-01-09T14:35:02.514Z",
  "delegationState" : null,
  "delegationStateString" : null,
  "deleted" : false,
  "description" : null,
  "dueDate" : null,
  "eventName" : null,
  "executionId" : "79",
  "name" : "Evaluate request",
    "owner" : null,
    "parentTaskId" : null,
    "priority" : 50,
    "processDefinitionId" : "managedUserApproval:1:4",
    "processInstanceId" : "79",
    "processVariables" : { },
    "queryVariables" : null,
    "revisionNext" : 2,
    "suspended" : false,
    "suspensionState" : 1,
    "taskDefinitionKey" : "evaluateRequest",
    "taskLocalVariables" : { },
    "tenantId" : "",
     "assignee" : "openidm-admin"
 } ],
 "resultCount" : 1,
 "pagedResultsCookie" : null,
 "totalPagedResultsPolicy" : "NONE",
 "totalPagedResults" : -1,
 "remainingPagedResults" : -1
}
```

- Query task instances based on candidate users or candidate groups. For example:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/taskinstance?_queryId=filtered-
query&taskCandidateUser=manager1"
```

or

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/taskinstance?_queryId=filtered-
query&taskCandidateGroup=management"
```

Note that you can include both users and groups in the same query.

openidm/workflow/taskinstance/{id}

- Obtain detailed information for a running task, based on the task ID. For example:

```
$ curl \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request GET \
  "http://localhost:8080/openidm/workflow/taskinstance/34"
{
  "_id" : "34",
  "_rev" : "1",
  "activityInstanceVariables" : { },
  "cachedElContext" : null,
  "category" : null,
  "createTime" : "2018-01-10T11:19:15.699Z",
  "delegationState" : null,
  "delegationStateString" : null,
  "deleted" : false,
  "description" : null,
  "dueDate" : null,
  "eventName" : null,
  "executionId" : "5",
  "name" : "Evaluate request",
  "owner" : null,
  "parentTaskId" : null,
  "priority" : 50,
  "processDefinitionId" : "managedUserApproval:1:4",
  "processInstanceId" : "5",
  "processVariables" : { },
  "queryVariables" : null,
  "revisionNext" : 2,
  "suspended" : false,
  "suspensionState" : 1,
  "taskDefinitionKey" : "evaluateRequest",
  "taskLocalVariables" : { },
  "tenantId" : "",
  "formProperties" : [ {
    "requesterName" : "bjensen"
    }, {
    "requestApproved" : null
    } ],
    "assignee" : "manager",
    "openidmAssigneeId" : "09bd693a-1c73-45d4-b33b-b0ddfed275be",
    "variables" : {
      "sourceId" : "bjensen",
      "mapping" : "systemCsvfileAccounts_managedUser",
      "openidmObjectId" : "openidm-admin",
      "ambiguousTargetIds" : null,
      "action" : "CREATE",
      "linkQualifier" : "default",
      "_action" : "performAction",
      "reconId" : "bbaa07d5-4d08-4406-80cd-05d7beaa786e-847",
      "situation" : "ABSENT"
    },
    "candidates" : {
      "candidateUsers" : [ "manager" ],
      "candidateGroups" : [ ]
  }
}
```

- Update task-related data stored in the Activiti workflow engine. For example:

```
$ curl \
  --header "Content-Type: application/json" \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --header "If-Match : *" \
  --request PUT \
  --data '{"description":"Evaluate the new managed user request"}' \
  "http://localhost:8080/openidm/workflow/taskinstance/34"
```

- Complete the specified task. The variables required by the task are provided in the request body. For example:

```
$ curl \
  --header "Content-Type: application/json" \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request POST \
  --data '{"requestApproved":"true"}' \
  "http://localhost:8080/openidm/workflow/taskinstance/34?_action=complete"
```

- Claim the specified task. A user who claims a task has that task inserted into his list of pending tasks. The ID of the user who claims the task is provided in the request body. For example:

```
$ curl \
  --header "Content-Type: application/json" \
  --header "X-OpenIDM-Username: openidm-admin" \
  --header "X-OpenIDM-Password: openidm-admin" \
  --request POST \
  --data '{"userId":"manager1"}' \
  "http://localhost:8080/openidm/workflow/taskinstance/34?_action=claim"
```

Chapter 21
Logging Audit Information

The audit service publishes and logs information to one or more specified targets, including local data files, the repository, and remote systems.

Audit logs help you to record activity by account. With audit data, you can monitor logins, identify problems such as unresponsive devices, and collect information to comply with regulatory requirements.

The audit service logs information from six audit events: access details, system activity, authentication operations, configuration changes, reconciliations, and synchronizations. Auditing provides the data for all relevant reports, including those related to orphan accounts.

You can customize data from all six audit events.

Regardless of where audit information is logged, you can query audit logs over the REST interface. For more information, see "Querying Audit Logs Over REST".

21.1. Configuring the Audit Service

You can access the audit logging configuration over REST under the `openidm/config/audit` context path and in the file *`project-dir`*`/conf/audit.json`.

You can use the Admin UI to configure the audit service. Select Configure > System Preferences and click on the Audit tab. The fields on that form correspond to the configuration parameters described in this section.

You can also configure the audit service by editing corresponding parameters in the `audit.json` file.

The following list describes the major options that you can configure for the audit service.

- OpenIDM includes several configurable *audit event handlers*, as described in "Configuring Audit Event Handlers".

  The `availableAuditEventHandlers` property in the `audit.json` file provides the array of audit event handlers available to OpenIDM.

- You *must* configure one audit event handler to manage queries on the audit logs. The default audit query handler is the OpenIDM repository, but you can configure one of the other available event handlers to handle queries. Note that the JMS, Syslog, and Splunk handlers can *not* be used as the handler for queries. The audit event handler that you configure to manage queries must be `enabled`, either by including its definition in `audit.json`, or setting it to Enabled in the Admin UI.

To specify which audit event handler should be used for queries, set the `handlerForQueries` property in the `audit.json` file, as follows:

```
{
   "auditServiceConfig" : {
      "handlerForQueries" : "repo",
      "availableAuditEventHandlers" : [
        "org.forgerock.audit.handlers.csv.CsvAuditEventHandler",
        "org.forgerock.audit.handlers.elasticsearch.ElasticsearchAuditEventHandler",
        "org.forgerock.audit.handlers.jms.JmsAuditEventHandler",
        "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
        "org.forgerock.openidm.audit.impl.RepositoryAuditEventHandler",
        "org.forgerock.openidm.audit.impl.RouterAuditEventHandler",
        "org.forgerock.audit.handlers.splunk.SplunkAuditEventHandler",
        "org.forgerock.audit.handlers.syslog.SyslogAuditEventHandler"
    ],
```

In this case, the `handlerForQueries` is set to `repo`, which is the `name` of the `RepositoryAuditEventHandler`.

- To configure the audit service to log an event, include it in the list of `events` for the specified audit event handler.

- You can allow a common `transactionId` for audit data from all ForgeRock products. To do so, edit the `system.properties` file in your `project-dir/conf` directory and set:

```
org.forgerock.http.TrustTransactionHeader=true
```

Important

- Do not use a file-based audit event handler, such as CSV or JSON, to handle queries *in a clustered environment*. Rather use the repo audit event handler or an external database for queries, in conjunction with your file-based audit handler.

  In a clustered environment, file-based audit logs are really useful only for offline review and parsing with external tools.

  You can use a file-based audit handler for queries in a non-clustered demonstration or evaluation environment. However, be aware that these handlers do not implement paging, and are therefore subject to general query performance limitations.

- The JMS, Syslog, and Splunk handlers can *not* be used as the handler for queries.

- Logging via CSV or JSON may lead to errors in one or more mappings in the Admin UI.

21.2. Configuring Audit Event Handlers

An audit event handler manages audit events, sends audit output to a defined location, and controls the output format. OpenIDM supports several default audit event handlers, plus audit event handlers for third-party log management tools, as described in "*Audit Log Reference*".

Each audit event handler has a set of basic configuration properties, listed in "Common Audit Event Handler Property Configuration". Specific audit event handlers have additional configuration properties described, per handler, in "Audit Event Handler Configuration".

The following sections illustrate how you can configure the standard OpenIDM audit event handlers. For additional audit event handlers, see "*Audit Log Reference*".

21.2.1. JSON Audit Event Handler

The JSON audit event handler logs events as JSON objects to a set of JSON files. This is the default file-based audit event handler in an OpenIDM configuration.

The following excerpt of an `audit.json` file shows a sample JSON audit event handler configuration:

```
"eventHandlers" : [
    {
        "class" : "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
        "config" : {
            "name" : "json",
            "logDirectory" : "&{launcher.working.location}/audit",
            "buffering" : {
                "maxSize" : 100000,
                "writeInterval" : "100 millis"
            },
            "topics" : [
                "access",
                "activity",
                "recon",
                "sync",
                "authentication",
                "config"
            ]
        }
    },
```

A JSON audit event handler configuration includes the following mandatory properties:

`name`

The audit event handler name (`json`).

`logDirectory`

The name of the directory in which the JSON log files should be written, relative to the *working location*. For more information on the working location, see "Specifying the Startup Configuration".

You can use property value substitution to direct log files to another location on the file system. The example provided in "Custom Audit Log Location" shows how to direct audit logs to a user home directory.

`buffering` - `maxSize`

The maximum number of events that can be buffered. The default (and minimum) number of buffered events is 100000.

`buffering` - `writeInterval`

The delay after which the file-writer thread is scheduled to run after encountering an empty event buffer. The default delay is 100 milliseconds.

`topics`

The list of topics for which audit events are logged.

One JSON file is created for each audit topic that is included in this list:

`access.audit.json`
`activity.audit.json`
`authentication.audit.json`
`config.audit.json`
`recon.audit.json`
`sync.audit.json`

For a description of all the configurable properties of the JSON audit event handler, see "JSON Audit Event Handler `config` Properties".

The following excerpt of an `authentication.audit.json` file shows the log message format for authentication events:

```
{
 "context": {
  "ipAddress": "0:0:0:0:0:0:0:1"
 },
 "entries": [{
  "moduleId": "JwtSession",
  "result": "FAILED",
  "reason": {},
  "info": {}
 },
 ...
 {
  "moduleId": "INTERNAL_USER",
  "result": "SUCCESSFUL",
  "info": {
   "org.forgerock.authentication.principal": "openidm-admin"
  }
 }],
 "principal": ["openidm-admin"],
 "result": "SUCCESSFUL",
 "userId": "openidm-admin",
 "transactionId": "94b9b85f-fbf1-4c4c-8198-ab1ff52ed0c3-24",
 "timestamp": "2016-10-11T12:12:03.115Z",
 "eventName": "authentication",
 "trackingIds": ["5855a363-a1e0-4894-a2dc-fd5270fb99d1"],
 "_id": "94b9b85f-fbf1-4c4c-8198-ab1ff52ed0c3-30"
} {
 "context": {
  "component": "repo/internal/user",
  "roles": ["openidm-admin", "openidm-authorized"],
  "ipAddress": "0:0:0:0:0:0:0:1",
  "id": "openidm-admin",
  "moduleId": "INTERNAL_USER"
 }...
```

21.2.2. CSV Audit Event Handler

The CSV audit event handler logs events to a comma-separated value (CSV) file. The following code is an excerpt of the `audit.json` file, which depicts a sample CSV audit event handler configuration:

```
"eventHandlers" : [
{
    "class" : "org.forgerock.audit.events.handlers.csv.CSVAuditEventHandler",
    "config" : {
        "name" : "csv",
        "logDirectory" : "&{launcher.working.location}/audit",
        "topics" : [ "access", "activity", "recon", "sync", "authentication", "config" ]
    }
}
```

The `logDirectory` property indicates the name of the directory in which log files should be written, relative to the *working location*. For more information on the working location, see "Specifying the Startup Configuration".

You can use property value substitution to direct logs to another location on the file system. The example provided in "Custom Audit Log Location" shows how to direct audit logs to a user home directory.

If you set up a custom CSV Audit Event Handler, you may configure over 20 different properties, as described in "Common Audit Event Handler Property Configuration".

Audit file names are fixed and correspond to the event being audited:

`access.csv`
`activity.csv`
`authentication.csv`
`config.csv`
`recon.csv`
`sync.csv`

21.2.2.1. Minimum Admin UI CSV Audit Handler Configuration Requirements

If you configure the CSV Audit Event Handler in the Admin UI, you should at minimum, configure the following:

- The `logDirectory`, the full path to the directory with audit logs, such as `/path/to/openidm/audit`. You can substitute &{launcher.install.location} for `/path/to/openidm`.
- Differing entries for the quote character, `quoteChar` and delimiter character, `delimiterChar`.
- If you enable the CSV tamper-evident configuration, you should include the `keystoreHandlerName`, *or* a `filename` and `password`. Do not include all three options.

  Before including tamper-evident features in the audit configuration, set up the keys as described in "Configuring Keys to Protect Audit Logs Against Tampering".

Note

The `signatureInterval` property supports time settings in a human-readable format (default = 1 hour). Examples of allowable `signatureInterval` settings are:

- 3 days, 4 m
- 1 hour, 3 sec

Allowable time units include:

- days, day, d
- hours, hour, h
- minutes, minute, min, m
- seconds, second, sec, s

21.2.2.2. Configuring Keys to Protect Audit Logs Against Tampering

If the integrity of your audit files is important, you can configure the CSV Audit Event Handler for tamper detection. Before you do so, you must set the keys required to support tamper detection.

OpenIDM already has a Java Cryptography Extension Keystore (JCEKS), `keystore.jceks`, in the `/path/to/openidm/security` directory.

You'll need to initialize a key pair using the RSA encryption algorithm, using the SHA256 hashing mechanism.

```
$ cd /path/to/openidm
$ keytool \
 -genkeypair \
 -alias "Signature" \
 -dname CN=openidm \
 -keystore security/keystore.jceks \
 -storepass changeit \
 -storetype JCEKS \
 -keypass changeit \
 -keyalg RSA \
 -sigalg SHA256withRSA
```

You can now set up a secret key, in Hash-based message authentication code, using the SHA256 hash function (HmacSHA256)

```
$ keytool \
 -genseckey \
 -alias "Password" \
 -keystore security/keystore.jceks \
 -storepass changeit \
 -storetype JCEKS \
 -keypass changeit \
 -keyalg HmacSHA256 \
 -keysize 256
```

To verify your new entries, run the following command:

```
$ keytool \
 -list \
 -keystore security/keystore.jceks \
 -storepass changeit \
 -storetype JCEKS
    Keystore type: JCEKS
Keystore provider: SunJCE

Your keystore contains 5 entries

signature, May 10, 2016, PrivateKeyEntry,
Certificate fingerprint (SHA1): 62:2E:E4:36:74:F1:7F:E9:06:08:8D:77:82:1C:F6:D4:05:D1:20:01
openidm-sym-default, May 10, 2016, SecretKeyEntry,
password, May 10, 2016, SecretKeyEntry,
openidm-selfservice-key, May 10, 2016, SecretKeyEntry,
openidm-localhost, May 10, 2016, PrivateKeyEntry,
Certificate fingerprint (SHA1): 31:D2:33:93:E3:63:E8:06:66:CC:C1:4F:7F:DF:0A:F8:C4:D8:0E:BD
```

21.2.2.3. Configuring Tamper Protection for CSV Audit Logs

Tamper protection can ensure the integrity of OpenIDM audit logs written to CSV files. You can activate tamper protection in the `audit.json` file directly, or by editing the CSV Audit Event Handler through the Admin UI.

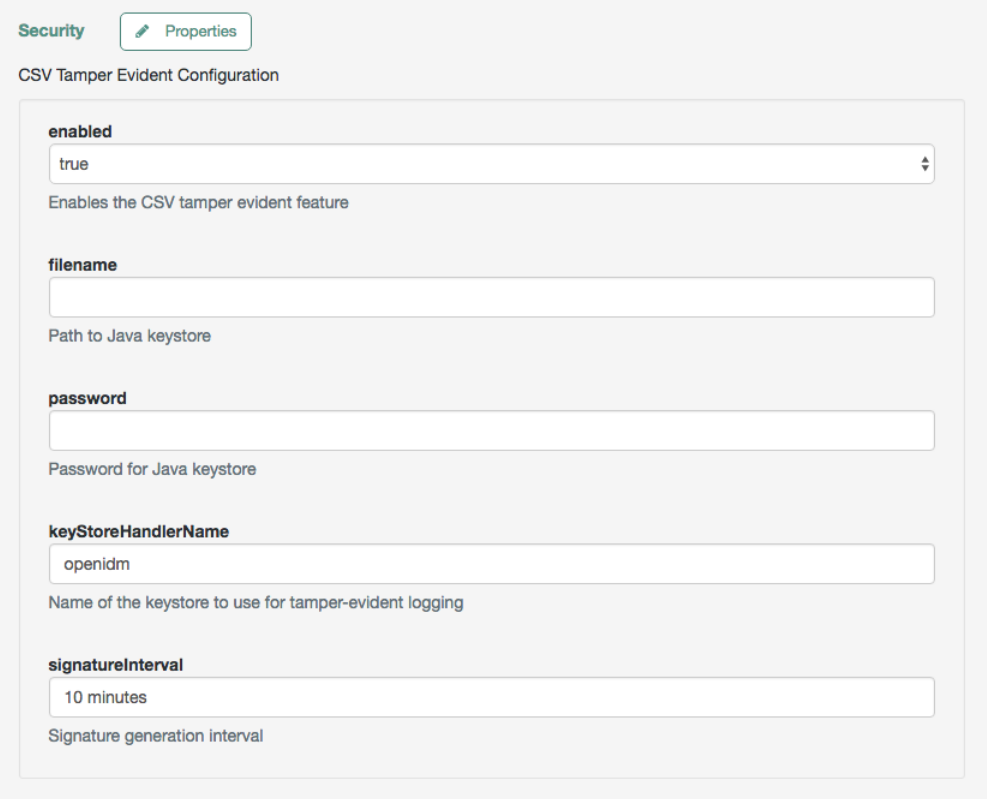
Once configured, the relevant code snippet in your *project-dir*`/conf/audit.conf` file should appear as follows:

```
{
   "class" : "org.forgerock.audit.handlers.csv.CsvAuditEventHandler",
   "config" : {
   ...
      "security" : {
        "enabled" : true,
        "filename" : "",
        "password" : "",
        "keyStoreHandlerName" : "openidm",
        "signatureInterval" : "10 minutes"
      },
     ...
```

This particular code snippet reflects a tamper-evident configuration where a signature is written to a new line in each CSV file, every 10 minutes. That signature uses the default OpenIDM keystore, configured in the *project-dir*/conf/boot/boot.properties file. The properties are described in "Common Audit Event Handler Property Configuration".

To import a certificate into the OpenIDM keystore, or create your own self-signed certificate, read "Configuring Keys to Protect Audit Logs Against Tampering".

To make these same changes in the Admin UI, log into https://localhost:8443/admin, and click Configure > System Preferences > Audit. You can either edit an existing CSV audit event handler, or create one of your own, with the options just described.



CSV Tamper Evident Configuration

enabled
true
Enables the CSV tamper evident feature

filename

Path to Java keystore

password

Password for Java keystore

keyStoreHandlerName
openidm
Name of the keystore to use for tamper-evident logging

signatureInterval
10 minutes
Signature generation interval

Before saving these tamper-evident changes to your audit configuration, move or delete any current audit CSV files with commands such as:

```
$ cd /path/to/openidm
$ mv audit/*.csv /tmp
```

Once you've saved tamper-evident configuration changes, you should see the following files in the /path/to/openidm/audit directory:

```
tamper-evident-access.csv
tamper-evident-access.csv.keystore
tamper-evident-activity.csv
tamper-evident-activity.csv.keystore
tamper-evident-authentication.csv
tamper-evident-authentication.csv.keystore
tamper-evident-config.csv
tamper-evident-config.csv.keystore
tamper-evident-recon.csv
tamper-evident-recon.csv.keystore
tamper-evident-sync.csv
tamper-evident-sync.csv.keystore
```

21.2.2.4. Checking the Integrity of Audit Log Files

Now that you've configured keystore and tamper-evident features, you can periodically check the integrity of your log files.

For example, the following command can verify the CSV files in the **--archive** subdirectory (`audit/`), which belong to the access **--topic**, verified with the `keystore.jceks` keystore, using the OpenIDM CSV audit handler bundle, `forgerock-audit-handler-csv-version.jar`:

```
$ java -jar \
bundle/forgerock-audit-handler-csv-version.jar
 \
--archive audit/
 \
--topic access
 \
--keystore security/keystore.jceks
 \
--password changeit
```

If there are changes to your `tamper-evident-access.csv` file, you'll see a message similar to:

```

    FAIL tamper-evident-access.csv-2016.05.10-11.05.43 The HMac at row 3 is not correct.

```

Note

Note the following restrictions on verification of CSV audit files:

- You can only verify audit files that have already been rotated. You cannot verify an audit file that is currently being written to.
- Verification of tampering is supported only for CSV audit files with the following format:

```
"formatting" : {
    "quoteChar" : "\"",
    "delimiterChar" : ",",
    "endOfLineSymbols" : "\n"
},
```

- A tamper-evident audit configuration rotates files automatically and pairs the rotated file with the required keystore file. Files that are rotated manually cannot be verified as the required keystore information is not appended.

21.2.3. Router Audit Event Handler

The router audit event handler logs events to any external or custom endpoint, such as `system/scriptedsql` or `custom-endpoint/myhandler`. It uses target-assigned values of `_id`.

A sample configuration for a `router` event handler is provided in the `audit.json` file in the `openidm/samples/audit-sample/conf` directory, and described in "Audit Sample Configuration Files" in the *Samples Guide*. This sample directs log output to a JDBC repository. The audit configuration file (`conf/audit.json`) for the sample shows the following event handler configuration:

```
{
    "class": "org.forgerock.openidm.audit.impl.RouterAuditEventHandler",
    "config": {
        "name": "router",
        "topics" : [ "access", "activity", "recon", "sync", "authentication", "config" ],
        "resourcePath" : "system/auditdb"
    }
},
```

The `"resourcePath"` property in the configuration indicates that logs should be directed to the `system/auditdb` endpoint. This endpoint, and the JDBC connection properties, are defined in the connector configuration file (`conf/provisioner.openicf-scriptedsql.json`), as follows:

```
{
    "name" : "auditdb",
...
    "configurationProperties" : {
        "username" : "root",
        "password" : "password",
        "driverClassName" : "com.mysql.jdbc.Driver",
        "url" : "jdbc:mysql://&{openidm.repo.host}:&{openidm.repo.port}/audit",
        "autoCommit" : true,
        "reloadScriptOnExecution" : false,
        "jdbcDriver" : "com.mysql.jdbc.Driver",
        "scriptRoots" : ["&{launcher.project.location}/tools"],
        "createScriptFileName" : "CreateScript.groovy",
        "testScriptFileName" : "TestScript.groovy",
        "searchScriptFileName" : "SearchScript.groovy"
    },
...
```

Include the correct URL or IP address of your remote JDBC repository in the `boot.properties` file for your project.

When JSON information is sent to the router audit event handler, the value of `_id` is replaced with `eventId`.

21.2.4. Repository Audit Event Handler

The repository audit event handler sends information to the OpenIDM repository. The log entries vary by repository:

- In the OrientDB repository, OpenIDM stores log entries in the following tables:
    1. `audit_access`
    2. `audit_activity`
    3. `audit_authentication`
    4. `audit_config`
    5. `audit_recon`
    6. `audit_sync`
- In a JDBC repository, OpenIDM stores log entries in the following tables:
    1. `auditaccess`
    2. `auditactivity`
    3. `auditauthentication`
    4. `auditconfig`
    5. `auditrecon`
    6. `auditsync`

You can use the repository audit event handler to generate reports that combine information from multiple tables.

Each of these JDBC tables maps to an object in the database table configuration file (`repo.jdbc.json`). The following excerpt of that file illustrates the mappings for the `auditauthentication` table:

```
"audit/authentication" : {
    "table" : "auditauthentication",
    "objectToColumn" : {
        "_id" : "objectid",
        "transactionId" : "transactionid",
        "timestamp" : "activitydate",
        "userId" : "userid",
        "eventName" : "eventname",
        "result" : "result",
        "principal" : {"column" : "principals", "type" : "JSON_LIST"},
        "context" : {"column" : "context", "type" : "JSON_MAP"},
        "entries" : {"column" : "entries", "type" : "JSON_LIST"},
        "trackingIds" : {"column" : "trackingids", "type" : "JSON_LIST"},
    }
},
```

The tables correspond to the `topics` listed in the `audit.json` file. For example:

```
{
    "class": "org.forgerock.openidm.audit.impl.RepositoryAuditEventHandler",
    "config": {
        "name": "repo",
        "topics" : [ "access", "activity", "recon", "sync", "authentication", "config" ]
    }
},
```

21.2.5. JMS Audit Event Handler

You can configure a Java Message Service (JMS) Audit Event Handler. The Java Message Service (JMS) is a Java API for sending messages between clients. A JMS audit event handler can record messages between a JMS message broker and one or more clients. The default ForgeRock JMS message broker is *Apache ActiveMQ*. For a demonstration, see "Show Audit Events Published on a JMS Topic" in the *Samples Guide*.

Alternatively, you can use the *TIBCO Enterprise Message Service*, as described in this chapter.

The JMS API architecture includes a *JMS provider*, the messaging system, along with *JMS clients*, the Java programs and components that consume messages. This implementation supports the *Publish/Subscribe Messaging Domain*.

As with other audit event handlers, you can configure it directly through the `conf/audit.json` file for your project or through the Admin UI.

Tip

The JMS audit event handler does not support queries. If you enable JMS, you must also enable a second handler that supports queries. You'll see that handler in the `audit.json` file with the `handlerForQueries` property, or in the Admin UI with the `Use For Queries` option.

The ForgeRock JMS audit event handler supports JMS communication, based on the following components:

- A JMS message broker, which provides clients with connectivity, along with message storage and message delivery functionality.

- JMS messages, which follow a specific format described in "JMS Message Format".

- Destinations, maintained by the message broker, such as the ForgeRock audit service. They may be batched in queues, and can be acknowledged in one of three modes: automatically, by the client, or with direction to accept duplication. The acknowledgement mode is based on the JMS session.

- Topics: JMS topics differ from ForgeRock audit event topics. The ForgeRock implementation of JMS topics uses the publish/subscribe messaging domain, which can direct messages to the JMS audit event handler. In contrast, ForgeRock audit event topics specify categories of events, including access, activity, authentication, configuration, reconciliation, and synchronization.

- JMS clients include both the producer and consumer of a JMS message.

Depending on the configuration, you can expect some or all of these components to be included in JMS audit log messages.

In the following sections, you can configure the JMS audit event handler in the Admin UI, and through your project's `audit.json` file. For detailed configuration options, see "JMS Audit Event Handler Unique `config` Properties". But first, you should add several bundles to your OpenIDM deployment.

21.2.5.1. Adding Required Bundles for the JMS Messaging

To test JMS messaging, you'll download the binary distribution of ActiveMQ, as well as four OSGi Bundle JAR files.

- The ActiveMQ 5.13.2 binary, which you can download from http://activemq.apache.org/activemq-5132-release.html.
- ActiveMQ Client.
- The *bnd* JAR for working with OSGi bundles, which you can download from https://repo1.maven.org/maven2/biz/aQute/bnd/bnd/.
- The Apache Geronimo J2EE management bundle, `geronimo-j2ee-management_1.1_spec-1.0.1.jar`, which you can download from https://repo1.maven.org/maven2/org/apache/geronimo/specs/geronimo-j2ee-management_1.1_spec/1.0.1/>.
- The *hawtbuf* Maven-based protocol buffer compiler JAR, which you can download from hawtbuf-1.11.jar).

> **Note**
>
> The JMS event handler has been tested and documented with the noted versions of the files that you've just downloaded.

Unpack the `apache-activemq-5.13.2-*` binary.

Make sure at least the first two JAR files, for *the Active MQ Client* and *bnd*, are in the same directory. Navigate to that directory, and create an OSGi bundle with the following steps:

1. Create a BND file named `activemq.bnd` with the following contents:

```
version=5.13.2
Export-Package: *;version=${version}
Bundle-Name: ActiveMQ :: Client
Bundle-SymbolicName: org.apache.activemq
Bundle-Version: ${version}
```

2. Run the following command to create the OSGi bundle archive file:

```
$ java \
-jar \
bnd-1.50.0.jar \
wrap \
-properties \
activemq.bnd \
activemq-client-5.13.2.jar
```

3. Rename the `activemq-client-5.13.2.bar` file that appears to `activemq-client-5.13.2-osgi.jar` and copy it to the `/path/to/openidm/bundle` directory.

Copy the other two bundle files, *Apache Geronimo* and *hawtbuf*, to the `/path/to/openidm/bundle` directory.

21.2.5.2. Configuring JMS at the Admin UI

To configure JMS at the Admin UI, select Configure > System Preferences > Audit. Under Event Handlers, select `JmsAuditEventHandler` and select `Add Event Handler`. You can then configure the JMS audit event handler in the pop-up window that appears. For guidance, see "JMS Configuration File".

21.2.5.3. JMS Configuration File

You can configure JMS directly in the `conf/audit.json` file, or indirectly through the Admin UI. The following code is an excerpt of the audit.json file, which depicts a sample JMS audit event handler configuration:

```
{
  "class" : "org.forgerock.audit.handlers.jms.JmsAuditEventHandler",
  "config" : {
    "name": "jms",
    "enabled" : true,
    "topics": [ "access", "activity", "config", "authentication", "sync", "recon" ],
    "deliveryMode": "NON_PERSISTENT",
    "sessionMode": "AUTO",
    "batch": {
      "batchEnabled": true,
      "capacity": 1000,
      "threadCount": 3,
      "maxBatchedEvents": 100
    },
    "jndi": {
      "contextProperties": {
        "java.naming.factory.initial" : "org.apache.activemq.jndi.ActiveMQInitialContextFactory",
        "java.naming.provider.url" : "tcp://127.0.0.1:61616?daemon=true",
        "topic.audit" : "audit"
      },
      "topicName": "audit",
      "connectionFactoryName": "ConnectionFactory"
    }
  }
}
```

As you can see from the properties, in this configuration, the JMS audit event handler is `enabled`, with `NON_PERSISTENT` delivery of audit events in batches. It is configured to use the Apache ActiveMQ Java Naming and Directory Interface (JNDI) message broker, configured on port 61616. For an example of how to configure Apache ActiveMQ, see "Show Audit Events Published on a JMS Topic" in the *Samples Guide*.

If you substitute a different JNDI message broker, you'll have to change the `jndi` `contextProperties`. If you configure the JNDI message broker on a remote system, substitute the associated IP address.

To set up SSL, change the value of the `java.naming.provider.url` to:

```
ssl://127.0.0.1:61617?daemon=true&socket.enabledCipherSuites=
     SSL_RSA_WITH_RC4_128_SHA,SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
```

You'll also need to set up keystores and truststores, as described in "JMS, ActiveMQ, and SSL".

21.2.5.4. JMS, ActiveMQ, and SSL

If the security of your audit data is important, you can configure SSL for JMS. To do so, you'll need to take the following steps to generate an ActiveMQ broker certificate keystore, a broker export certificate, a client keystore, and a server truststore. You can then import that client certificate into the OpenIDM security truststore.

> **Note**
>
> This section is based in part on the ActiveMQ documentation on *How do I use SSL*. As of this writing, it includes the following caution: "In Linux, do not use absolute path to keystore".

But first, you should export two environment variables:

- Navigate to the directory where you unpacked the ActiveMQ binary:

  ```
  $ cd /path/to/apache-activemq-x.y.z
  ```

- **ACTIVEMQ_SSL_OPTS**. Set the `ACTIVEMQ_SSL_OPTS` variable to point to the ActiveMQ broker keystore:

  ```
  $ export \
  ACTIVEMQ_SSL_OPTS=\
  '-Djavax.net.ssl.keyStore=/usr/local/activemq/keystore/broker.ks -Djavax.net.ssl
  .keyStorePassword=changeit'
  ```

- **MAVEN_OPTS** Set the `MAVEN_OPTS` variable, for the sample consumer described in "Configuring and Using a JMS Consumer Application" in the *Samples Guide*:

  ```
  $ export \
  MAVEN_OPTS=\
  "-Djavax.net.ssl.keyStore=client.ks -Djavax.net.ssl
  .keyStorePassword=changeit
  -Djavax.net.ssl.trustStore=client.ts -Djavax.net.ssl.trustStorePassword=changeit"
  ```

Note that these commands use the default keystore `changeit` password. The commands which follow assume that you use the same password when creating ActiveMQ certificates.

- Create an ActiveMQ broker certificate (`broker.ks`):

```
$ keytool \
-genkey
 \
-alias broker
 \
-keyalg RSA
 \
-keystore broker.ks
```

- Export the certificate to `broker_cert`, so you can share it with clients:

```
$ keytool \
-export
 \
-alias broker
 \
-keystore broker.ks
 \
-file broker_cert
```

- Create a client keystore file (`client.ks`):

```
$ keytool \
-genkey
 \
-alias client
 \
-keyalg RSA
 \
-keystore client.ks
```

- Create a client truststore file, `client.ts`, and import the broker certificate, `broker_cert`:

```
$ keytool \
-import
 \
-alias broker
 \
-keystore client.ts
 \
-file broker_cert
```

- Export the client keystore, `client.ks`, into a client certificate file (`client.crt`):

```
$ keytool \
-export
 \
-alias client
 \
-keystore client.ks
 \
--file client.crt
```

- Now make this work with OpenIDM. Import the client certificate file into the OpenIDM truststore:

```
$ keytool \
-import
 \
-trustcacerts
 \
-alias client
 \
-file client.crt
 \
-keystore /path/to/openidm/security/truststore
```

With these certificate files, you can now set up SSL in the ActiveMQ configuration file, `activemq.xml`, in the `/path/to/apache-activemq-x.y.z/conf` directory.

You'll add one line to the `<transportConnectors>` code block with `<transportConnector name="ssl"`, as shown here:

```
<transportConnectors>
     <!-- DOS protection, limit concurrent connections to 1000 and frame size to 100MB -->
     <transportConnector name="openwire" uri="tcp://0.0.0.0:61616?
        maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/>
     <transportConnector name="ssl" uri="ssl://0.0.0.0:61617?transport.enabledCipherSuites=
        SSL_RSA_WITH_RC4_128_SHA,SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
        &amp;maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600&transport.daemon=true"/>
     <transportConnector name="amqp" uri="amqp://0.0.0.0:5672?maximumConnections=1000&amp;
        wireFormat.maxFrameSize=104857600"/>
     <transportConnector name="stomp" uri="stomp://0.0.0.0:61613?maximumConnections=1000&amp;
        wireFormat.maxFrameSize=104857600"/>
     <transportConnector name="mqtt" uri="mqtt://0.0.0.0:1883?maximumConnections=1000&amp;
        wireFormat.maxFrameSize=104857600"/>
     <transportConnector name="ws" uri="ws://0.0.0.0:61614?maximumConnections=1000&amp;
        wireFormat.maxFrameSize=104857600"/>
</transportConnectors>
```

You can now make a corresponding change to the OpenIDM audit configuration file, `audit.json`, as described in "JMS Configuration File".

You can now start the ActiveMQ event broker, and start OpenIDM, as described in "Starting the ActiveMQ Broker and OpenIDM" in the *Samples Guide*.

21.2.5.5. JMS Message Format

The following JMS message reflects the authentication of the `openidm-admin` user, logging into the Admin UI from a remote location, IP address 172.16.209.49.

```
{
  "event": {
    "_id": "134ee773-c081-436b-ae61-a41e8158c712-565",
    "trackingIds": [
      "4dd1f9de-69ac-4721-b01e-666df388fb17",
      "185b9120-406e-47fe-ba8f-e95fd5e0abd8"
    ],
  "context": {
    "id": "openidm-admin",
    "ipAddress": "172.16.209.49",
    "roles": [
      "openidm-admin",
      "openidm-authorized"
    ],
    "component": "repo/internal/user"
  },
  "entries": [
    {
      "info": {
        "org.forgerock.authentication.principal": "openidm-admin"
      },
      "result": "SUCCESSFUL",
      "moduleId": "JwtSession"
    }
  ],
  "principal": [
    "openidm-admin"
  ],
    "result": "SUCCESSFUL",
    "userId": "openidm-admin",
    "transactionId": "134ee773-c081-436b-ae61-a41e8158c712-562",
    "timestamp": "2016-04-15T14:57:53.114Z",
    "eventName": "authentication"
  },
  "auditTopic": "authentication"
}
```

21.2.5.6. JMS, TIBCO, and SSL

OpenIDM also supports integration between the *TIBCO Enterprise Message Service* and the JMS audit event handler.

You'll need to use two bundles from your TIBCO installation: `tibjms.jar`, and if you're setting up a secure connection, `tibcrypt.jar`. With the following procedure, you'll process `tibjms.jar` into an OSGi bundle:

1. Download the *bnd* JAR for working with OSGi bundles, from bnd-1.50.0.jar. If you've previously set up the ActiveMQ server, as described in "Adding Required Bundles for the JMS Messaging", you may have already downloaded this JAR archive.

2. In the same directory, create a file named `tibco.bnd`, and add the following lines to that file:

   ```
   version=8.3.0
   Export-Package: *;version=${version}
   Bundle-Name: TIBCO Enterprise Message Service
   Bundle-SymbolicName: com/tibco/tibjms
   Bundle-Version: ${version}
   ```

3. Add the `tibco.jar` file to the same directory.

4. Run the following command to create the bundle:

```
$ java \
 -jar bnd-1.50.0.jar wrap \
 -properties tibco.bnd tibjms.jar
```

5. Rename the newly created `tibjms.bar` file to `tibjms-osgi.jar`, and copy it to the `/path/to/openidm/bundle` directory.

6. If you're configuring SSL, copy the `tibcrypt.jar` file from your TIBCO installation to the `/path/to/openidm/bundle` directory.

You also need to configure your project's `audit.conf` configuration file. The options are similar to those listed earlier in "JMS Configuration File", except for the following `jndi` code block:

```
"jndi": {
   "contextProperties": {
      "java.naming.factory.initial" : "com.tibco.tibjms.naming.TibjmsInitialContextFactory",
      "java.naming.provider.url" : "tibjmsnaming://localhost:7222"
   },
   "topicName": "audit",
   "connectionFactoryName": "ConnectionFactory"
}
```

If your TIBCO server is on a remote system, substitute appropriately for `localhost`. If you're configuring a secure TIBCO installation, you'll want to configure a different code block:

```
"jndi": {
   "contextProperties": {
      "java.naming.factory.initial" : "com.tibco.tibjms.naming.TibjmsInitialContextFactory",
      "java.naming.provider.url" : "ssl://localhost:7243",
      "com.tibco.tibjms.naming.security_protocol" : "ssl",
      "com.tibco.tibjms.naming.ssl_trusted_certs" : "/path/to/tibco/server/certificate/cert.pem",
      "com.tibco.tibjms.naming.ssl_enable_verify_hostname" : "false"
   },
   "topicName": "audit",
   "connectionFactoryName": "SSLConnectionFactory"
}
```

Do not add the TIBCO certificate to the OpenIDM `truststore` file. The formats are not compatible.

Once this configuration work is complete, don't forget to start your TIBCO server before starting OpenIDM. For more information, see the following *TIBCO Enterprise Message Service Users's Guide* .

21.2.6. Elasticsearch Audit Event Handler

OpenIDM supports the configuration of third-party audit event handlers, such as Elasticsearch, which allows you to log OpenIDM events in file formats compatible with the Elasticsearch search server. Note that ForgeRock does not endorse or support the use of any third-party tools.

The examples in this section assume that the Elasticsearch search server is configured on the same system as your OpenIDM instance. In a deployment environment, such third-party tools are more likely to be running on a remote system. If you have configured a third-party tool on a remote system, the reliability of audit data may vary, depending on the reliability of your network connection. However, you can limit the risks with appropriate buffer settings, which can mitigate issues related to your network connection, free space on your system, and related resources such as RAM. (This is not an exhaustive list.)

21.2.6.1. Installing and Configuring Elasticsearch

This appendix assumes that you are installing Elasticsearch on the same system as OpenIDM. For Elasticsearch downloads and installation instructions, see the Elasticsearch *Getting Started* document.

You can set up Elasticsearch Shield with basic authentication to help protect your audit logs. To do so, read the following Elasticsearch document on *Getting Started with Shield*. Follow up with the following Elasticsearch document on how you can *Control Access with Basic Authentication*.

You can configure SSL for Elasticsearch Shield. For more information, see the following Elasticsearch document: *Setting Up SSL/TLS on a Cluster*.

Import the certificate that you use for Elasticsearch into OpenIDM's truststore, with the following command:

```
$ keytool \
 -import \
 -trustcacerts \
 -alias elasticsearch \
 -file /path/to/cacert.pem \
 -keystore /path/to/openidm/security/truststore
```

Once imported, you can activate the `useSSL` option in the `audit.json` file. If you created an Elasticsearch Shield username and password, you can also associate that information with the `username` and `password` entries in that same `audit.json` file.

21.2.6.2. Creating an Audit Index for Elasticsearch

If you want to create an audit index for Elasticsearch, you must set it up *before* starting OpenIDM, for the audit event topics described in this section: "OpenIDM Audit Event Topics".

To do so, execute the REST call shown in the following audit index file. Note the properties that are `not_analyzed`. Such fields are not indexed within Elasticsearch.

The REST call in the audit index file includes the following URL:

```
http://myUsername:myPassword@localhost:9200/audit
```

That URL assumes that your Elasticsearch deployment is on the localhost system, accessible on default port 9200, configured with an `indexName` of `audit`.

It also assumes that you have configured basic authentication on Elasticsearch Shield, with a username of `myUsername` and a password of `myPassword`.

If any part of your Elasticsearch deployment is different, revise the URL accordingly.

> **Warning**
>
> Do not transmit usernames and passwords over an insecure connection. Enable the `useSSL` option, as described in "Configuring the Elasticsearch Audit Event Handler".

21.2.6.3. Configuring the Elasticsearch Audit Event Handler

"Configuring the Elasticsearch Audit Event Handler via the Admin UI" and "Configuring the Elasticsearch Audit Event Handler in `audit.json`" illustrate how you can configure the Elasticsearch Audit Event Handler.

If you activate the Elasticsearch audit event handler, we recommend that you enable buffering for optimal performance, by setting:

```
"enabled" : true,
```

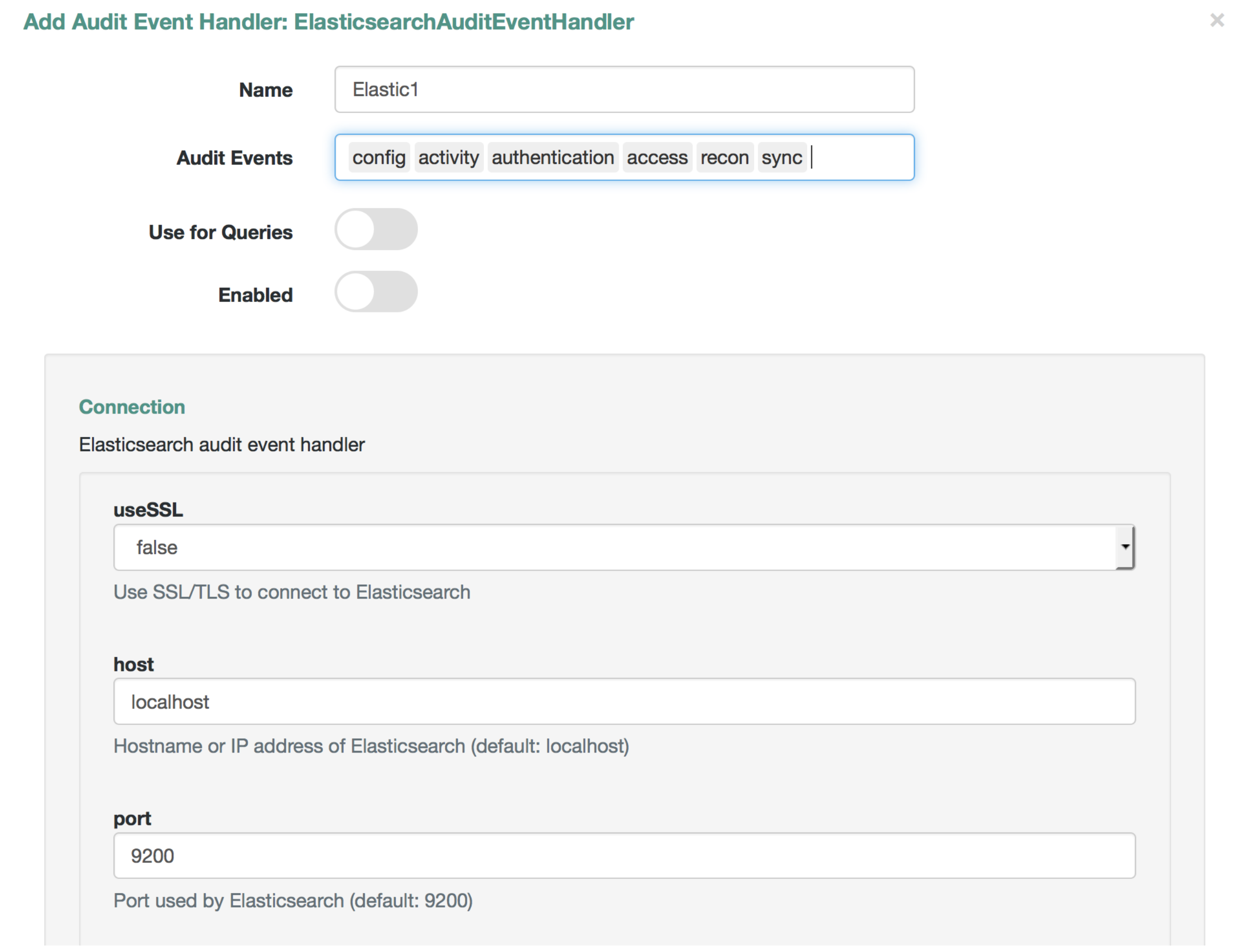
The `buffering` settings shown are not recommendations for any specific environment. If performance and audit data integrity are important in your environment, you may need to adjust these numbers.

If you choose to protect your Elasticsearch deployment with the plugin known as *Shield*, and configure the ability to *Control Access with Basic Authentication*, you can substitute your Elasticsearch Shield `admin` or `power_user` credentials for `myUsername` and `myPassword`.

If you activate the `useSSL` option, install the SSL certificate that you use for Elasticsearch into the OpenIDM keystore. For more information, see the following section: "Accessing the Security Management Service".

21.2.6.3.1. Configuring the Elasticsearch Audit Event Handler via the Admin UI

To configure this event handler through the Admin UI, click Configure > System Preferences > Audit. Select `ElasticsearchAuditEventHandler` from the drop-down text box, click Add Event Handler, and configure it in the window that appears.



Add Audit Event Handler: ElasticsearchAuditEventHandler

Name: Elastic1

Audit Events: config activity authentication access recon sync

Use for Queries: off

Enabled: off

Connection

Elasticsearch audit event handler

useSSL: false

Use SSL/TLS to connect to Elasticsearch

host: localhost

Hostname or IP address of Elasticsearch (default: localhost)

port: 9200

Port used by Elasticsearch (default: 9200)

For a list of properties, see "Common Audit Event Handler Property Configuration".

21.2.6.3.2. Configuring the Elasticsearch Audit Event Handler in `audit.json`

Alternatively, you can configure the Elasticsearch audit event handler in the `audit.json` file for your project.

The following code is an excerpt from the `audit.json` file, with Elasticsearch configured as the handler for audit queries:

```
{
   "auditServiceConfig" : {
      "handlerForQueries" : "elasticsearch",
      "availableAuditEventHandlers" : [
        "org.forgerock.audit.handlers.csv.CsvAuditEventHandler",
        "org.forgerock.audit.handlers.elasticsearch.ElasticsearchAuditEventHandler",
        "org.forgerock.audit.handlers.jms.JmsAuditEventHandler",
        "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
        "org.forgerock.openidm.audit.impl.RepositoryAuditEventHandler",
        "org.forgerock.openidm.audit.impl.RouterAuditEventHandler",
        "org.forgerock.audit.handlers.splunk.SplunkAuditEventHandler",
        "org.forgerock.audit.handlers.syslog.SyslogAuditEventHandler"
    ],
```

You should also set up configuration for the Elasticsearch event handler. The entries shown are defaults, and can be configured. In fact, if you have set up Elasticsearch Shield, with or without SSL/TLS, as described in "Installing and Configuring Elasticsearch", you should change some of these defaults.

```
    "eventHandlers" : [
    {
       "name" : "elasticsearch"
       "class" : "org.forgerock.audit.handlers.elasticsearch.ElasticsearchAuditEventHandler",
       "config" : {
          "connection" : {
             "useSSL" : false,
             "host" : "localhost",
             "port" : "9200"
          },
          "indexMapping" : {
             "indexName" : "audit"
          },
          "buffering" : {
             "enabled" : false,
             "maxSize" : 20000,
             "writeInterval" : "1 second",
             "maxBatchedEvents" : "500"
          }
          "topics" : [
             "access",
             "activity",
             "recon",
             "sync",
             "authentication",
             "config"
          ]
       }
    }
 ],
```

If you set `useSSL` to true, add the following properties to the `connection` code block:

```
    "username" : "myUsername",
     "password" : "myPassword",
```

For more information on the other options shown in `audit.json`, see "Common Audit Event Handler Property Configuration".

21.2.6.4. Querying and Reading Elasticsearch Audit Events

By default, Elasticsearch uses pagination. As noted in the following Elasticsearch document on *Pagination*, queries are limited to the first 10 results.

For example, the following query is limited to the first 10 results:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request GET \
 "http://localhost:8080/openidm/audit/access?_queryFilter=true"
```

To override the limit of 10 results, follow the guidance shown in "Paging and Counting Query Results" for `pageSize`.

To set up a `queryFilter` that uses a "starts with" `sw` or "equals" `eq` comparison expression, you will need to set it up as a `not_analyzed` string field, as described in the following Elasticsearch document on *Term Query.*. You should also review the section on "Comparison Expressions". If you haven't already done so, you may need to modify and rerun the REST call described in "Creating an Audit Index for Elasticsearch".

The `queryFilter` output should include UUIDs as `id` values for each audit event. To read audit data for that event, include that UUID in the URL. For example, the following REST call specifies an access event, which includes data on the client:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request GET
 "http://localhost:8080/openidm/audit/access/75ca07f5-836c-4e7b-beaa-ae968325a529-622"
```

21.2.7. Syslog Audit Event Handler

The Syslog audit event handler enables you to log messages to a Syslog server, based on the *Syslog Protocol*.

You can configure the Syslog audit event handler in the Admin UI, or directly through the `audit.json` file for your project. The following excerpt from this file depicts a possible Syslog configuration in `audit.json`:

```
{
    "class" : "org.forgerock.audit.handlers.syslog.SyslogAuditEventHandler",
    "config" : {
        "protocol" : "UDP",
        "host" : "172.16.206.5",
        "port" : 514,
        "connectTimeout" : 5,
        "facility" : "KERN",
        "severityFieldMappings" : [
            {
                "topic" : "recon",
                "field" : "exception",
                "valueMappings" : {
                    "SEVERE" : "EMERGENCY",
                    "INFO" : "INFORMATIONAL"
                }
            }
        ],
        "buffering" : {
            "enabled" : false
        },
        "name" : "syslog1",
        "topics" : [
            "config",
            "activity",
            "authentication",
            "access",
            "recon",
            "sync"
        ],
        "enabled" : true
    }
}
```

The `name`, `topics`, and `enabled` options in the last part of the excerpt are common to all audit event handlers. For detailed information on the remaining properties, see "Syslog Audit Event Handler Unique `config` Properties".

21.2.8. Splunk Audit Event Handler

The Splunk audit event handler logs OpenIDM events to a Splunk system.

> **Important**
>
> Currently, the Splunk audit event handler can only be used to write events to Splunk. It cannot read or query audit events. You must therefore use the Splunk audit event handler in tandem with another event handler that is configured to handle queries.

Splunk enables you to define the structure of the incoming data. To use the event handler with OpenIDM, create a new data Source Type in Splunk, that will be associated with the incoming OpenIDM log data. Because the audit event handler users the HTTP endpoints in Splunk, you must also enable a Splunk HTTP Event Collector. The HTTP Event Collector provides an authorization token that allows OpenIDM to log events to Splunk.

The following procedure assumes a Splunk instance running on the same host as OpenIDM. Adjust the instructions for your Splunk system:

1. Create a new source type:

    a. In the Splunk UI, select Data > Source Types > New Source Type.

   b. Provide a name for the source type, for example, `openidm`.

   c. Under Event Breaks, specify how the incoming messages are split.

      The Splunk audit event handler supports bulk handing, so it passes multiple audit events to Splunk at a time, as a large JSON payload.

      Select Regex and enter `^{` to indicate how the bulk messages are separated.

   d. Under Timestamp, click Auto to specify that Splunk should generate the timestamp, then click Save.

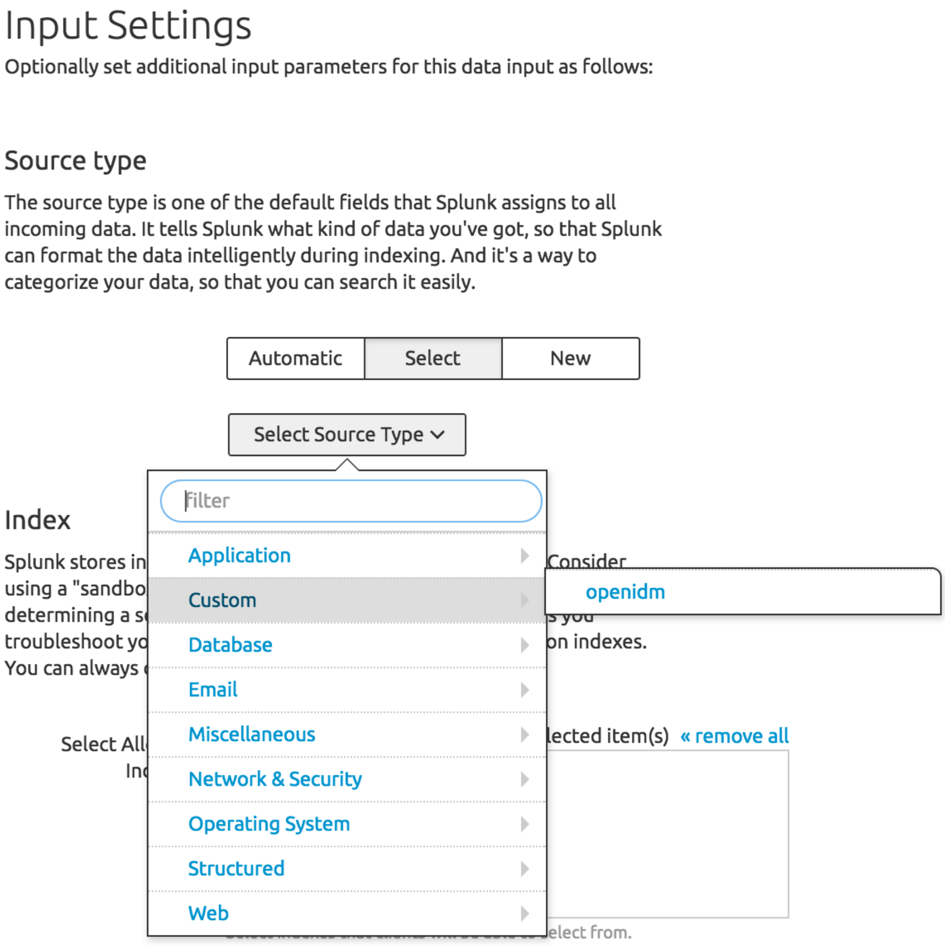
2. Create a new HTTP Event Collector.

   a. Select Data Inputs > HTTP Event Collector > New Token.

   b. Enter a Name that will be associated with this token, for example, `openidm`.

      Other fields are optional.

   c. On the Input Settings screen, click Select under Source Type, then select Custom > openidm from the Select Source Type list.



Input Settings

Optionally set additional input parameters for this data input as follows:

Source type

The source type is one of the default fields that Splunk assigns to all incoming data. It tells Splunk what kind of data you've got, so that Splunk can format the data intelligently during indexing. And it's a way to categorize your data, so that you can search it easily.

d. Click Review, then Submit.

> **Important**
>
> Splunk provides the authorization token that you must add as the value of the `authzToken` property in the Splunk audit event handler configuration.

e. Make sure that the Global Settings for HTTP Event Collectors do not conflict with the settings you have configured for this OpenIDM HTTP Event Collector.

To add the Splunk audit event handler to your OpenIDM configuration, update your project's `audit.json` file or select Configure > System Preferences > Audit in the Admin UI, then select SplunkAuditEventHandler and click Add Event Handler.

The following excerpt of an `audit.json` file shows a sample Splunk audit event handler configuration. Adjust the connection settings and `authzToken` to match your Splunk system.

```
{
    "class" : "org.forgerock.audit.handlers.splunk.SplunkAuditEventHandler",
    "config" : {
        "connection" : {
            "useSSL" : false,
            "host" : "localhost",
            "port" : 8088
        },
        "buffering" : {
            "maxSize" : 10000,
            "writeInterval" : "100 ms",
            "maxBatchedEvents" : 500
        },
        "authzToken" : "87E9C00F-F5E6-47CF-B62F-E415A8142355",
        "name" : "Splunk",
        "topics" : [
            "config",
            "activity",
            "authentication",
            "access",
            "recon",
            "sync"
        ],
        "enabled" : true
    }
}
```

All properties are mandatory. For a complete list of the configurable properties for this audit event handler, see "Splunk Audit Event Handler `config` Properties".

21.2.9. Reviewing Active Audit Event Handlers

To review the audit event handlers available for your OpenIDM deployment, along with each setting shown in the `audit.json` file, use the following command to POST a request for `availableHandlers`:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request POST \
"http://localhost:8080/openidm/audit?_action=availableHandlers"
```

The output includes a full set of options for each audit event handler, which have been translated in the Admin UI. You can see "human-readable" details when you log into the Admin UI. Click Configure > System Preferences > Audit, and create or customize the event handler of your choice.

Not all audit event handlers support queries. You'll see this in the REST call output as well as in the Admin UI. In the output for `availableHandlers`, you'll see:

```
"isUsableForQueries" : false
```

In the Admin UI, when you configure the JMS audit event handler, you won't be able to enable the `Use For Queries` option.

21.3. Audit Log Event Topics

The OpenIDM audit service logs information from six audit topics: access, activity, authentication, configuration, reconciliation, and synchronization.

When you start OpenIDM, it creates audit log files in the `openidm/audit` directory. The default file-based audit event handler is the JSON handler, which creates one JSON file for each audit event topic.

To configure default and custom audit topics in the Admin UI, select Configure > System Preferences. Click on the Audit tab, and scroll down to Event Topics.

21.3.1. OpenIDM Audit Event Topics

The OpenIDM audit service logs the following event topics by default:

Access Event Topics

OpenIDM writes messages at *system boundaries*, that is REST endpoints and the invocation of scheduled tasks in this log. In short, it includes who, what, and output for every access request.

Default file: `openidm/audit/access.audit.json`

Activity Event Topics

OpenIDM logs operations on internal (managed) and external (system) objects to this log.

Entries in the activity log contain identifiers, both for the action that triggered the activity, and for the original caller and the relationships between related actions, on internal and external objects.

Default file: `openidm/audit/activity.audit.json`

Authentication Event Topics

OpenIDM logs the results of authentication operations to this log, including situations and the actions taken on each object, including when and how a user authenticated and related events. The activity log contains additional detail about each authentication action.

Default file: `openidm/audit/authentication.audit.json`

Configuration Event Topics

OpenIDM logs the changes to the configuration in this log. The configuration log includes the "before" and "after" settings for each configuration item, with timestamps.

Default file: `openidm/audit/config.audit.json`

Reconciliation Event Topics

OpenIDM logs the results of reconciliation runs to this log (including situations and the resulting actions taken). The activity log contains details about the actions, where log entries display parent activity identifiers, `recon/reconID`, links, and policy events by datastore.

Default file: `openidm/audit/recon.audit.json`

Synchronization Event Topics

OpenIDM logs the results of automatic synchronization operations (liveSync and implicit synchronization) to this log, including situations and the actions taken on each object, by account. The activity log contains additional detail about each action.

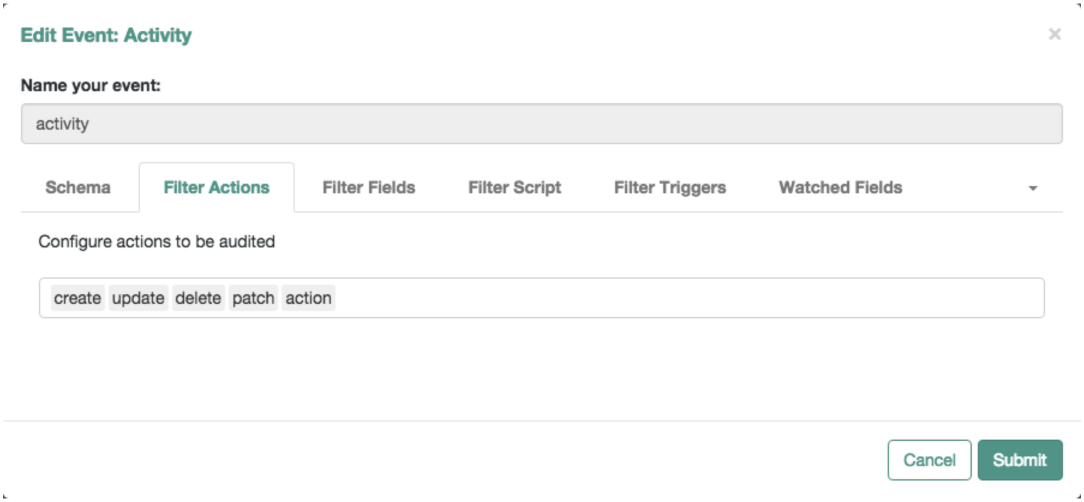
Default file: `openidm/audit/sync.audit.json`

For detailed information about each audit event topic, see "*Audit Log Reference*".

21.4. Event Topics: Filtering

The audit configuration, defined in the `audit.json` file, includes a `filter` parameter that enables you to specify what should be logged, per event type. The information that is logged can be filtered in various ways. The following sections describe the filters that can be applied to each event type.

You can edit these filtering fields in the Admin UI. Click Configure > System Preferences > Audit. Scroll down to Event Topics, and next to the event of your choice, click the pencil icon. You can edit the filtering fields of your choice, as shown in the following figure.



Name your event: activity

Configure actions to be audited

create update delete patch action

If you do not see some of the options in the Admin UI, look for a drop-down arrow on the right side of the window. If your window looks like this figure, you will see the Password Fields tab in the drop-down menu.

21.4.1. Filter Actions: Filtering Audit Entries by Action

The `filter actions` list enables you to specify the actions that are logged, per event type. This filter is essentially a `fields` filter (as described in "Filter Fields: Filtering Audit Entries by Field") that filters log entries by the value of their `actions` field.

The following configuration specifies certain action operations: (create, update, delete, patch, and action). The Audit Service may check filter actions, scripts, and more, when included in the `audit.json` file.

```
"eventTopics" : {
...
    "activity": {
        "filter" : {
            "actions" : [
                "create",
                "update",
                "delete",
                "patch",
                "action"
            ]
        },
        "watchedFields" : [ ],
        "passwordFields" : [
            "password"
        ]
    }
}
```

The list of actions that can be filtered into the log depend on the event type. The following table lists the actions that can be filtered, per event type.

Actions that can be Filtered Per Event Type

Event Type	Actions	Description
Activity and Configuration	`read`	When an object is read by using its identifier. By default, read actions are not logged. Add the `"read"` action to the list of actions to log all read actions. Note that due to the potential result size in the case of read operations on `system/` endpoints, only the read is logged, and not the resource detail. If you really need to log the complete resource detail, add the following line to your `conf/boot/boot.properties` file: `openidm.audit.logFullObjects=true`
	`create`	When an object is created.
	`update`	When an object is updated.
	`delete`	When an object is deleted.
	`patch`	When an object is partially modified. (Activity only.)
	`query`	When a query is performed on an object. By default, query actions are not logged. Add the `"query"` action to the list of actions to log all query actions. Note that, due to the potential result size in the case of query operations on `system/` endpoints, only the query is logged, and not the resource detail. If you really need to log the complete resource detail, add the following line to your `conf/boot/boot.properties` file: `openidm.audit.logFullObjects=true`
	`action`	When an action is performed on an object. (Activity only.)
Reconciliation and Synchronization	`create`	When a target object is created.
	`delete`	When a target object is deleted.
	`update`	When a target object is updated.
	`link`	When a link is created between a source object and an existing target object.
	`unlink`	When a link is removed between a source object and a target object.
	`exception`	When the synchronization situation results in an exception. For more information, see "Synchronization Situations and Actions".
	`ignore`	When the target object is ignored, that is, no action is taken.
Authentication and Access	`-`	No actions can be specified for the authentication or the access log event type.

21.4.2. Filter Fields: Filtering Audit Entries by Field

You can add a list of `filter fields` to the audit configuration, that enables you to filter log entries by specific fields. For example, you might want to restrict the reconciliation or audit log so that only summary information is logged for each reconciliation operation. The following addition to the `audit.json` file specifies that entries are logged in the reconciliation log only if their `entryType` is `start` or `summary`.

```
"eventTopics" : {
    ...
    "activity" : {
        "filter" : {
            "actions" : [
                "create",
                "update",
                "delete",
                "patch",
                "action
            ],
            "fields" : [
                {
                    "name" : "entryType",
                    "values" : [
                        "start",
                        "summary"
                    ]
                }
            ]
        }
    }
    ...
},
...
```

To use nested properties, specify the field name as a JSON pointer. For example, to filter entries according to the value of the `authentication.id`, you would specify the field name as `authentication/id`.

21.4.3. Filter Script: Using a Script to Filter Audit Data

Apart from the audit filtering options described in the previous sections, you can use a JavaScript or Groovy script to specify what is logged in your audit logs. Audit filter scripts are referenced in the audit configuration file (`conf/audit.json`), and can be configured per event type. The following sample configuration references a script named `auditfilter.js`, which is used to limit what is logged in the reconciliation audit log:

```
{
    "eventTopics" : {
        ...
        "recon" : {
            "filter" : {
                "script" : {
                    "type" : "text/javascript",
                    "file" : "auditfilter.js"
                }
            }
        },
        ...
}
```

OpenIDM makes the `request` and `context` objects available to the script. Before writing the audit entry, OpenIDM can access the entry as a `request.content` object. For example, to set up a script to log just the summary entries for mapping managed users in an LDAP data store, you could include the following in the `auditfilter.js` script:

```
(function() {
    return request.content.entryType == 'summary' &&
    request.content.mapping == 'systemLdapAccounts_managedUser'
}());
```

The script must return `true` to include the log entry; `false` to exclude it.

21.4.4. Filter Triggers: Filtering Audit Entries by Trigger

You can add a `filter triggers` list to the audit configuration, that specifies the actions that will be logged for a specific trigger. For example, the following addition to the `audit.json` file specifies that only `create` and `update` actions are logged for in the activity log, for an activity that was triggered by a `recon`.

```
"eventTopics" : {
    "activity" : {
        "filter" : {
            "actions" : [
            ...
            ],
            "triggers" : {
                "recon" : [
                    "create",
                    "update"
                ]
            }
    ...
```

If a trigger is provided, but no actions are specified, nothing is logged for that trigger. If a trigger is omitted, all actions are logged for that trigger. In the current OpenIDM release, only the `recon` trigger is implemented. For a list of reconciliation actions that can be logged, see "Synchronization Actions".

21.4.5. Watched Fields: Defining Fields to Monitor

For the activity log only, you can specify fields whose values are considered particularly important in terms of logging.

The watchedFields parameter, configured in the audit.json file, is not really a filtering mechanism, but enables you to define a list of properties that should be monitored for changes. When the value of one of the properties in this list changes, the change is logged in the activity log, under the column "changedFields". This parameter enables you to have quick access to important changes in the log.

Properties to monitor are listed as values of the watchedFields parameter, separated by commas, for example:

```
"watchedFields" : [ "email", "address" ]
```

You can monitor changes to any field in this way.

21.4.6. Password Fields: Defining a Password Field

Also in the activity log, you can include a passwordFields parameter to specify a list of password properties. This parameter functions much like the watchedFields parameter in that changes to these property values are logged in the activity log, under the column "changedFields". In addition, when a password property is changed, the boolean "passwordChanged" flag is set to true in the activity log. Properties that should be considered as passwords are listed as values of the passwordFields parameter, separated by commas. For example:

```
"passwordFields" : [ "password", "userPassword" ]
```

21.5. Filtering Audit Logs by Policy

By default, the audit.json file for OpenIDM includes the following code snippet for filterPolicies:

```
"filterPolicies" : {
    "value" : {
        "excludeIf" : [
            "/access/http/request/headers/Authorization",
            "/access/http/request/headers/X-OpenIDM-Password",
            "/access/http/request/cookies/session-jwt",
            "/access/http/response/headers/Authorization",
            "/access/http/response/headers/X-OpenIDM-Password"
        ],
        "includeIf" : [ ]
    }
}
```

The excludeIf code snippet lists HTTP access log data that the audit service excludes from log files.

The includeIf directive is available for custom audit event handlers, for items that you want included in log files.

21.6. Configuring an Audit Exception Formatter

The OpenIDM Audit service includes an *exception formatter*, configured in the following snippet of the `audit.json` file:

```
"exceptionFormatter" : {
   "type" : "text/javascript",
   "file" : "bin/defaults/script/audit/stacktraceFormatter.js"
},
```

As shown, you may find the script that defines how the exception formatter works in the `stacktraceFormatter.js` file. That file handles the formatting and display of exceptions written to the audit logger.

21.7. Adjusting Audit Write Behavior

OpenIDM supports buffering to minimize the writes on your systems. To do so, you can configure buffering either in the `project-dir/conf/audit.json` file, or through the Admin UI.

You can configure audit buffering through an event handler. To access an event handler in the Admin UI, click Configure > System Preferences and click on the Audit Tab. When you customize or create an event handler, you can configure the following settings:

Audit Buffering Options

Property	UI Text	Description
`enabled`	True or false	Enables / disables buffering
`autoFlush`	True or false; whether the Audit Service automatically flushes events after writing them to disk	

The following sample code illustrates where you would configure these properties in the `audit.json` file.

```
...
    "eventHandlers" : [
      {
        "config" : {
          ...
          "buffering" : {
            "autoFlush" : false,
            "enabled" : false
          }
        },
...
```

You can set up `autoFlush` when buffering is enabled. OpenIDM then writes data to audit logs asynchronously, while `autoFlush` functionality ensures that the audit service writes data to logs on a regular basis.

If audit data is important, do activate autoFlush. It minimizes the risk of data loss in case of a server crash.

21.8. Purging Obsolete Audit Information

If reconciliation audit volumes grow "excessively" large, any subsequent reconciliations, as well as queries to audit tables, can become "sluggish". In a deployment with limited resources, a lack of disk space can affect system performance.

You might already have restricted what is logged in your audit logs by setting up filters, as described in "Event Topics: Filtering". You can also use specific queries to purge reconciliation audit logs, or you can purge reconciliation audit entries older than a specific date, using timestamps.

OpenIDM includes a sample purge script, autoPurgeRecon.js in the bin/defaults/script/audit directory. This script purges reconciliation audit log entries only from the internal repository. It does not purge data from the corresponding JSON files or external repositories.

To purge reconciliation audit logs on a regular basis, you must set up a schedule. A sample schedule is provided in the schedule-autoPurgeAuditRecon.json file (in the openidm/samples/schedules subdirectory). You can change that schedule as required, and copy the file to the conf/ directory of your project, in order for it to take effect.

The sample purge schedule file is as follows:

```
{
   "enabled" : false,
   "type" : "cron",
   "schedule" : "0 0 */12 * * ?",
   "persisted" : true,
   "misfirePolicy" : "doNothing",
   "invokeService" : "script",
   "invokeContext" : {
      "script" : {
         "type" : "text/javascript",
         "file" : "audit/autoPurgeAuditRecon.js",
         "input" : {
            "mappings" : [ "%" ],
            "purgeType" : "purgeByNumOfReconsToKeep",
            "numOfRecons" : 1,
            "intervalUnit" : "minutes",
            "intervalValue" : 1
         }
      }
   }
}
```

For information about the schedule-related properties in this file, see "Scheduling Synchronization".

Beyond scheduling, the following parameters are of interest for purging the reconciliation audit logs:

input

Input information. The parameters below specify different kinds of input.

mappings

An array of mappings to prune. Each element in the array can be either a string or an object.

Strings must contain the mapping(s) name and can use "%" as a wild card value that will be used in a LIKE condition.

Objects provide the ability to specify mapping(s) to include/exclude and must be of the form:

```
{
      "include" : "mapping1",
      "exclude" : "mapping2"
      ...
}
```

purgeType

The type of purge to perform. Can be set to one of the following values:

purgeByNumOfReconsToKeep

Uses the `deleteFromAuditReconByNumOf` function and the `numOfRecons` config variable.

purgeByExpired

Uses the `deleteFromAuditReconByExpired` function and the config variables `intervalUnit` and `intervalValue`.

num-of-recons

The number of recon summary entries to keep for a given mapping, including all child entries.

intervalUnit

The type of time interval when using `purgeByExpired`. Acceptable values include: `minutes`, `hours`, or `days`.

intervalValue

The value of the time interval when using `purgeByExpired`. Set to an integer value.

21.8.1. Configuring Audit Log Rotation

The file-based audit event handlers enable you to rotate audit log files, either automatically, based on a set of criteria, or by using a REST call.

To configure automatic log file rotation, set the following properties in your project's `audit.json` file:

```
{
    "class" : "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
    "config" : {
        "fileRotation" : {
            "rotationEnabled" : true,
            "maxFileSize" : 0,
            "rotationFilePrefix" : "",
            "rotationTimes" : [ ],
            "rotationFileSuffix" : "",
            "rotationInterval" : ""
},
```

The file rotation properties are described in "JSON Audit Event Handler `config` Properties".

If you have enabled file rotation (`"rotationEnabled" : true`), you can rotate the JSON log files manually for a specific audit event topic, over REST. The following command saves the current access log file with a date and time stamp, then starts logging to a new file with the same base name.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/audit/access?handler=json&_action=rotate"
{
  "status": "OK"
}
```

If the command is successful, you will see two `access.audit.json` files in the `openidm/audit` directory, for example:

```
access.audit.json   access.audit.json-2016.10.12-17.54.41
```

The file with the extension (`2016.10.12-17.54.41`) indicates that audit logging to this file ended on October 12, 2016, at 5:54:41 pm.

To configure log rotation in the Admin UI, click Configure > System Preferences > Audit, and edit the JSON audit event handler (or the CSV audit event handler if you are logging to CSV). You can set all the log rotation properties on this screen.

21.8.2. Configuring Audit Log File Retention

Log file retention specifies how long audit files remain on disk before they are automatically deleted.

To configure log file retention, set the following properties in your project's `audit.json` file:

```
"fileRetention" : {
    "maxNumberOfHistoryFiles" : 100,
    "maxDiskSpaceToUse" : 1000,
    "minFreeSpaceRequired" : 10
},
```

The file retention properties are described in "JSON Audit Event Handler `config` Properties".

To configure log file retention in the Admin UI, click Configure > System Preferences > Audit, and edit the JSON audit event handler (or the CSV audit event handler if you are logging to CSV). You can set all the log retention properties on this screen.

21.9. Querying Audit Logs Over REST

Regardless of where audit events are stored, they are accessible over REST on the `/audit` endpoint. The following sections describe how to query the reconciliation, activity and sync logs over REST. These instructions can be applied to all the other log types.

> **Note**
>
> Queries on the audit endpoint must use `queryFilter` syntax. Predefined queries are not supported. For more information, see "Constructing Queries".

21.9.1. Querying the Reconciliation Audit Log

With the default audit configuration, reconciliation operations are logged in the file `/path/to/openidm/audit/recon.audit.json`, and in the repository. You can read and query the reconciliation audit logs over the REST interface, as outlined in the following examples.

To return all reconciliation operations logged in the audit log, query the `audit/recon` endpoint, as follows:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/audit/recon?_queryFilter=true"
```

The following code extract shows the reconciliation audit log after the first reconciliation operation in Sample 1. The output has been truncated for legibility.

```

  "result": [
    {
      "_id": "b9f1a555-ce7c-4453-ba1b-adfd3beaca75-485",
      "_rev": "1",
      "transactionId": "b9f1a555-ce7c-4453-ba1b-adfd3beaca75-480",
      "timestamp": "2016-10-12T15:23:21.613Z",
      "eventName": "recon",
      "userId": "openidm-admin",
      "exception": null,
      "linkQualifier": null,
      "mapping": "systemXmlfileAccounts_managedUser",
      "message": "Reconciliation initiated by openidm-admin",
      "sourceObjectId": null,
      "targetObjectId": null,
      "reconciling": null,
      "ambiguousTargetObjectIds": null,
      "reconAction": "recon",
      "entryType": "start",
      "reconId": "b9f1a555-ce7c-4453-ba1b-adfd3beaca75-480"
    },
...
    {
      "_id": "b9f1a555-ce7c-4453-ba1b-adfd3beaca75-496",
      "_rev": "1",
      "transactionId": "b9f1a555-ce7c-4453-ba1b-adfd3beaca75-480",
      "timestamp": "2016-10-12T15:23:21.981Z",
      "eventName": "recon",
      "userId": "openidm-admin",
      "exception": null,
      "linkQualifier": null,
      "mapping": "systemXmlfileAccounts_managedUser",
      "message": "SOURCE_IGNORED: 0 UNASSIGNED: 0 AMBIGUOUS: 0 CONFIRMED: 0 FOUND_ALREADY_LINKED: 0
                 UNQUALIFIED: 0 ABSENT: 2 TARGET_IGNORED: 0 SOURCE_MISSING: 0 MISSING: 0 FOUND: 0 ",
      "messageDetail": {
        "_id": "b9f1a555-ce7c-4453-ba1b-adfd3beaca75-480",
        "mapping": "systemXmlfileAccounts_managedUser",
        "state": "SUCCESS",
        "stage": "COMPLETED_SUCCESS",
        "stageDescription": "reconciliation completed.",
        "progress": {
          "source": {
            "existing": {
              "processed": 2,
              "total": "2"
            }
          },
          ...
}
```

Most of the fields in the reconciliation audit log are self-explanatory. Each distinct reconciliation operation is identified by its `reconId`. Each entry in the log is identified by a unique `_id`. The first log entry indicates the status for the complete reconciliation operation. Successive entries indicate the status for each entry affected by the reconciliation.

To obtain information about a specific log entry, include its entry `_id` in the URL. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/audit/recon/414a4921-5d9d-4398-bf86-7d5312a9f5d1-146"
```

The following sample output shows the results of a read operation on a specific reconciliation audit entry. The entry shows the creation of bjensen's account in the managed user repository, as the result of a reconciliation operation.

```
{
   "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-146",
   "_rev" : "1",
   "transactionId" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-135",
   "timestamp" : "2015-11-23T00:18:34.711Z",
   "eventName" : "recon",
   "userId" : "openidm-admin",
   "action" : "CREATE",
   "exception" : null,
   "linkQualifier" : "default",
   "mapping" : "systemXmlfileAccounts_managedUser",
   "message" : null,
   "situation" : "ABSENT",
   "sourceObjectId" : "system/xmlfile/account/scarter",
   "status" : "SUCCESS",
   "targetObjectId" : "managed/user/scarter",
   "reconciling" : "source",
   "ambiguousTargetObjectIds" : "",
   "entryType" : "entry",
   "reconId" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-135"
}
```

To obtain information for a specific reconciliation operation, include the `reconId` in the query. You can filter the log so that the query returns only the fields you want to see, by adding the `_fields` parameter.

The following query returns the `"mapping"`, `"timestamp"`, and `"entryType"` fields for a specific reconciliation operation.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/audit/recon?_queryFilter=/reconId+eq+"4261227f-1d44-4042-ba7e
-1dcbc6ac96b8"&_fields=mapping,timestamp,entryType'
   {
  "result" : [ {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-148",
    "_rev" : "1",
    "mapping" : "systemXmlfileAccounts_managedUser",
    "timestamp" : "2015-11-23T00:18:34.732Z",
    "entryType" : "summary"
  }, {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-146",
    "_rev" : "1",
    "mapping" : "systemXmlfileAccounts_managedUser",
    "timestamp" : "2015-11-23T00:18:34.711Z",
    "entryType" : "entry"
  }, {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-147",
    "_rev" : "1",
    "mapping" : "systemXmlfileAccounts_managedUser",
    "timestamp" : "2015-11-23T00:18:34.711Z",
    "entryType" : "entry"
  }, {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-139",
    "_rev" : "1",
    "mapping" : "systemXmlfileAccounts_managedUser",
    "timestamp" : "2015-11-23T00:18:34.432Z",
    "entryType" : "start"
  } ],
  "resultCount" : 4,
  "pagedResultsCookie" : null,
  "totalPagedResultsPolicy" : "NONE",
  "totalPagedResults" : -1,
  "remainingPagedResults" : -1
}
```

To query the reconciliation audit log for a particular reconciliation situation, include the `reconId` and the `situation` in the query. For example, the following query returns all ABSENT entries that were found during the specified reconciliation operation:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/audit/recon?_queryFilter=/reconId+eq+"414a4921-5d9d-4398-bf86-7d5312a9f5d1
-135"+and+situation+eq+"ABSENT"'
   {
  "result" : [ {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-146",
    "_rev" : "1",
    "situation" : "ABSENT",
    "reconId" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-135",
    "transactionId" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-135",
    "timestamp" : "2015-11-23T00:18:34.711Z",
    "eventName" : "recon",
    "userId" : "openidm-admin",
    "action" : "CREATE",
    "exception" : null,
    "linkQualifier" : "default",
    "mapping" : "systemXmlfileAccounts_managedUser",
    "message" : null,
    "sourceObjectId" : "system/xmlfile/account/scarter",
    "status" : "SUCCESS",
    "targetObjectId" : "managed/user/scarter",
    "reconciling" : "source",
    "ambiguousTargetObjectIds" : "",
    "entryType" : "entry"
  }, {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-147",
    "_rev" : "1",
    "situation" : "ABSENT",
    "reconId" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-135",
    "transactionId" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-135",
    "timestamp" : "2015-11-23T00:18:34.711Z",
    "eventName" : "recon",
    "userId" : "openidm-admin",
    "action" : "CREATE",
    "exception" : null,
    "linkQualifier" : "default",
    "mapping" : "systemXmlfileAccounts_managedUser",
    "message" : null,
    "sourceObjectId" : "system/xmlfile/account/bjensen",
    "status" : "SUCCESS",
    "targetObjectId" : "managed/user/bjensen",
    "reconciling" : "source",
    "ambiguousTargetObjectIds" : "",
    "entryType" : "entry"
  } ],
  "resultCount" : 2,
  "pagedResultsCookie" : null,
  "totalPagedResultsPolicy" : "NONE",
  "totalPagedResults" : -1,
  "remainingPagedResults" : -1
}
```

21.9.2. Querying the Activity Audit Log

The activity logs track all operations on internal (managed) and external (system) objects. Entries in the activity log contain identifiers for the reconciliation or synchronization action that triggered an activity, and for the original caller and the relationships between related actions.

You can access the activity logs over REST with the following call:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/audit/activity?_queryFilter=true"
```

The following extract of the activity log shows one entry that created user bjensen.

```
  }, {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-145",
    "_rev" : "1",
    "transactionId" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-135",
    "timestamp" : "2015-11-23T00:18:34.674Z",
    "eventName" : "activity",
    "userId" : "openidm-admin",
    "runAs" : "openidm-admin",
    "operation" : "CREATE",
    "before" : null,
    "after" : "{ \"mail\": \"bjensen@example.com\", \"givenName\": \"Barbara\", \"sn\": \"Jensen\",
      \"description\": \"Created By XML1\", \"_id\": \"bjensen\", \"userName\": \"bjensen@example.com\",
      \"password\": { \"$crypto\": { \"value\": { \"iv\": \"KHjYJYacmk4UrXzfoTDaSQ==\", \"data\":
      \"o0Lq5HYqgJPSrKSD4AXYsA==\", \"cipher\": \"AES/CBC/PKCS5Padding\", \"key\": \"openidm-sym-default
\" },
      \"type\": \"x-simple-encryption\" } }, \"telephoneNumber\": \"1234567\", \"accountStatus\": \"active
\",
      \"effectiveRoles\": null, \"effectiveAssignments\": [  ], \"_rev\": \"1\" }",
    "changedFields" : [ ],
    "revision" : "1",
    "message" : "create",
    "objectId" : "managed/user/bjensen",
    "passwordChanged" : true,
    "status" : "SUCCESS"
  } ],
...
```

To return the activity information for a specific action, include the _id of the action in the URL, for example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/audit/activity/414a4921-5d9d-4398-bf86-7d5312a9f5d1-145'
```

Each action in the activity log has a transactionId that is the same as the transactionId that was assigned to the incoming or initiating request. So, for example, if an HTTP request invokes a script that changes a user's password, the HTTP request is assigned a transactionId. The action taken by the script is assigned the same transactionId, which enables you to track the complete set of changes resulting from a single action. You can query the activity log for all actions that resulted from a specific transaction, by including the transactionId in the query.

The following command returns all actions in the activity log that happened as a result of a reconciliation, with a specific transactionId. The results of the query are restricted to only the objectId and the resourceOperation. You can see from the output that the reconciliation with this transactionId resulted in two CREATEs and two UPDATEs in the managed repository.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/audit/activity?_queryFilter=/transactionId+eq+"414a4921-5d9d-4398-bf86
-7d5312a9f5d1-135"&_fields=objectId,operation'
```

The following sample output shows the result of a query that created users scarter and bjensen.

```
{
  "result" : [ {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-144",
    "_rev" : "1",
    "objectId" : "managed/user/scarter",
    "operation" : "CREATE"
  }, {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-145",
    "_rev" : "1",
    "objectId" : "managed/user/bjensen",
    "operation" : "CREATE"
  } ],
  "resultCount" : 2,
  "pagedResultsCookie" : null,
  "totalPagedResultsPolicy" : "NONE",
  "totalPagedResults" : -1,
  "remainingPagedResults" : -1
}
```

21.9.3. Querying the Synchronization Audit Log

LiveSync and implicit sync operations are logged in the file /path/to/openidm/audit/sync.audit.json and in the repository. You can read the synchronization audit logs over the REST interface, as outlined in the following examples.

To return all operations logged in the synchronization audit log, query the `audit/sync` endpoint, as follows:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/audit/sync?_queryFilter=true"
{
  "result" : [ {
    "_id" : "53709f21-5b83-4ea0-ac35-9af39c3090cf-95",
    "_rev" : "1",
    "transactionId" : "53709f21-5b83-4ea0-ac35-9af39c3090cf-85",
    "timestamp" : "2015-11-23T05:07:39.376Z",
    "eventName" : "sync",
    "userId" : "openidm-admin",
    "action" : "UPDATE",
    "exception" : null,
    "linkQualifier" : "default",
    "mapping" : "managedUser_systemLdapAccounts",
    "message" : null,
    "situation" : "CONFIRMED",
    "sourceObjectId" : "managed/user/128e0e85-5a07-4e72-bfc8-4d9500a027ce",
    "status" : "SUCCESS",
    "targetObjectId" : "uid=jdoe,ou=People,dc=example,dc=com"
  },
 {
...
```

Most of the fields in the synchronization audit log are self-explanatory. Each entry in the log synchronization operation is identified by a unique `_id`. Each *synchronization operation* is identified with a `transactionId`. The same base `transactionId` is assigned to the incoming or initiating request - so if a modification to a user entry triggers an implicit synchronization operation, both the sync operation and the original change operation have the same `transactionId`. You can query the sync log for all actions that resulted from a specific transaction, by including the `transactionId` in the query.

To obtain information on a specific sync audit log entry, include its entry `_id` in the URL. For example:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/audit/sync/53709f21-5b83-4ea0-ac35-9af39c3090cf-95"
{
  "_id" : "53709f21-5b83-4ea0-ac35-9af39c3090cf-95",
  "_rev" : "1",
  "transactionId" : "53709f21-5b83-4ea0-ac35-9af39c3090cf-85",
  "timestamp" : "2015-11-23T05:07:39.376Z",
  "eventName" : "sync",
  "userId" : "openidm-admin",
  "action" : "UPDATE",
  "exception" : null,
  "linkQualifier" : "default",
  "mapping" : "managedUser_systemLdapAccounts",
  "message" : null,
  "situation" : "CONFIRMED",
  "sourceObjectId" : "managed/user/128e0e85-5a07-4e72-bfc8-4d9500a027ce",
  "status" : "SUCCESS",
  "targetObjectId" : "uid=jdoe,ou=People,dc=example,dc=com"
}
```

21.9.4. Querying the Authentication Audit Log

The authentication log includes details of all successful and failed authentication attempts. The output may be long. The output that follows is one excerpt from 114 entries. To obtain the complete audit log over REST, use the following query:

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/audit/authentication?_queryFilter=true"
{
  "result" : [ {
    "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-5",
    "_rev" : "1",
    "context" : {
      "id" : "anonymous",
      "component" : "repo/internal/user",
      "roles" : [ "openidm-reg" ],
      "ipAddress" : "127.0.0.1"
    },
    "entries" : [ {
      "moduleId" : "IDMAuthModuleWrapper",
      "result" : "FAILED",
      "reason" : { },
      "info" : { }
    }, {
      "moduleId" : "IDMAuthModuleWrapper",
      "result" : "SUCCESSFUL",
      "info" : {
      "org.forgerock.authentication.principal" : "anonymous"
   }
  } ],
  "principal" : [ "anonymous" ],
  "result" : "SUCCESSFUL",
  "userId" : "anonymous",
  "transactionId" : "be858917-764c-4b05-8a6b-ee91cfd8c7e7",
  "timestamp" : "2015-11-23T00:18:10.231Z",
  "eventName" : "authentication",
  "trackingIds" : [ "ea9e65f1-fd28-4153-abc2-891ccbfd482e" ]
}
...
```

You can filter the results to return only those audit entries that you are interested in. For example, the following query returns all authentication attempts made by a specific user (`user.0`) but displays only the security context and the result of the authentication attempt.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 'http://localhost:8080/openidm/audit/authentication?_queryFilter=/principal+eq+"user.0"&_fields=context
,result'
{
  "result": [
    {
      "context": {
        "id": "e98fdfbe-d436-4e09-b44e-f6727b1e293d",
        "component": "managed/user",
        "roles": [
          "openidm-authorized"
        ],
        "ipAddress": "0:0:0:0:0:0:0:1"
      },
      "result": "SUCCESSFUL"
    },
    {
      "context": {
        "ipAddress": "0:0:0:0:0:0:0:1"
      },
      "result": "FAILED"
    },
    {
      "context": {
        "ipAddress": "0:0:0:0:0:0:0:1"
      },
      "result": "FAILED"
    },
    {
      "context": {
        "id": "e98fdfbe-d436-4e09-b44e-f6727b1e293d",
        "component": "managed/user",
        "roles": [
          "openidm-authorized"
        ],
        "ipAddress": "0:0:0:0:0:0:0:1"
      },
      "result": "SUCCESSFUL"
    },
    {
      "context": {
        "id": "e98fdfbe-d436-4e09-b44e-f6727b1e293d",
        "component": "managed/user",
        "roles": [
          "openidm-authorized"
        ],
        "ipAddress": "0:0:0:0:0:0:0:1"
      },
      "result": "SUCCESSFUL"
    },
    {
      "context": {
        "id": "e98fdfbe-d436-4e09-b44e-f6727b1e293d",
        "component": "managed/user",
        "roles": [
          "openidm-authorized"
        ],
        "ipAddress": "0:0:0:0:0:0:0:1"
      },
      "result": "SUCCESSFUL"
    }
,
...
```

21.9.5. Querying the Configuration Audit Log

This audit log lists changes made to the configuration in the audited OpenIDM server. You can read through the changes in the `config.extension` file in the `openidm/audit` directory.

You can also read the complete audit log over REST with the following query:

```
$ curl \
--header "X-OpenIDM-Username: openidm-admin"
 \
--header "X-OpenIDM-Password: openidm-admin"
 \
--request GET \
"http://localhost:8080/openidm/audit/config?_queryFilter=true"
{
   "result" : [ {
     "_id" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-73",
     "_rev" : "1",
     "operation" : "CREATE",
     "userId" : "openidm-admin",
     "runAs" : "openidm-admin",
     "transactionId" : "414a4921-5d9d-4398-bf86-7d5312a9f5d1-58",
     "revision" : null,
     "timestamp" : "2015-11-23T00:18:17.808Z",
     "objectId" : "ui",
     "eventName" : "CONFIG",
     "before" : "",
     "after" : "{ \"icons\":
     ...
      } ],
   "resultCount" : 3,
   "pagedResultsCookie" : null,
   "totalPagedResultsPolicy" : "NONE",
   "totalPagedResults" : -1,
   "remainingPagedResults" : -1
}
```

The output includes a `"before"` and `"after"` entry, which represents the changes in OpenIDM configuration files.

Chapter 22
Clustering, Failover, and Availability

To ensure high availability of the identity management service, you can deploy multiple OpenIDM instances in a cluster. In a clustered environment, each instance must point to the same external repository. If the database is also clustered, OpenIDM points to the cluster as a single system.

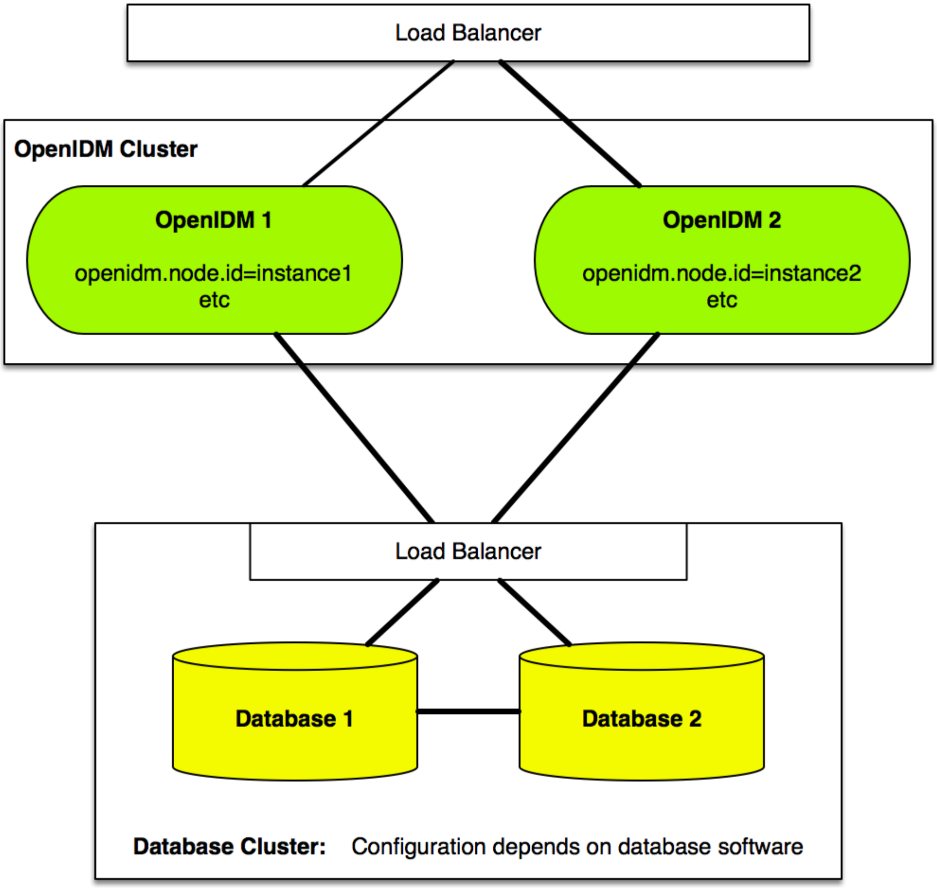
If one instance in a cluster shuts down or fails to check in with the cluster management service, a second instance will detect the failure. For example, if an instance named `instance1` loses connectivity while executing a scheduled task, the cluster manager notifies the scheduler service that `instance1` is not available. The scheduler service then attempts to clean up any jobs that `instance1` was running at that time.

Consistency and concurrency across cluster instances is ensured using multi-version concurrency control (MVCC). MVCC provides consistency because each instance updates only the particular revision of the object that was specified in the update.

All instances in a cluster run simultaneously. When a clustered deployment is configured with a load balancer, the deployment works as an active-active high availability cluster.

This chapter describes the changes required to configure multiple OpenIDM instances in a single cluster. However, it does not specify how you might configure a load balancer. When configured with the scheduler service, the different instances claim jobs in a random order. For more information, see "Managing Scheduled Tasks Across a Cluster".

The following diagram depicts a relatively simple cluster configuration.


Load Balancer
OpenIDM Cluster
OpenIDM 1
openidm.node.id=instance1
etc
OpenIDM 2
openidm.node.id=instance2
etc
Load Balancer
Database 1
Database 2
Database Cluster: Configuration depends on database software

Important

A clustered deployment relies on system heartbeats to assess the cluster state. For the heartbeat mechanism to work, you *must* synchronize the system clocks of all the machines in the cluster using a time synchronization service that runs regularly. The system clocks 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.

22.1. Configuring an OpenIDM Instance as Part of a Cluster

To configure an OpenIDM instance as a part of a clustered deployment, follow these steps:

1. If the server is running, shut it down using the OSGi console:

   ```
   -> shutdown
   ```

2. If you have not already done so, set up a supported repository, as described in "*Installing a Repository For Production*" in the *Installation Guide*.

   Each instance in the cluster must be configured to use the same repository, that is, the database connection configuration file (`datasource.jdbc-default.json`) for each instance must point to the same port number and IP address for the database.

   In "*Installing a Repository For Production*" in the *Installation Guide*, you will see a reference to a data definition language script file. Do not run that script for each instance in the cluster - run it just once to set up the tables required for OpenIDM.

   > **Important**
   >
   > If an instance is *not* participating in the cluster, it must *not* share a repository with nodes that are participating in the cluster. Having non-clustered nodes use the same repository as clustered nodes will result in unexpected behavior.

3. Specify a unique node ID (`openidm.node.id`) for each instance.

   You can specify the node ID in one of the following ways:

   - Set the value of `openidm.node.id` in the `conf/boot/boot.properties` file of the instance, for example:

     ```
     openidm.node.id = node1
     ```

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

     ```
     $ export OPENIDM_OPTS="-Xmx1024m -Xms1024m -Dopenidm.node.id=node1"
     $ ./startup.sh
     Executing ./startup.sh...
     Using OPENIDM_HOME:   /path/to/openidm
     Using PROJECT_HOME:   /path/to/openidm
     Using OPENIDM_OPTS:   -Xmx1024m -Xms1024m -Dopenidm.node.id=node1
     Using LOGGING_CONFIG: -Djava.util.logging.config.file=/path/to/openidm/conf/logging.properties
     Using boot properties at /path/to/openidm/conf/boot/boot
     .properties
     -> OpenIDM version "5.0.0"
     OpenIDM ready
     ```

   You can set any value for the `openidm.node.id`, as long as the value is unique within the cluster. The cluster manager detects unavailable OpenIDM instances by their node ID.

   You *must* set a node ID for each instance, otherwise the instance fails to start. The default `conf/boot/boot.properties` file sets the node ID to `openidm.node.id=node1`.

4. Set the cluster configuration.

   The cluster configuration is defined in the `conf/cluster.json` file of each instance. By default, configuration changes are persisted in the repository so changes that you make in this file apply to all nodes in the cluster.

   The default version of the `cluster.json` file assumes that the cluster management service is enabled:

   ```
   {
     "instanceId" : "&{openidm.node.id}",
     "instanceTimeout" : "30000",
     "instanceRecoveryTimeout" : "30000",
     "instanceCheckInInterval" : "5000",
     "instanceCheckInOffset" : "0",
     "enabled" : true
   }
   ```

   - The `instanceId` is set to the value of each instance's `openidm.node.id` that you set in the previous step.
   - The `instanceTimeout` specifies the length of time (in milliseconds) that a member of the cluster can be "down" before the cluster manager considers that instance to be in recovery mode.

     Recovery mode indicates that the `instanceTimeout` of an OpenIDM instance has expired, and that another OpenIDM instance in the cluster has detected that event.

     The scheduler component of the second OpenIDM instance then moves any incomplete jobs into the queue for the cluster.
   - The `instanceRecoveryTimeout` specifies the time (in milliseconds) that an OpenIDM instance can be in recovery mode before it is considered to be offline.

     This property sets a limit after which other members of the cluster stop trying to access an unavailable OpenIDM instance.
   - The `instanceCheckInInterval` specifies the frequency (in milliseconds) that OpenIDM instances check in with the cluster manager to indicate that they are still online.
   - The `instanceCheckInOffset` specifies an offset (in milliseconds) for the checkin timing, when multiple OpenIDM instances in a cluster are started simultaneously.

     The checkin offset prevents multiple OpenIDM instances from checking in simultaneously, which would strain the cluster manager resource.
   - The `enabled` property specifies whether the cluster management service is enabled when you start OpenIDM. This property is set to `true` by default.

Important

Disabling the cluster manager while clustered nodes are running (by setting the `enabled` property to `false` in an instance's `cluster.json` file), has the following consequences:

- The cluster manager thread that causes instances to *check in* is not activated.
- Nodes in the cluster no longer receive cluster *events*, which are used to broadcast configuration changes when they occur over the REST interface.
- Nodes are unable to detect and attempt to recover failed instances within the cluster.
- Persisted schedules associated with failed instances can not be recovered by other nodes.

5. Optionally, configure your cluster so that each instance reads its configuration only from the files in its `conf/` directory and not from the shared repository. For more information, see "Specifying an Authoritative File-Based Configuration".

6. If your deployment uses scheduled tasks, configure persistent schedules so that jobs and tasks are launched only once across the cluster. For more information, see "Configuring Persistent Schedules".

7. Make sure that each node in the cluster has the same keystore and truststore. You can do this in one of the following ways:

   - When the first OpenIDM instance has been started, copy the initialized keystore (`/path/to/openidm/security/keystore.jceks`) and truststore (`/path/to/openidm/security/truststore`) to all other instances in the cluster.
   - Use a single keystore that is shared between all the nodes. The shared keystore might be on a mounted filesystem, a Hardware Security Module (HSM) or something similar. If you use this method, set the following properties in the `conf/boot/boot.properties` file of each instance to point to the shared keystore:

     ```
     openidm.keystore.location=path/to/keystore
     openidm.truststore.location=path/to/truststore
     ```

     For information on configuring OpenIDM to use an HSM device, see "Configuring a Hardware Security Module (HSM) Device".

8. Start each OpenIDM instance in the cluster.

Important

The OpenIDM audit service logs configuration changes only on the modified instance of OpenIDM. Although configuration changes are persisted in the repository, and thus replicated on other instances by default, those changes are not logged separately for each instance. For more information on the audit service, see "*Logging Audit Information*".

Although configuration changes are persisted by default, changes to workflows and scripts, and extensions to the UI are not. Any changes that you make in these areas must be manually copied to each node in the cluster.

22.1.1. Specifying an Authoritative File-Based Configuration

Each OpenIDM instance includes two properties in its `conf/system.properties` file that determine how configuration changes are handled:

- `openidm.fileinstall.enabled` specifies that OpenIDM reads its configuration from the files in its `conf/` directory.

  This parameter is `true` by default because the following line is commented out:

  ```
  # openidm.fileinstall.enabled=false
  ```

  If you want the file-based configuration to be authoritative, set this property to `true` (or leave the existing line commented out) on every instance in the cluster.

  For information on changing this setting, see "Disabling Automatic Configuration Updates".

- `openidm.config.repo.enabled` specifies that OpenIDM reads its configuration from the repository.

  This parameter is `true` by default because the following line is commented out:

  ```
  # openidm.repo.enabled=false
  ```

  If you want the file-based configuration to be authoritative, set this property to `false` by removing the comment from the existing line on every instance in the cluster.

With this configuration:

- Each node in the cluster does not persist its configuration to the repository.
- Each node has its own version of the configuration in memory only.
- Any changes made to one node's file configuration must be applied manually across all nodes in the cluster for the configuration to be consistent.

  When new nodes are deployed, you must ensure that the configuration is consistent across all nodes.

22.2. Managing Scheduled Tasks Across a Cluster

In a clustered environment, the scheduler service looks for pending jobs and handles them as follows:

- Non-persistent (in-memory) jobs execute on each node in the cluster.
- Persistent scheduled jobs are picked up and executed by a single node in the cluster.

- Jobs that are configured as persistent but *not concurrent* run on only one instance in the cluster. That job will not run again at the scheduled time, on any instance in the cluster, until the current job is complete.

  For example, a reconciliation operation that runs for longer than the time between scheduled intervals will not trigger a duplicate job while it is still running.

OpenIDM instances in a cluster claim jobs in a random order. If one instance fails, the cluster manager automatically reassigns unstarted jobs that were claimed by that failed instance.

For example, if OpenIDM instance A claims a job but does not start it, and then loses connectivity, OpenIDM instance B can claim that job.

In contrast, if OpenIDM instance A claims a job, starts it, and then loses connectivity, other OpenIDM instances in the cluster cannot claim that job. If the failed instance of OpenIDM does not complete the task, the next action depends on the *misfire policy*, defined in the scheduler configuration. For more information, see `misfirePolicy`.

> **Note**
>
> This behavior varies from OpenIDM 2.1.0, in which an unavailable OpenIDM instance would have to reconnect to the cluster to free a job that it had already claimed.

You can override this behavior with an external load balancer.

If a liveSync operation leads to multiple changes, a single OpenIDM instance processes all changes related to that operation.

Because all nodes in a cluster read their configuration from a single repository, you must use an instance's `conf/boot/boot.properties` file to define a specific scheduler configuration for that instance.

By default, instances in a cluster are able to execute persistent schedules. The setting in the `boot.properties` file that governs this behaviour is:

```
openidm.scheduler.execute.persistent.schedules=true
```

To prevent a specific OpenIDM instance from claiming pending jobs, or processing clustered schedules, set `openidm.scheduler.execute.persistent.schedules=false` in the `boot.properties` file of that instance.

22.3. Managing Nodes Over REST

You can manage clusters and individual nodes over the REST interface, at the URL `https://localhost:8443/openidm/cluster/`. The following sample REST commands demonstrate the cluster information that is available over REST.

Displaying the Nodes in the Cluster

The following REST request displays the nodes configured in the cluster, and their status.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/cluster"

{
  "results": [
    {
       "state" : "running",
       "instanceId" : "instance2",
       "startup" : "2015-08-28T12:50:37.209-07:00",
       "shutdown" : ""
    },
    {
       "state" : "running",
       "instanceId" : "instance1",
       "startup" : "2015-08-28T11:33:12.650-07:00",
       "shutdown" : ""
    }
  ]
}
```

Checking the State of an Individual Node

To check the status of a specific node, include its node ID in the URL, for example:

```
$  curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/cluster/instance1"
{
     "state" : "running",
     "instanceId" : "instance1",
     "startup" : "2015-08-28T11:33:12.650-07:00",
     "shutdown" : ""
}
```

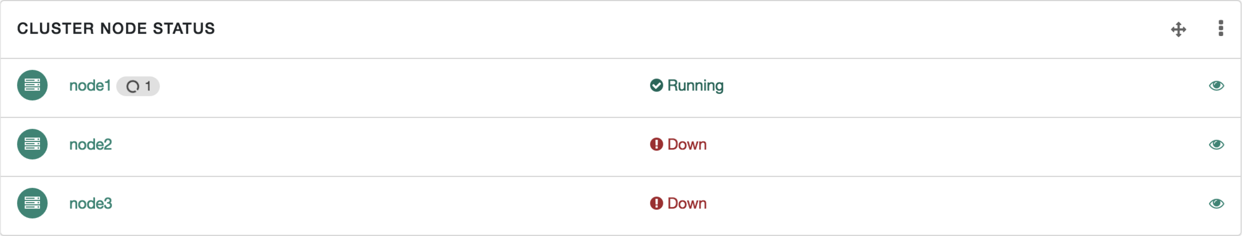
22.4. Managing Nodes Through the Admin UI

The Admin UI provides a status widget that enables you to monitor the activity and status of all nodes in a cluster. To add the widget, click Add Widgets on the Dashboard, then scroll down to Cluster Node Status and click Add.

The cluster node status widget shows the current status, last startup time, and last shutdown time of each node. The widget also indicates whether a node is currently running any tasks.

The following image shows a cluster with three nodes, two of which are down. The first node is currently running one job. To obtain more information on running jobs, click on the job icon next to the node name.

Cluster Node Status Widget

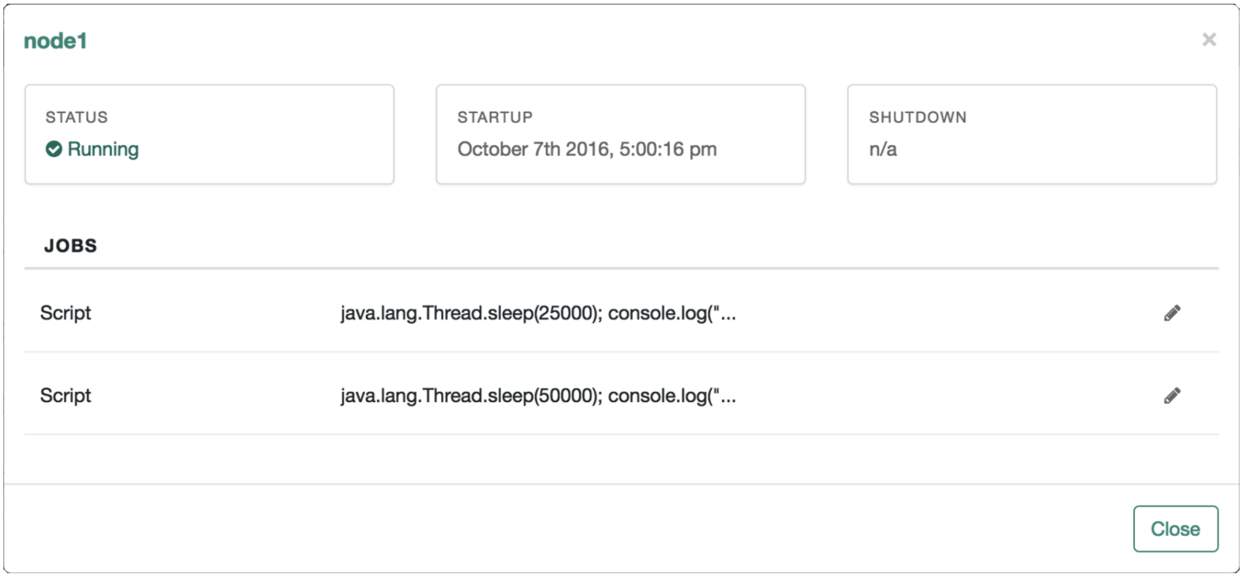


CLUSTER NODE STATUS

node1 ○ 1	✓ Running
node2	❗ Down
node3	❗ Down

To obtain more information on each node, click the view icon to the right of the node. The following image shows the additional detail for node1:

Cluster Node Detail



node1

STATUS	STARTUP	SHUTDOWN
✓ Running	October 7th 2016, 5:00:16 pm	n/a

JOBS

Script	java.lang.Thread.sleep(25000); console.log("...
Script	java.lang.Thread.sleep(50000); console.log("...

The widget can be managed in the same way as any other dashboard widget. For more information, see "Creating and Modifying Dashboards".

Chapter 23
Configuring Outbound Email

This chapter shows you how to configure the outbound email service, so that you can send email through IDM, either by script or through the REST API.

You can also configure the outbound email service in the Admin UI, by clicking Configure > System Preferences > Email. The fields on that screen correspond to what is described in the following sections.

To Set Up Outbound Email

The outbound email service relies on a configuration object to identify the email account that is used to send messages. A sample configuration is provided in `openidm/samples/misc/external.email.json`. To set up the external email service, follow these steps.

1. You do not have to shut down OpenIDM.

   If you are setting up outbound email through the UI, start configuring an outbound email server directly from the noted UI screen.

2. Copy the sample email configuration to the `conf` directory of your project. For example:

   ```
   $ cd /path/to/openidm/
   $ cp samples/misc/external.email.json conf/
   ```

3. Edit `external.email.json` to reflect the account that is used to send messages, for example:

```
{
    "host" : "smtp.gmail.com",
    "port" : 587,
    "debug" : false,
    "auth" : {
        "enable" : true,
        "username" : "admin",
        "password" : "Passw0rd"
    },
    "from" : "admin@example.com",
    "timeout" : 300000,
    "writetimeout" : 300000,
    "connectiontimeout" : 300000,
    "starttls" : {
        "enable" : true
    },
    "ssl" : {
        "enable" : false
    },
    "smtpProperties" : [
        "mail.smtp.ssl.protocols=TLSv1.2",
        "mail.smtps.ssl.protocols=TLSv1.2"
    ],
    "threadPoolSize" : 20
}
```

IDM encrypts the password when you restart the server (or if you configure outgoing email through the Admin UI).

You can specify the following outbound email configuration properties:

`host`

The host name or IP address of the SMTP server. This can be the `localhost`, if the mail server is on the same system as IDM.

`port`

SMTP server port number, such as 25, 465, or 587.

> **Note**
>
> Many SMTP servers require the use of a secure port such as 465 or 587. Many ISPs flag email from port 25 as spam.

`debug`

When set to `true`, this option outputs diagnostic messages from the JavaMail library. Debug mode can be useful if you are having difficulty configuring the external email endpoint with your mail server.

`auth`

The authentication details for the mail account from which emails will be sent.

- `enable`—indicates whether you need login credentials to connect to the SMTP server.

> **Note**
>
> If `"enable" : false`, you can leave the entries for `"username"` and `"password"` empty:
>
> ```
> "enable" : false,
> "username" : "",
> "password" : ""
> ```

- `username`—the account used to connect to the SMTP server.
- `password`—the password used to connect to the SMTP server.

`starttls`

If `"enable" : true`, enables the use of the STARTTLS command (if supported by the server) to switch the connection to a TLS-protected connection before issuing any login commands. If the server does not support STARTTLS, the connection continues without the use of TLS.

`from`

(Optional) Specifies a default `From:` address, that users see when they receive emails from IDM.

`ssl`

Set `"enable" : true` to use SSL to connect, and to use the SSL port by default.

`smtpProperties`

Specifies the SSL protocols that will be enabled for SSL connections. Protocols are specified as a whitespace-separated list. The default protocol is TLSv1.2.

`threadPoolSize`

(Optional) Emails are sent in separate threads managed by a thread pool. This property sets the number of concurrent emails that can be handled at a specific time. The default thread pool size (if none is specified) is `20`.

`connectiontimeout` **(integer, optional)**

The socket connection timeout, in milliseconds. The default connection timeout (if none is specified) is `300000` milliseconds, or 5 minutes. A setting of 0 disables this timeout.

`timeout` **(integer, optional)**

The socket read timeout, in milliseconds. The default read timeout (if none is specified) is `300000` milliseconds, or 5 minutes. A setting of 0 disables this timeout.

`writetimeout` **(integer, optional)**

The socket write timeout, in milliseconds. The default write timeout (if none is specified) is `300000` milliseconds, or 5 minutes. A setting of 0 disables this timeout.

4. Start IDM if it is not running.

5. Check that the email service is enabled and active:

```
-> scr list
...
 [ 130]   org.forgerock.openidm.external.email  enabled
    [  21] [active      ] org.forgerock.openidm.external.email
...
```

23.1. Sending Mail Over REST

Although you are more likely to send mail from a script in production, you can send email using the REST API by sending an HTTP POST to `/openidm/external/email`, to test that your configuration works. You pass the message parameters as part of the POST payload, URL encoding the content as necessary.

The following example sends a test email using the REST API.

```
$ curl \
 --cacert self-signed.crt \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 --data '{
   "from":"openidm@example.com",
   "to":"your_email@example.com",
   "subject":"Test",
   "body":"Test"}' \
 "https://localhost:8443/openidm/external/email?_action=send"
{
 "status": "OK"
}
```

23.2. Sending Mail From a Script

You can send email by using the resource API functions, with the `external/email` context. For more information about these functions, see "Function Reference". In the following example, `params` is an object that contains the POST parameters.

```
var params =  new Object();
params.from = "openidm@example.com";
params.to = "your_email@example.com";
params.cc = "bjensen@example.com,scarter@example.com";
params.subject = "OpenIDM recon report";
params.type = "text/html";
params.body = "<html><body><p>Recon report follows...</p></body></html>";

openidm.action("external/email", "send", params);
```

OpenIDM supports the following POST parameters.

`from`

Sender mail address

`to`

Comma-separated list of recipient mail addresses

`cc`

Optional comma-separated list of copy recipient mail addresses

`bcc`

Optional comma-separated list of blind copy recipient mail addresses

`subject`

Email subject

`body`

Email body text

`type`

Optional MIME type. One of `"text/plain"`, `"text/html"`, or `"text/xml"`.

Chapter 24
Accessing External REST Services

You can access remote REST services by using the `openidm/external/rest` endpoint, or by specifying the `external/rest` resource in your scripts. Note that this service is not intended as a full connector to synchronize or reconcile identity data, but as a way to make dynamic HTTP calls as part of the OpenIDM logic. For more declarative and encapsulated interaction with remote REST services, and for synchronization or reconciliation operations, you should rather use the scripted REST connector.

An external REST call via a script might look something like the following:

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

The `call` parameter specifies the action name to be used for this invocation, and is the standard method signature for the `openidm.action` method.

An external REST call over REST might look something like the following:

```
$ curl \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 --data '{
  "url": "http://www.december.com/html/demo/hello.html",
  "method": "GET",
  "headers": { "custom-header": "custom-header-value" }
  }' \
 "http://localhost:8080/openidm/external/rest?_action=call"
{
  "headers": {
    "Accept-Ranges": [
      "bytes"
    ],
    "Content-Length": [
      "665"
    ],
    "Content-Type": [
      "text/html"
    ],
    "Date": [
      "Thu, 28 Jul 2016 09:13:38 GMT"
    ],
    "ETag": [
      "\"299-4175ff09d1140\""
    ],
    "Last-Modified": [
      "Thu, 29 Jun 2006 17:05:33 GMT"
    ],
    "Server": [
      "Apache"
    ]
  },
  "body": "<!DOCTYPE html PUBLIC \"-//IETF//DTD HTML 2.0//EN\">\r\n
          <html>\r\n
           <head>\r\n
            <title>\r\n   Hello World Demonstration Document\r\n  </title>\r\n
           </head>\r\n
           <body>\r\n
            <h1>\r\n   Hello, World!\r\n  </h1>\r\n
            <p>\r\n   This is a minimal \"hello world\" HTML document. It
             demonstrates the\r\n   basic structure of an HTML file and anchors.\r\n
            </p>\r\n
            <p>\r\n   For more information, see the HTML Station at:
             <a href= \r\n   \"http://www.december.com/html/\">http://www.december.com/html/</a>\r\n
            </p>\r\n
            <hr>\r\n
            <address>\r\n   &copy;
            <a href=\"http://www.december.com/john/\">John December</a>
           (<a\r\n   href=\"mailto:john@december.com\">john@december.com</a>) / 2001-04-06\r\n
            </address>\r\n </body>\r\n</html>\r\n"}
```

HTTP 2xx responses are represented as regular, successful responses to the invocation. All other responses, including redirections, are returned as exceptions, with the HTTP status code in the exception `code`, and the response body in the exception `detail`, within the `content` element.

24.1. Invocation Parameters

The following parameters are passed in the resource API parameters map. These parameters can override the static configuration (if present) on a per-invocation basis.

`url`

The target URL to invoke, in string format.

`method`

The HTTP action to invoke, in string format.

Possible actions include `POST`, `GET`, `PUT`, `DELETE`, and `OPTIONS`.

`headers` **(optional)**

The HTTP headers to set, in a map format from string (*header-name*) to string (*header-value*). For example, `Accept-Language: en-US`.

`contentType` **(optional)**

The media type of the data that is sent, for example `"contentType" : "application/json"`. This parameter is applied only if no `Content-Type` header is included in the request. (If a `Content-Type` header is included, that header takes precedence over this `contentType` parameter.) If no `Content-Type` is provided (in the header or with this parameter), the default content type is `application/json; charset=utf-8`.

`body` **(optional)**

The body or resource representation to send (for PUT and POST operations), in string format.

`base64` **(boolean, optional)**

Indicates that the `body` is base64-encoded, and should be decoded prior to transmission.

`forceWrap` **(boolean, optional)**

Indicates that the response must be wrapped in the headers/body JSON message format, even if the response was JSON and would otherwise have been passed-through unchanged.

`authenticate`

The authentication type, and the details with which to authenticate.

OpenIDM supports the following authentication types:

- `basic` authentication with a username and password, for example:

```
"authenticate" : {
    "type": "basic",
    "user" : "john",
    "password" : "Passw0rd"
}
```

- `bearer` authentication, with an OAuth token instead of a username and password, for example:

```
"authenticate" : {
    "type": "bearer",
    "token" : "ya29.iQDWKpn8AHy09p....."
}
```

If no `authenticate` parameter is specified, no authentication is used.

24.2. Support for Non-JSON Responses

The external REST service supports any arbitrary payload (currently in stringified format). If the response is anything other than JSON, a JSON message object is returned:

- For text-compatible (non-JSON) content, OpenIDM returns a JSON object similar to the following:

```
{
    "headers": { "Content-Type": ["..."] },
    "body": "..."
}
```

- Content that is not text-compatible (such as JPEGs) is base64-encoded in the response `body` and returned as follows:

```
{
    "headers": { "Content-Type": ["..."] },
    "body": "...",
    "base64": true
}
```

Note

If the response format is JSON, the raw JSON response is returned. If you want to inspect the response headers, set `forceWrap` to `true` in your request. This setting returns a JSON message object with `headers` and `body`, similar to the object returned for text-compatible content.

24.3. Setting the TLS Version

By default, Transport Layer Security (TLS) connections made via the external REST service use TLS version 1.2. In rare cases, you might need to specify a different TLS version, for example, if you are connecting to a legacy system that supports an old version of TLS that is not accommodated by the backward-compatibility mode of your Java client. If you need to specify that the external REST service use a different TLS version, uncomment the `openidm.external.rest.tls.version` property towards the end of your project's `conf/boot/boot.properties` file and set its value, for example:

```
openidm.external.rest.tls.version=TLSv1.1
```

Valid versions for this parameter include `TLSv1.1` and `TLSv1.2`.

24.4. Configuring the External REST Service

In addition to the TLS version, described in "Setting the TLS Version", you can configure several properties of the external REST service.

A sample configuration file that lists these properties (with their default values where applicable) is provided in `/path/to/openidm/samples/misc/external.rest.json`. To change any of the default settings, copy this file to your project's `conf` directory and edit the values. The sample file has the following configuration:

```
{
  "socketTimeout" : "10 s",
  "connectionTimeout" : "10 s",
  "reuseConnections" : "true",
  "retryRequests" : "true",
  "maxConnections" : "64",
  "tlsVersion": "&{openidm.external.rest.tls.version}",
  "hostnameVerifier": "STRICT",
  "proxySystem" : "false",
  "proxy" : {
    "proxyUri" : "",
    "userName" : "",
    "password" : ""
  }
}
```

Note that all these properties must be passed in as strings because the configuration supports property substitution. For information, see "Using Property Value Substitution In the Configuration".

`socketTimeout` **(string)**

> The TCP socket timeout, in seconds, when waiting for HTTP responses. The default timeout is 10 seconds.

`connectionTimeout` **(string)**

> The TCP connection timeout for new HTTP connections, in seconds. The default timeout is 10 seconds.

`reuseConnections` **(string)**

> Specifies whether HTTP connections should be kept alive and reused for additional requests. By default, connections will be reused if possible.

`retryRequests` **(string)**

> Specifies whether requests should be retried if a failure is detected. By default requests will be retried.

`maxConnections` **(string)**

> The maximum number of connections that should be pooled by the HTTP client. At most `64` connections will be pooled by default.

`tlsVersion` **(string)**

> The TLS version that should be used for connections. For more information, see "Setting the TLS Version".

`hostnameVerifier` **(string)**

> Specifies whether the external REST service should check that the hostname to which an SSL client has connected is allowed by the certificate that is presented by the server. By default, with

`"hostnameVerifier": "STRICT"`, hostnames are validated. To specify that the REST service should trust all SSL certificates, set `"hostnameVerifier": "ALLOW_ALL"`.

`proxySystem` (string)

Specifies whether an outbound proxy system should be used. There is no outbound proxy by default. If this value is set to `true`, specify the connection details to the proxy server with the `proxyUri`, `userName`, and `password` properties.

Chapter 25
Deployment Best Practices

This chapter lists points to check when implementing an identity management solution with OpenIDM.

25.1. Implementation Phases

Any identity management project should follow a set of well defined phases, where each phase defines discrete deliverables. The phases take the project from initiation to finally going live with a tested solution.

25.1.1. Initiation

The project's initiation phase involves identifying and gathering project background, requirements, and goals at a high level. The deliverable for this phase is a statement of work or a mission statement.

25.1.2. Definition

In the definition phase, you gather more detailed information on existing systems, determine how to integrate, describe account schemas, procedures, and other information relevant to the OpenIDM deployment. The deliverable for this phase is one or more documents that define detailed requirements for the project, and that cover project definition, the business case, use cases to solve, and functional specifications.

The definition phase should capture at least the following.

User Administration and Management

> Procedures for managing users and accounts, who manages users, what processes look like for joiners, movers and leavers, and what is required of OpenIDM to manage users

Password Management and Password Synchronization

> Procedures for managing account passwords, password policies, who manages passwords, and what is required of OpenIDM to manage passwords

Security Policy

> What security policies defines for users, accounts, passwords, and access control

Target Systems

Target systems and resources with which OpenIDM must integrate. Information such as schema, attribute mappings and attribute transformation flow, credentials and other integration specific information.

Entitlement Management

Procedures to manage user access to resources, individual entitlements, grouping provisioning activities into encapsulated concepts such as roles and groups

Synchronization and Data Flow

Detailed outlines showing how identity information flows from authoritative sources to target systems, attribute transformations required

Interfaces

How to secure the REST, user and file-based interfaces, and to secure the communication protocols involved

Auditing and Reporting

Procedures for auditing and reporting, including who takes responsibility for auditing and reporting, and what information is aggregated and reported. Characteristics of reporting engines provided, or definition of the reporting engine to be integrated.

Technical Requirements

Other technical requirements for the solution such as how to maintain the solution in terms of monitoring, patch management, availability, backup, restore and recovery process. This includes any other components leveraged such as a ConnectorServer and plug-ins for password synchronization on Active Directory, or OpenDJ.

25.1.3. Design

This phase focuses on solution design including on OpenIDM and other components. The deliverables for this phase are the architecture and design documents, and also success criteria with detailed descriptions and test cases to verify when project goals have been met.

25.1.4. Configure and Test

This phase configures and tests the solution prior to moving the solution into production.

Configure a Connector

Most deployments include a connection to one or more remote data stores. You should first define all properties for your connector configuration as described in "Supported Connectors".

If you have custom attributes, you can add them as described in: "Adding Attributes to Connector Configurations".

Test Communication to Remote Data Stores

You can then test communication with each remote data store with appropriate REST calls, such as those described in: "Checking the Status of External Systems Over REST". When your tests succeed, you can have confidence in the way you configured OpenIDM to communicate with your remote data stores.

Set Up a Mapping

You can now set up a mapping between data stores. "*Synchronizing Data Between Resources*" includes an extensive discussion of how you can customize a mapping in the `sync.json` file.

Once complete, you should set up associated custom configuration files in a directory *outside* of the OpenIDM installation directory (in other words, outside the `/path/to/openidm` directory tree).

25.1.5. Production

This phase deploys the solution into production until an application steady state is reached and maintenance routines and procedures can be applied.

Chapter 26
Troubleshooting

When things are not working check this chapter for tips and answers.

26.1. Server Stopped in Background

When you start OpenIDM in the background without having disabled the text console, the job can stop immediately after startup.

```
$ ./startup.sh &
[2] 346
$ ./startup.sh
Using OPENIDM_HOME:   /path/to/openidm
Using OPENIDM_OPTS:   -Xmx1024m -Xms1024m
Using LOGGING_CONFIG:
 -Djava.util.logging.config.file=/path/to/openidm/conf/logging.properties
Using boot properties at /path/to/openidm/conf/boot/boot.properties
->

[2]+  Stopped                 ./startup.sh
```

To resolve this problem, make sure you remove `openidm/bundle/org.apache.felix.shell.tui-1.4.1.jar` before starting OpenIDM, and also remove Felix cache files in `openidm/felix-cache/`.

26.2. The scr list Command Shows Sync Service As Unsatisfied

You might encounter this message in the logs.

```

WARNING: Loading configuration file /path/to/openidm/conf/sync.json failed
org.forgerock.openidm.config.InvalidException:
 Configuration for org.forgerock.openidm.sync could not be parsed and may not
     be valid JSON : Unexpected character ('}' (code 125)): expected a value
     at [Source: java.io.StringReader@3951f910; line: 24, column: 6]
 at org.forgerock.openidm.config.crypto.ConfigCrypto.parse...
 at org.forgerock.openidm.config.crypto.ConfigCrypto.encrypt...
 at org.forgerock.openidm.config.installer.JSONConfigInstaller.setConfig...
```

This indicates a syntax error in `openidm/conf/sync.json`. After fixing your configuration, change to the `/path/to/openidm/` directory, and use the **cli.sh validate** command to check that your configuration files are valid.

```
$ cd /path/to/openidm ; ./cli.sh validate
Using boot properties at /path/to/openidm/conf/boot/boot.properties
...................................................................
[Validating] Load JSON configuration files from:
[Validating]  /path/to/openidm/conf
[Validating] audit.json ................................. SUCCESS
[Validating] authentication.json ........................ SUCCESS
[Validating] managed.json ............................... SUCCESS
[Validating] provisioner.openicf-xml.json ............... SUCCESS
[Validating] repo.orientdb.json ......................... SUCCESS
[Validating] router.json ................................ SUCCESS
[Validating] scheduler-reconcile_systemXmlAccounts_managedUser.json  SUCCESS
[Validating] sync.json .................................. SUCCESS
```

26.3. JSON Parsing Error

You might encounter this error message in the logs.

```
"Configuration for org.forgerock.openidm.provisioner.openicf could not be
 parsed and may not be valid JSON : Unexpected character ('}' (code 125)):
 was expecting double-quote to start field name"
```

The error message usually indicates the precise point where the JSON file has the syntax problem. The error above was caused by an extra comma in the JSON file, `{"attributeName":{},{},}`. The second comma is redundant.

The situation usually results in the service that the specific JSON file configures being left in the `unsatisfied` state.

After fixing your configuration, change to the `/path/to/openidm/` directory, and use the **cli.sh validate** command to check that your configuration files are valid.

26.4. System Not Available

OpenIDM throws the following error as a result of a reconciliation where the source systems configuration can not be found.

```
{
    "error": "Conflict",
    "description": "Internal Server Error:
        org.forgerock.openidm.sync.SynchronizationException:
        org.forgerock.openidm.objset.ObjectSetException:
        System: system/HR/account is not available.:
        org.forgerock.openidm.objset.ObjectSetException:
        System: system/HR/account is not available.:
        System: system/HR/account is not available."
}
```

This error occurs when the `"name"` property value in `provisioner.resource.json` is changed from `HR` to something else.

The same error occurs when a provisioner configuration fails to load due to misconfiguration, or when the path to the data file for a CSV or XML connector is incorrectly set.

26.5. Bad Connector Host Reference in Provisioner Configuration

You might see the following error when a provisioner configuration loads.

```
Wait for meta data for config org.forgerock.openidm.provisioner.openicf-scriptedsql
```

In this case the configuration fails to load because information is missing. One possible cause is an incorrect value for `connectorHostRef` in the provisioner configuration file.

For local Java connector servers, the following rules apply.

- If the connector .jar is installed as a bundle under `openidm/bundle`, then the value must be `"connectorHostRef" : "osgi:service/org.forgerock.openicf.framework.api.osgi.ConnectorManager",`.
- If the connector .jar is installed as a connector under `openidm/connectors`, then the value must be `"connectorHostRef" : "#LOCAL",`.

26.6. Missing Name Attribute

In this case, the situation in the audit recon log shows "NULL".

A missing name attribute error, followed by an `IllegalArgumentException`, points to misconfiguration of the correlation rule, with the correlation query pointing to the external system. Such queries usually reference the "name" field which, if empty, leads to the error below.

```
Jan 20, 2012 1:59:58 PM
 org.forgerock.openidm.provisioner.openicf.commons.AttributeInfoHelper build
SEVERE: Failed to build name attribute out of [null]
Jan 20, 2012 1:59:58 PM
 org.forgerock.openidm.provisioner.openicf.impl.OpenICFProvisionerService query
SEVERE: Operation [query, system/ad/account] failed with Exception on system
 object: java.lang.IllegalArgumentException: Attribute value must be an
 instance of String.
Jan 20, 2012 1:59:58 PM org.forgerock.openidm.router.JsonResourceRouterService
 handle
WARNING: JSON resource exception
org.forgerock.json.resource.JsonResourceException: IllegalArgumentException
 at org.forgerock.openidm.provisioner....OpenICFProvisionerService.query...
 at org.forgerock.openidm.provisioner.....OpenICFProvisionerService.handle...
 at org.forgerock.openidm.provisioner.impl.SystemObjectSetService.handle...
 at org.forgerock.json.resource.JsonResourceRouter.handle...
```

Check your `correlationQuery`. Another symptom of a broken correlation query is that the audit recon log shows a situation of "NULL", and no onCreate, onUpdate or similar scripts are executed.

Chapter 27
Advanced Configuration

OpenIDM is a highly customizable, extensible identity management system. For the most part, the customization and configuration required for a "typical" deployment is described earlier in this book. This chapter describes advanced configuration methods that would usually not be required in a deployment, but that might assist in situations that require a high level of customization.

27.1. Advanced Startup Configuration

A customizable startup configuration file (named `launcher.json`) enables you to specify how the OSGi Framework is started. You specify the startup configuration file with the `-c` option of the **startup** command.

Unless you are working with a highly customized deployment, you should not modify the default framework configuration.

If no configuration file is specified, the default configuration (defined in `/path/to/openidm/bin/launcher.json`) is used. The following command starts OpenIDM with an alternative startup configuration file:

```
$ ./startup.sh -c /Users/admin/openidm/bin/launcher.json
```

You can modify the default startup configuration file to specify a different startup configuration.

The customizable properties of the default startup configuration file are as follows:

- `"location" : "bundle"` - resolves to the install location. You can also load OpenIDM from a specified zip file (`"location" : "openidm.zip"`) or you can install a single jar file (`"location" : "openidm-system-2.2.jar"`).
- `"includes" : "**/openidm-system-*.jar"` - the specified folder is scanned for jar files relating to the system startup. If the value of `"includes"` is `*.jar`, you must specifically exclude any jars in the bundle that you do not want to install, by setting the `"excludes"` property.
- `"start-level" : 1` - specifies a start level for the jar files identified previously.
- `"action" : "install.start"` - a period-separated list of actions to be taken on the jar files. Values can be one or more of `"install.start.update.uninstall"`.
- `"config.properties"` - takes either a path to a configuration file (relative to the project location) or a list of configuration properties and their values. The list must be in the format `"string":"string"`, for example:

```
"config.properties" :
    {
        "property" : "value"
    },
```

- `"system.properties"` - takes either a path to a `system.properties` file (relative to the project location) or a list of system properties and their values. The list must be in the format `"string":"string"`, for example:

```
"system.properties" :
    {
        "property" : "value"
    },
```

- `"boot.properties"` - takes either a path to a `boot.properties` file (relative to the project location) or a list of boot properties and their values.The list must be in the format `"string":object`, for example:

```
"boot.properties" :
    {
        "property" : true
    },
```

Appendix A. Host and Port Information

By default, OpenIDM listens on the following ports (specified in the file /path/to/openidm/conf/boot/boot.properties):

8080

HTTP access to the REST API, requiring OpenIDM authentication. This port is not secure, exposing clear text passwords and all data that is not encrypted. This port is therefore not suitable for production use.

8443

HTTPS access to the REST API, requiring OpenIDM authentication

8444

HTTPS access to the REST API, requiring SSL mutual authentication. Clients that present certificates found in the truststore under openidm/security/ are granted access to the system.

If you have another network service that uses any of these ports, change the port numbers shown in the following excerpt of the boot.properties file:

```
openidm.port.http=8080
openidm.port.https=8443
openidm.port.mutualauth=8444
openidm.host=localhost

openidm.auth.clientauthonlyports=8444
```

Note

Some Social Identity providers expect specific ports in their configuration, usually ports 80 and 443. This information is noted in "*Social ID Provider Configuration Details*" when it occurs.

By default, OpenIDM uses `localhost` as its hostname. This can be changed through the `openidm.host` property in your `boot.properties` file.

The Jetty configuration (in `openidm/conf/jetty.xml`) references the host and ports that are specified in the `boot.properties` file.

Appendix B. Data Models and Objects Reference

OpenIDM allows you to customize a variety of objects that can be addressed via a URL or URI, and that have a common set of functions that OpenIDM can perform on them such as CRUD, query, and action.

Depending on how you intend to use them, different objects are appropriate.

OpenIDM Objects

Object Type	Intended Use	Special Functionality
Managed objects	Serve as targets and sources for synchronization, and to build virtual identities.	Provide appropriate auditing, script hooks, declarative mappings and so forth in addition to the REST interface.
Configuration objects	Ideal for look-up tables or other custom configuration, which can be configured externally like any other system configuration.	Adds file view, REST interface, and so forth
Repository objects	The equivalent of arbitrary database table access. Appropriate for managing data purely through the underlying data store or repository API.	Persistence and API access
System objects	Representation of target resource objects, such as accounts, but also resource objects such as groups.	

Object Type	Intended Use	Special Functionality
Audit objects	Houses audit data in the OpenIDM internal repository.	
Links	Defines a relation between two objects.	

B.1. Managed Objects

A *managed object* in OpenIDM is an object which represents the identity-related data managed by OpenIDM. Managed objects are stored by OpenIDM in its data store. All managed objects are JSON-based data structures.

B.1.1. Managed Object Schema

OpenIDM provides a default schema for typical managed object types, such as users and roles, but does not control the structure of objects that you store in the OpenIDM repository. You can modify or extend the schema for the default object types, and you can set up a new managed object type for any item that can be collected in a data set.

The `_rev` property of a managed object is reserved by OpenIDM for internal use, and is not explicitly part of its schema. This property specifies the revision of the object in the repository. This is the same value that is exposed as the object's ETag through the REST API. The content of this attribute is not defined. No consumer should make any assumptions of its content beyond equivalence comparison. This attribute may be provided by the underlying data store.

Schema validation is performed by the policy service and can be configured according to the requirements of your deployment. For more information, see "*Using Policies to Validate Data*".

Properties can be defined to be strictly derived from other properties within the object. This allows computed and composite values to be created in the object. Such properties are named *virtual properties*. The value of a virtual property is computed only when that property is retrieved.

B.1.2. Data Consistency

Single-object operations are consistent within the scope of the operation performed, limited by the capabilities of the underlying data store. Bulk operations have no consistency guarantees. OpenIDM does not expose any transactional semantics in the managed object access API.

For information on conditional header access through the REST API, see "Conditional Operations".

B.1.3. Managed Object Triggers

Triggers are user-definable functions that validate or modify object or property state.

B.1.3.1. State Triggers

Managed objects are resource-oriented. A set of triggers is defined to intercept the supported request methods on managed objects. Such triggers are intended to perform authorization, redact, or modify objects before the action is performed. The object being operated on is in scope for each trigger, meaning that the object is retrieved by the data store before the trigger is fired.

If retrieval of the object fails, the failure occurs before any trigger is called. Triggers are executed before any optimistic concurrency mechanisms are invoked. The reason for this is to prevent a potential attacker from getting information about an object (including its presence in the data store) before authorization is applied.

onCreate

Called upon a request to create a new object. Throwing an exception causes the create to fail.

postCreate

Called after the creation of a new object is complete.

onRead

Called upon a request to retrieve a whole object or portion of an object. Throwing an exception causes the object to not be included in the result. This method is also called when lists of objects are retrieved via requests to its container object; in this case, only the requested properties are included in the object. Allows for uniform access control for retrieval of objects, regardless of the method in which they were requested.

onUpdate

Called upon a request to store an object. The `oldObject` and `newObject` variables are in-scope for the trigger. The `oldObject` represents a complete object, as retrieved from the data store. The trigger can elect to change `newObject` properties. If, as a result of the trigger, the values of the `oldObject` and `newObject` are identical (that is, update is reverted), the update ends prematurely, but successfully. Throwing an exception causes the update to fail.

postUpdate

Called after an update request is complete.

onDelete

Called upon a request to delete an object. Throwing an exception causes the deletion to fail.

postDelete

Called after an object is deleted.

onSync

Called when a managed object is changed, and the change triggers an implicit synchronization operation. The implicit synchronization operation is triggered by calling the sync service, which attempts to to go through all the configured managed-system mappings, defined in `sync.json`. The sync service returns either a response or an error. For both the response and the error, script that is referenced by the `onSync` hook is called.

You can use this hook to inject business logic when the sync service either fails or succeeds to synchronize all applicable mappings. For an example of how the `onSync` hook is used to revert partial successful synchronization operations, see "Configuring Synchronization Failure Compensation".

B.1.3.2. Object Storage Triggers

An object-scoped trigger applies to an entire object. Unless otherwise specified, the object itself is in scope for the trigger.

onValidate

Validates an object prior to its storage in the data store. If an exception is thrown, the validation fails and the object is not stored.

onStore

Called just prior to when an object is stored in the data store. Typically used to transform an object just prior to its storage (for example, encryption).

B.1.3.3. Property Storage Triggers

A property-scoped trigger applies to a specific property within an object. Only the property itself is in scope for the trigger. No other properties in the object should be accessed during execution of the trigger. Unless otherwise specified, the order of execution of property-scoped triggers is intentionally left undefined.

onValidate

Validates a given property value after its retrieval from and prior to its storage in the data store. If an exception is thrown, the validation fails and the property is not stored.

onRetrieve

Called in the result of a query request. Executed only when the `executeOnRetrieve` condition shows a full managed object.

onStore

Called prior to when an object is stored in the data store. Typically used to transform a given property prior to its object's storage.

B.1.3.4. Storage Trigger Sequences

Triggers are executed in the following order:

Object Retrieval Sequence

1. Retrieve the raw object from the data store
2. The `executeOnRetrieve` boolean is used to see if a full managed object is returned. The sequence continues if the boolean is set to `true`.
3. Call object `onRetrieve` trigger
4. Per-property within the object, call property `onRetrieve` trigger

Object Storage Sequence

1. Per-property within the object:
    - Call property `onValidate` trigger
    - Call object `onValidate` trigger
2. Per-property trigger within the object:
    - Call property `onStore` trigger
    - Call object `onStore` trigger
    - Store the object with any resulting changes to the data store

B.1.4. Managed Object Encryption

Sensitive object properties can be encrypted prior to storage, typically through the property `onStore` trigger. The trigger has access to configuration data, which can include arbitrary attributes that you define, such as a symmetric encryption key. Such attributes can be decrypted during retrieval from the data store through the property `onRetrieve` trigger.

B.1.5. Managed Object Configuration

Configuration of managed objects is provided through an array of managed object configuration objects.

```
{
  "objects": [ managed-object-config object, ... ]
}
```

objects

array of managed-object-config objects, required

Specifies the objects that the managed object service manages.

Managed-Object-Config Object Properties

Specifies the configuration of each managed object.

```
{
  "name"      : string,
  "schema"    : {
     json-schema object,
     "properties": { property-configuration objects },
  }
  "onCreate"  : script object,
  "postCreate": script object,
  "onRead"    : script object,
  "onUpdate"  : script object,
  "postUpdate": script object,
  "onDelete"  : script object,
  "postDelete": script object,
  "onValidate": script object,
  "onRetrieve": script object,
  "onStore"   : script object,
  "onSync"   : script object
}
```

name

string, required

The name of the managed object. Used to identify the managed object in URIs and identifiers.

schema

json-schema object, optional

The schema to use to validate the structure and content of the managed object. The schema-object format is specified by the JSON Schema specification.

properties

list of property-config objects, optional

A list of property specifications.

onCreate

script object, optional

A script object to trigger when the creation of an object is being requested. The object to be created is provided in the root scope as an `object` property. The script can change the object. If an exception is thrown, the create aborts with an exception.

postCreate

script object, optional

A script object to trigger after an object is created, but before any targets are synchronized.

onRead

script object, optional

A script object to trigger when the read of an object is being requested. The object being read is provided in the root scope as an `object` property. The script can change the object. If an exception is thrown, the read aborts with an exception.

onUpdate

script object, optional

A script object to trigger when an update to an object is requested. The old value of the object being updated is provided in the root scope as an `oldObject` property. The new value of the object being updated is provided in the root scope as a `newObject` property. The script can change the `newObject`. If an exception is thrown, the update aborts with an exception.

postUpdate

script object, optional

A script object to trigger after an update to an object is complete, but before any targets are synchronized. The value of the object before the update is provided in the root scope as an `oldObject` property. The value of the object after the update is provided in the root scope as a `newObject` property.

onDelete

script object, optional

A script object to trigger when the deletion of an object is being requested. The object being deleted is provided in the root scope as an `object` property. If an exception is thrown, the deletion aborts with an exception.

postDelete

script object, optional

A script object to trigger after a delete of an object is complete, but before any further synchronization. The value of the deleted object is provided in the root scope as an `oldObject` property.

onValidate

script object, optional

A script object to trigger when the object requires validation. The object to be validated is provided in the root scope as an `object` property. If an exception is thrown, the validation fails.

onRetrieve

script object, optional

A script object to trigger when an object is retrieved from the repository. The object that was retrieved is provided in the root scope as an `object` property. The script can change the object. If an exception is thrown, then object retrieval fails.

onStore

script object, optional

A script object to trigger when an object is about to be stored in the repository. The object to be stored is provided in the root scope as an `object` property. The script can change the object. If an exception is thrown, then object storage fails.

onSync

script object, optional

A script object to trigger when a change to a managed object triggers an implicit synchronization operation. The script has access to the `syncResults` object, the `request` object, the state of the object before the change (`oldObject`) and the state of the object after the change (`newObject`). The script can change the object.

Script Object Properties

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

type

string, required

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

source, file

string, required (only one, source or file is required)

Specifies the source code of the script to be executed (if the keyword is "source"), or a pointer to the file that contains the script (if the keyword is "file").

Property Config Properties

```
{
  "property-name"   : string,
  "onValidate"      : script object,
  "onRetrieve"      : script object,
  "onStore"         : script object,
  "encryption"      : property-encryption object,
  "secureHash"      : property-hash object,
  "scope"           : string,
  "title"           : string,
  "viewable"        : boolean true/false,
  "type"            : data type,
  "searchable"      : boolean true/false,
  "userEditable"    : boolean true/false,
  "minLength"       : positive integer,
  "pattern"         : string,
  "policies"        : policy object,
  "required"        : boolean true/false,
  "isVirtual"       : boolean true/false,
  "returnByDefault" : boolean true/false
}
```

property-name

string, required

The name of the property being configured.

onValidate

script object, optional

A script object to trigger when the property requires validation. The value of the property to be validated is provided in the root scope as the `property` property. If an exception is thrown, validation fails.

onRetrieve

script object, optional

A script object to trigger once a property is retrieved from the repository. That property may be one of two related variables: `property` and `propertyName`. The property that was retrieved is provided in the root scope as the `propertyName` variable; its value is provided as the `property` variable. If an exception is thrown, then object retrieval fails.

onStore

script object, optional

A script object to trigger when a property is about to be stored in the repository. That property may be one of two related variables: `property` and `propertyName`. The property that was retrieved is provided in the root scope as the `propertyName` variable; its value is provided as the `property` variable. If an exception is thrown, then object storage fails.

encryption

property-encryption object, optional

Specifies the configuration for encryption of the property in the repository. If omitted or null, the property is not encrypted.

secureHash

property-hash object, optional

Specifies the configuration for hashing of the property value in the repository. If omitted or null, the property is not hashed.

scope

string, optional

Specifies whether the property should be filtered from HTTP/external calls. The value can be either `"public"` or `"private"`. `"private"` indicates that the property should be filtered, `"public"` indicates no filtering. If no value is set, the property is assumed to be public and thus not filtered.

title

string, required

A human-readable string, used to display the property in the UI.

viewable

boolean, true/false

Specifies whether this property is viewable in the object's profile in the UI. True by default.

type

data type, required

The data type for the property value; can be String, Array, Boolean, Integer, Number, Object, or Resource Collection.

searchable

boolean, true/false

Specifies whether this property can be used in a search query on the managed object. A searchable property is visible within the Managed Object data grid in the Self-Service UI. False by default.

userEditable

boolean, true/false

Specifies whether users can edit the property value in the UI. This property applies in the context of the self-service UI, in which users are able to edit certain properties of their own accounts. False by default.

minLength

positive integer, optional

The minimum number of characters that the value of this property must have.

pattern

string, optional

Any specific pattern to which the value of the property must adhere. For example, a property whose value is a date might require a specific date format. Patterns specified here must follow regular expression syntax.

policies

policy object, optional

Any policy validation that must be applied to the property.

required

boolean, true/false

Specifies whether or the property must be supplied when an object of this type is created.

isVirtual

boolean, true/false

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

The most recently calculated value of a virtual property is persisted by default. The persistence of virtual property values allows OpenIDM to compare the new value of the property against the last calculated value, and therefore to detect change events during synchronization.

Virtual property values are not persisted by default if you are using an explicit mapping.

returnByDefault

boolean, true/false

For virtual properties, specifies whether the property will be returned in the results of a query on an object of this type if it is not explicitly requested. Virtual attributes are not returned by default.

Property Encryption Object

```
{
  "cipher": string,
  "key"   : string
}
```

cipher

string, optional

The cipher transformation used to encrypt the property. If omitted or null, the default cipher of `"AES/CBC/PKCS5Padding"` is used.

key

string, required

The alias of the key in the OpenIDM cryptography service keystore used to encrypt the property.

Property Hash Object

```
{
    "algorithm" : "string",
    "type" : "string"
}
```

algorithm

string, required

The algorithm that should be used to hash the value. The following hash algorithms are supported: `MD5`, `SHA-1`, `SHA-256`, `SHA-384`, `SHA-512`.

type

string, optional

The type of hashing. Currently only salted hash is supported. If this property is omitted or null, the default `"salted-hash"` is used.

B.1.6. Custom Managed Objects

Managed objects in OpenIDM are inherently fully user definable and customizable. Like all OpenIDM objects, managed objects can maintain relationships to each other in the form of links. Managed objects are intended for use as targets and sources for synchronization operations to represent domain objects, and to build up virtual identities. The name comes from the intention that OpenIDM stores and manages these objects, as opposed to system objects that are present in external systems.

OpenIDM can synchronize and map directly between external systems (system objects), without storing intermediate managed objects. Managed objects are appropriate, however, as a way to cache the data—for example, when mapping to multiple target systems, or when decoupling the availability of systems—to more fully report and audit on all object changes during reconciliation, and to build up views that are different from the original source, such as transformed and combined or virtual views. Managed objects can also be allowed to act as an authoritative source if no other appropriate source is available.

Other object types exist for other settings that should be available to a script, such as configuration or look-up tables that do not need audit logging.

B.1.6.1. Setting Up a Managed Object Type

To set up a managed object, you declare the object in the `conf/managed.json` file where OpenIDM is installed. The following example adds a simple `foobar` object declaration after the user object type.

```
{
    "objects": [
        {
            "name": "user"
        },
        {
            "name": "foobar"
        }
    ]
}
```

B.1.6.2. Manipulating Managed Objects Declaratively

By mapping an object to another object, either an external system object or another internal managed object, you automatically tie the object life cycle and property settings to the other object. For more information, see "*Synchronizing Data Between Resources*".

B.1.6.3. Manipulating Managed Objects Programmatically

You can address managed objects as resources using URLs or URIs with the `managed/` prefix. This works whether you address the managed object internally as a script running in OpenIDM or externally through the REST interface.

You can use all resource API functions in script objects for create, read, update, delete operations, and also for arbitrary queries on the object set, but not currently for arbitrary actions. For more information, see "*Scripting Reference*".

OpenIDM supports concurrency through a multi version concurrency control (MVCC) mechanism. In other words, each time an object changes, OpenIDM assigns it a new revision.

Objects can be arbitrarily complex as long as they use supported types, such as maps, lists, numbers, strings, and booleans as defined in JSON.

B.1.6.3.1. Creating Objects

The following script example creates an object type.

```
openidm.create("managed/foobar", "myidentifier", mymap)
```

B.1.6.3.2. Updating Objects

The following script example updates an object type.

```
var expectedRev = origMap._rev
openidm.update("managed/foobar/myidentifier", expectedRev, mymap)
```

The MVCC mechanism requires that `expectedRev` be set to the expected revision of the object to update. You obtain the revision from the object's `_rev` property. If something else changes the object concurrently, OpenIDM rejects the update, and you must either retry or inspect the concurrent modification.

B.1.6.3.3. Patching Objects

You can partially update a managed or system object using the patch method, which changes only the specified properties of the object.

The following script example updates an object type.

```
openidm.patch("managed/foobar/myidentifier", rev, value)
```

The patch method supports a revision of `"null"`, which effectively disables the MVCC mechanism, that is, changes are applied, regardless of revision. In the REST interface, this matches the `If-Match: "*"` condition supported by patch. Alternatively, you can omit the "If-Match: *" header.

For managed objects, the API supports patch by query, so the caller does not need to know the identifier of the object to change.

```
$ curl \
 --cacert self-signed.crt \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "Content-Type: application/json" \
 --request POST \
 --data '[{
  "operation":"replace",
  "field":"/password",
  "value":"Passw0rd"
  }]' \
 "https://localhost:8443/openidm/managed/user?_action=patch&_queryId=for-userName&uid=DDOE"
```

For the syntax on how to formulate the query `_queryId=for-userName&uid=DDOE` see "Querying Object Sets".

B.1.6.3.4. Deleting Objects

The following script example deletes an object type.

```
var expectedRev = origMap._rev
openidm.delete("managed/foobar/myidentifier", expectedRev)
```

The MVCC mechanism requires that `expectedRev` be set to the expected revision of the object to update. You obtain the revision from the object's `_rev` property. If something else changes the object concurrently, OpenIDM rejects deletion, and you must either retry or inspect the concurrent modification.

B.1.6.3.5. Reading Objects

The following script example reads an object type.

```

val = openidm.read("managed/foobar/myidentifier")
```

B.1.6.3.6. Querying Object Sets

You can query managed objects using common query filter syntax, or by configuring predefined queries in your repository configuration. The following script example queries managed user objects whose userName is Smith.

```
var qry = {
    "_queryFilter" : "/userName eq \"smith\""
};
val = openidm.query("managed/user", qry);
```

For more information, see "Defining and Calling Queries".

B.1.7. Accessing Managed Objects Through the REST API

OpenIDM exposes all managed object functionality through the REST API unless you configure a policy to prevent such access. In addition to the common REST functionality of create, read, update, delete, patch, and query, the REST API also supports patch by query. For more information, see "*REST API Reference*".

OpenIDM requires authentication to access the REST API. The authentication configuration is provided in your project's `conf/authentication.json` file. The default authorization filter script is `openidm/bin/defaults/script/router-authz.js`. For more information, see "The Authentication Model".

B.2. Configuration Objects

OpenIDM provides an extensible configuration to allow you to leverage regular configuration mechanisms.

Unlike native OpenIDM configuration, which OpenIDM interprets automatically and can start new services, OpenIDM stores custom configuration objects and makes them available to your code through the API.

For an introduction to the standard configuration objects, see "*Configuring the Server*".

B.2.1. When To Use Custom Configuration Objects

Configuration objects are ideal for metadata and settings that need not be included in the data to reconcile. In other words, use configuration objects for data that does not require audit log, and does not serve directly as a target or source for mappings.

Although you can set and manipulate configuration objects both programmatically and manually, configuration objects are expected to change slowly, perhaps through a mix of both manual file updates and programmatic updates. To store temporary values that can change frequently and that you do not expect to be updated by configuration file changes, custom repository objects might be more appropriate.

B.2.2. Custom Configuration Object Naming Conventions

By convention custom configuration objects are added under the reserved context, `config/custom`.

You can choose any name under `config/context`. Be sure, however, to choose a value for *context* that does not clash with future OpenIDM configuration names.

B.2.3. Mapping Configuration Objects To Configuration Files

If you have not disabled the file based view for configuration, you can view and edit all configuration including custom configuration in `openidm/conf/*.json` files. The configuration maps to a file named `context-config-name.json`, where *context* for custom configuration objects is `custom` by convention, and *config-name* is the configuration object name. A configuration object named `escalation` thus maps to a file named `conf/custom-escalation.json`.

OpenIDM detects and automatically picks up changes to the file.

OpenIDM also applies changes made through APIs to the file.

By default, OpenIDM stores configuration objects in the repository. The file view is an added convenience aimed to help you in the development phase of your project.

B.2.4. Configuration Objects File & REST Payload Formats

By default, OpenIDM maps configuration objects to JSON representations.

OpenIDM represents objects internally in plain, native types like maps, lists, strings, numbers, booleans, null. OpenIDM constrains the object model to simple types so that mapping objects to external representations is trivial.

The following example shows a representation of a configuration object with a look-up map.

```
{
    "CODE123" : "ALERT",
    "CODE889" : "IGNORE"
}
```

In the JSON representation, maps are represented with braces (`{ }`), and lists are represented with brackets (`[ ]`). Objects can be arbitrarily complex, as in the following example.

```
{
    "CODE123" : {
        "email" : ["sample@sample.com", "john.doe@somedomain.com"],
        "sms" : ["555666777"]
    }
    "CODE889" : "IGNORE"
}
```

B.2.5. Accessing Configuration Objects Through the REST API

You can list all available configuration objects, including system and custom configurations, using an HTTP GET on `/openidm/config`.

The `_id` property in the configuration object provides the link to the configuration details with an HTTP GET on `/openidm/config/id-value`. By convention, the *id-value* for a custom configuration object called `escalation` is `custom/escalation`.

OpenIDM supports REST mappings for create, read, update, delete, patch, and query of configuration objects.

B.2.6. Accessing Configuration Objects Programmatically

You can address configuration objects as resources using the URL or URI `config/` prefix both internally and also through the REST interface. The resource API provides script object functions for create, read, update, query, and delete operations.

OpenIDM supports concurrency through a multi version concurrency control mechanism. In other words, each time an object changes, OpenIDM assigns it a new revision.

Objects can be arbitrarily complex as long as they use supported types, such as maps, lists, numbers, strings, and booleans.

B.2.7. Creating Objects

The following script example creates an object type.

```
openidm.create("config/custom", "myconfig", mymap)
```

B.2.8. Updating Objects

The following script example updates a custom configuration object type.

```
openidm.update("config/custom/myconfig", mymap)
```

B.2.9. Deleting Objects

The following script example deletes a custom configuration object type.

```
openidm.delete("config/custom/myconfig")
```

B.2.10. Reading Objects

The following script example reads an object type.

```

val = openidm.read("config/custom/myconfig")
```

B.3. System Objects

System objects are pluggable representations of objects on external systems. They follow the same RESTful resource based design principles as managed objects. There is a default implementation for the OpenICF framework, which allows any connector object to be represented as a system object.

B.4. Audit Objects

Audit objects house audit data selected for local storage in the OpenIDM repository. For details, see "*Logging Audit Information*".

B.5. Links

Link objects define relations between source objects and target objects, usually relations between managed objects and system objects. The link relationship is established by provisioning activity that either results in a new account on a target system, or a reconciliation or synchronization scenario that takes a `LINK` action.

Appendix C. Synchronization Reference

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

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

C.1. Object-Mapping Objects

An object-mapping object specifies the configuration for a mapping of source objects to target objects.

```
{
  "name"              : string,
  "source"            : string,
  "target"            : string,
  "links"             : string,
  "enableSync"        : boolean,
  "validSource"       : script object,
  "validTarget"       : script object,
  "sourceCondition"   : script object or queryFilter string,
  "correlationQuery"  : script object,
  "correlationScript": script object,
  "linkQualifier"     : script object,
  "properties"        : [ property object, ... ],
  "policies"          : [ policy object, ... ],
  "onCreate"          : script object,
  "onUpdate"          : script object,
  "onDelete"          : script object,
  "onLink"            : script object,
  "onUnlink"          : script object,
  "result"            : script object
}
```

Mapping Object Properties

name

string, required

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

source

string, required

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

target

string, required

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

links

string, optional

Enables reuse of the links created in another mapping. Example: `"systemLdapAccounts_managedUser"` reuses the links created by a previous mapping whose `name` is `"systemLdapAccounts_managedUser"`.

enableSync

boolean, true or false

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

Default : `true`

validSource

script object, optional

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

validTarget

script object, optional

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

sourceCondition

script object or `queryFilter` string, optional

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

correlationQuery

script object, optional

A script that yields a query object to query the target object set when a source object has no linked target. The syntax for writing the query depends on the target system of the correlation. For examples of correlation queries, see "Correlating Source Objects With Existing Target Objects". The source object is provided in the `"source"` property in the script scope.

correlationScript

script object, optional

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

properties

array of property-mapping objects, optional

Specifies mappings between source object properties and target object properties, with optional transformation scripts.

policies

array of policy objects, optional

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

onCreate

script object, optional

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

onUpdate

script object, optional

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

onDelete

script object, optional

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

onLink

script object, optional

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

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

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

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

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

onUnlink

script object, optional

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

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

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

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

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

result

script object, optional

A script for each mapping event, executed only after a successful reconciliation.

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

- `source` - provides statistics about the source phase of the reconciliation
- `target` - provides statistics about the target phase of the reconciliation
- `global` - provides statistics about the entire reconciliation operation

C.1.1. Property Objects

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

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

Property Object Properties

target

string, required

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

source

string, optional

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

transform

script object, optional

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

condition

script object, optional

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

default

any value, optional

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

C.1.2. Policy Objects

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

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

Policy Object Properties

condition

script object or queryFilter condition, optional

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

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

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

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

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

situation

string, required

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

action

string or script object, required

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

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

postAction

script object, optional

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

The `postAction` script has the following variables available in its scope: `source`, `target`, `action`, `sourceAction`, `linkQualifier`, and `reconID`. `sourceAction` is `true` if the action was performed during the source reconciliation phase, and `false` if the action was performed during the target reconciliation phase. For more information, see "How OpenIDM Assesses Synchronization Situations".

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

C.1.2.1. Script Object

Script objects take the following form.

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

type

string, required

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

source

string, required

Specifies the source code of the script to be executed.

C.2. Links

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

```
links/mapping
```

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

Link entries have the following structure.

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

_id

string

The identifier of the link object.

_rev

string, required

The value of link object's revision.

linkType

string, required

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

firstId

string, required

The identifier of the first of the two linked objects.

secondId

string

The identifier of the second of the two linked objects.

C.3. Queries

OpenIDM performs the following queries on a link object set.

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

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

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

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

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

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

C.4. Reconciliation

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

1. Iterate through all objects for the object set specified as `"source"`. For each source object, carry out the following steps.

   a. Look for a link to a target object in the link object set, and perform a correlation query (if defined).

   b. Determine the link condition, as well as whether a target object can be found.

   c. Determine the action to perform based on the policy defined for the condition.

   d. Perform the action.

   e. Keep track of the target objects for which a condition and action has already been determined.

   f. Write the results.

2. Iterate through all object identifiers for the object set specified as `"target"`. For each identifier, carry out the following steps.

   a. Find the target in the link object set.

      Determine if the target object was handled in the first phase.

   b. Determine the action to perform based on the policy defined for the condition.

   c. Perform the action.

   d. Write the results.

3. Iterate through all link objects, carrying out the following steps.

   a. If the `reconId` is `"my"`, then skip the object.

      If the `reconId` is not recognized, then the source or the target is missing.

   b. Determine the action to perform based on the policy.

c. Perform the action.

d. Store the `reconId` identifer in the mapping to indicate that it was processed in this run.

> **Note**
>
> To optimize a reconciliation operation, the reconciliation process does not attempt to correlate source objects to target objects if the set of target objects is empty when the correlation is started. For information on changing this default behaviour, see "Optimizing Reconciliation Performance".

C.5. REST API

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

Request

Example:

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

Response (success)

Example:

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

Response (synchronization failure)

Example:

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

Appendix D. REST API Reference

Representational State Transfer (REST) is 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. OpenIDM provides a RESTful API for accessing managed objects, system objects, workflows, and some elements of the system configuration.

The following section describes the ForgeRock Common REST API. See "Common REST and OpenIDM" for information specific to the OpenIDM implementation of Common REST.

D.1. About ForgeRock Common REST

ForgeRock® Common REST is a common REST API framework. It works across the ForgeRock platform to provide common ways to access web resources and collections of resources. Adapt the examples in this section to your resources and deployment.

D.1.1. Common REST Resources

Servers generally return JSON-format resources, though resource formats can depend on the implementation.

Resources in collections can be found by their unique identifiers (IDs). IDs are exposed in the resource URIs. For example, if a server has a user collection under `/users`, then you can access a user at `/users/user-id`. The ID is also the value of the `_id` field of the resource.

Resources are versioned using revision numbers. A revision is specified in the resource's `_rev` field. Revisions make it possible to figure out whether to apply changes without resource locking and without distributed transactions.

D.1.2. Common REST Verbs

The Common REST APIs use the following verbs, sometimes referred to collectively as CRUDPAQ. For details and HTTP-based examples of each, follow the links to the sections for each verb.

Create

Add a new resource.

This verb maps to HTTP PUT or HTTP POST.

For details, see "Create".

Read

Retrieve a single resource.

This verb maps to HTTP GET.

For details, see "Read".

Update

Replace an existing resource.

This verb maps to HTTP PUT.

For details, see "Update".

Delete

Remove an existing resource.

This verb maps to HTTP DELETE.

For details, see "Delete".

Patch

Modify part of an existing resource.

This verb maps to HTTP PATCH.

For details, see "Patch".

Action

Perform a predefined action.

This verb maps to HTTP POST.

For details, see "Action".

Query

Search a collection of resources.

This verb maps to HTTP GET.

For details, see "Query".

D.1.3. Common REST Parameters

Common REST reserved query string parameter names start with an underscore, `_`.

Reserved query string parameters include, but are not limited to, the following names:

```
_action
_api
_crestapi
_fields
_mimeType
_pageSize
_pagedResultsCookie
_pagedResultsOffset
_prettyPrint
_queryExpression
_queryFilter
_queryId
_sortKeys
_totalPagedResultsPolicy
```

Note

Some parameter values are not safe for URLs, so URL-encode parameter values as necessary.

Continue reading for details about how to use each parameter.

D.1.4. Common REST Extension Points

The *action* verb is the main vehicle for extensions. For example, to create a new user with HTTP POST rather than HTTP PUT, you might use `/users?_action=create`. A server can define additional actions. For example, `/tasks/1?_action=cancel`.

A server can define *stored queries* to call by ID. For example, `/groups?_queryId=hasDeletedMembers`. Stored queries can call for additional parameters. The parameters are also passed in the query string. Which parameters are valid depends on the stored query.

D.1.5. Common REST API Documentation

Common REST APIs often depend at least in part on runtime configuration. Many Common REST endpoints therefore serve *API descriptors* at runtime. An API descriptor documents the actual API as it is configured.

Use the following query string parameters to retrieve API descriptors:

`_api`

> Serves an API descriptor that complies with the OpenAPI specification.
>
> This API descriptor represents the API accessible over HTTP. It is suitable for use with popular tools such as Swagger UI.

`_crestapi`

> Serves a native Common REST API descriptor.
>
> This API descriptor provides a compact representation that is not dependent on the transport protocol. It requires a client that understands Common REST, as it omits many Common REST defaults.

Note

Consider limiting access to API descriptors in production environments in order to avoid unnecessary traffic.

To provide documentation in production environments, see "To Publish OpenAPI Documentation" instead.

To Publish OpenAPI Documentation

In production systems, developers expect stable, well-documented APIs. Rather than retrieving API descriptors at runtime through Common REST, prepare final versions, and publish them alongside the software in production.

Use the OpenAPI-compliant descriptors to provide API reference documentation for your developers as described in the following steps:

1. Configure the software to produce production-ready APIs.

   In other words, the software should be configured as in production so that the APIs are identical to what developers see in production.

2. Retrieve the OpenAPI-compliant descriptor.

   The following command saves the descriptor to a file, `myapi.json`:

   ```
   $ curl -o myapi.json endpoint?_api
   ```

3. (Optional) If necessary, edit the descriptor.

   For example, you might want to add security definitions to describe how the API is protected.

   If you make any changes, then also consider using a source control system to manage your versions of the API descriptor.

4. Publish the descriptor using a tool such as Swagger UI.

   You can customize Swagger UI for your organization as described in the documentation for the tool.

D.1.6. Create

There are two ways to create a resource, either with an HTTP POST or with an HTTP PUT.

To create a resource using POST, perform an HTTP POST with the query string parameter `_action=create` and the JSON resource as a payload. Accept a JSON response. The server creates the identifier if not specified:

```
POST /users?_action=create HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: ...
Content-Type: application/json
{ JSON resource }
```

To create a resource using PUT, perform an HTTP PUT including the case-sensitive identifier for the resource in the URL path, and the JSON resource as a payload. Use the `If-None-Match: *` header. Accept a JSON response:

```
PUT /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: ...
Content-Type: application/json
If-None-Match: *
{ JSON resource }
```

The `_id` and content of the resource depend on the server implementation. The server is not required to use the `_id` that the client provides. The server response to the create request indicates the resource location as the value of the `Location` header.

If you include the `If-None-Match` header, its value must be `*`. In this case, the request creates the object if it does not exist, and fails if the object does exist. If you include the `If-None-Match` header with any value other than `*`, the server returns an HTTP 400 Bad Request error. For example, creating an object with `If-None-Match: revision` returns a bad request error. If you do not include `If-None-Match: *`, the request creates the object if it does not exist, and *updates* the object if it does exist.

Parameters

You can use the following parameters:

`_prettyPrint=true`

Format the body of the response.

`_fields=field[,field...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

D.1.7. Read

To retrieve a single resource, perform an HTTP GET on the resource by its case-sensitive identifier (`_id`) and accept a JSON response:

```
GET /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
```

Parameters

You can use the following parameters:

`_prettyPrint=true`

Format the body of the response.

`_fields=field[,field...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

`_mimeType=mime-type`

Some resources have fields whose values are multi-media resources such as a profile photo for example.

By specifying both a single *field* and also the *mime-type* for the response content, you can read a single field value that is a multi-media resource.

In this case, the content type of the field value returned matches the *mime-type* that you specify, and the body of the response is the multi-media resource.

The `Accept` header is not used in this case. For example, `Accept: image/png` does not work. Use the `_mimeType` query string parameter instead.

D.1.8. Update

To update a resource, perform an HTTP PUT including the case-sensitive identifier (`_id`) as the final element of the path to the resource, and the JSON resource as the payload. Use the `If-Match: _rev` header to check that you are actually updating the version you modified. Use `If-Match: *` if the version does not matter. Accept a JSON response:

```
PUT /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: ...
Content-Type: application/json
If-Match: _rev
{ JSON resource }
```

When updating a resource, include all the attributes to be retained. Omitting an attribute in the resource amounts to deleting the attribute unless it is not under the control of your application. Attributes not under the control of your application include private and read-only attributes. In addition, virtual attributes and relationship references might not be under the control of your application.

Parameters

You can use the following parameters:

`_prettyPrint=true`

Format the body of the response.

`_fields=`*`field`*`[,`*`field`*`...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

D.1.9. Delete

To delete a single resource, perform an HTTP DELETE by its case-sensitive identifier (`_id`) and accept a JSON response:

```
DELETE /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
```

Parameters

You can use the following parameters:

`_prettyPrint=true`

Format the body of the response.

`_fields=field[,field...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

D.1.10. Patch

To patch a resource, send an HTTP PATCH request with the following parameters:

- `operation`
- `field`
- `value`
- `from` (optional with copy and move operations)

You can include these parameters in the payload for a PATCH request, or in a JSON PATCH file. If successful, you'll see a JSON response similar to:

```
PATCH /users/some-id HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: ...
Content-Type: application/json
If-Match: _rev
{ JSON array of patch operations }
```

PATCH operations apply to three types of targets:

- **single-valued**, such as an object, string, boolean, or number.
- **list semantics array**, where the elements are ordered, and duplicates are allowed.

- **set semantics array**, where the elements are not ordered, and duplicates are not allowed.

ForgeRock PATCH supports several different `operations`. The following sections show each of these operations, along with options for the `field` and `value`:

D.1.10.1. Patch Operation: Add

The `add` operation ensures that the target field contains the value provided, creating parent fields as necessary.

If the target field is single-valued, then the value you include in the PATCH replaces the value of the target. Examples of a single-valued field include: object, string, boolean, or number.

An `add` operation has different results on two standard types of arrays:

- **List semantic arrays**: you can run any of these `add` operations on that type of array:
  - If you `add` an array of values, the PATCH operation appends it to the existing list of values.
  - If you `add` a single value, specify an ordinal element in the target array, or use the `{-}` special index to add that value to the end of the list.
- **Set semantic arrays**: The list of values included in a patch are merged with the existing set of values. Any duplicates within the array are removed.

As an example, start with the following list semantic array resource:

```
{
    "fruits" : [ "orange", "apple" ]
}
```

The following add operation includes the pineapple to the end of the list of fruits, as indicated by the `-` at the end of the `fruits` array.

```
{
    "operation" : "add",
    "field" : "/fruits/-",
    "value" : "pineapple"
}
```

The following is the resulting resource:

```
{
    "fruits" : [ "orange", "apple", "pineapple" ]
}
```

D.1.10.2. Patch Operation: Copy

The copy operation takes one or more existing values from the source field. It then adds those same values on the target field. Once the values are known, it is equivalent to performing an `add` operation on the target field.

The following `copy` operation takes the value from a field named `mail`, and then runs a `replace` operation on the target field, `another_mail`.

```
[
  {
     "operation":"copy",
     "from":"mail",
     "field":"another_mail"
  }
]
```

If the source field value and the target field value are configured as arrays, the result depends on whether the array has list semantics or set semantics, as described in "Patch Operation: Add".

D.1.10.3. Patch Operation: Increment

The `increment` operation changes the value or values of the target field by the amount you specify. The value that you include must be one number, and may be positive or negative. The value of the target field must accept numbers. The following `increment` operation adds `1000` to the target value of `/user/payment`.

```
[
  {
    "operation" : "increment",
    "field" : "/user/payment",
    "value" : "1000"
  }
]
```

Since the `value` of the `increment` is a single number, arrays do not apply.

D.1.10.4. Patch Operation: Move

The move operation removes existing values on the source field. It then adds those same values on the target field. It is equivalent to performing a `remove` operation on the source, followed by an `add` operation with the same values, on the target.

The following `move` operation is equivalent to a `remove` operation on the source field, `surname`, followed by a `replace` operation on the target field value, `lastName`. If the target field does not exist, it is created.

```
[
  {
     "operation":"move",
     "from":"surname",
     "field":"lastName"
  }
]
```

To apply a `move` operation on an array, you need a compatible single-value, list semantic array, or set semantic array on both the source and the target. For details, see the criteria described in "Patch Operation: Add".

D.1.10.5. Patch Operation: Remove

The `remove` operation ensures that the target field no longer contains the value provided. If the remove operation does not include a value, the operation removes the field. The following `remove` deletes the value of the `phoneNumber`, along with the field.

```
[
  {
    "operation" : "remove",
    "field" : "phoneNumber"
  }
]
```

If the object has more than one `phoneNumber`, those values are stored as an array.

A `remove` operation has different results on two standard types of arrays:

- **List semantic arrays**: A `remove` operation deletes the specified element in the array. For example, the following operation removes the first phone number, based on its array index (zero-based):

  ```
  [
     {
        "operation" : "remove",
        "field" : "/phoneNumber/0"
     }
  ]
  ```

- **Set semantic arrays**: The list of values included in a patch are removed from the existing array.

D.1.10.6. Patch Operation: Replace

The `replace` operation removes any existing value(s) of the targeted field, and replaces them with the provided value(s). It is essentially equivalent to a `remove` followed by a `add` operation. If the arrays are used, the criteria is based on "Patch Operation: Add". However, indexed updates are not allowed, even when the target is an array.

The following `replace` operation removes the existing `telephoneNumber` value for the user, and then adds the new value of `+1 408 555 9999`.

```
[
  {
    "operation" : "replace",
    "field" : "/telephoneNumber",
    "value" : "+1 408 555 9999"
  }
]
```

A PATCH replace operation on a list semantic array works in the same fashion as a PATCH remove operation. The following example demonstrates how the effect of both operations. Start with the following resource:

```
{
    "fruits" : [ "apple", "orange", "kiwi", "lime" ],
}
```

Apply the following operations on that resource:

```
[
  {
    "operation" : "remove",
    "field" : "/fruits/0",
    "value" : ""
  },
  {
    "operation" : "replace",
    "field" : "/fruits/1",
    "value" : "pineapple"
  }
]
```

The PATCH operations are applied sequentially. The `remove` operation removes the first member of that resource, based on its array index, (`fruits/0`), with the following result:

```
[
  {
    "fruits" : [ "orange", "kiwi", "lime" ],
  }
]
```

The second PATCH operation, a `replace`, is applied on the second member (`fruits/1`) of the intermediate resource, with the following result:

```
[
  {
    "fruits" : [ "orange", "pineapple", "lime" ],
  }
]
```

D.1.10.7. Patch Operation: Transform

The `transform` operation changes the value of a field based on a script or some other data transformation command. The following `transform` operation takes the value from the field named `/objects`, and applies the `something.js` script as shown:

```
[
  {
    "operation" : "transform",
    "field" : "/objects",
    "value" : {
      "script" : {
        "type" : "text/javascript",
        "file" : "something.js"
      }
    }
  }
]
```

D.1.10.8. Patch Operation Limitations

Some HTTP client libraries do not support the HTTP PATCH operation. Make sure that the library you use supports HTTP PATCH before using this REST operation.

For example, the Java Development Kit HTTP client does not support PATCH as a valid HTTP method. Instead, the method `HttpURLConnection.setRequestMethod("PATCH")` throws `ProtocolException`.

Parameters

You can use the following parameters. Other parameters might depend on the specific action implementation:

`_prettyPrint=true`

Format the body of the response.

`_fields=`*`field`*`[,`*`field`*`...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

D.1.11. Action

Actions are a means of extending Common REST APIs and are defined by the resource provider, so the actions you can use depend on the implementation.

The standard action indicated by `_action=create` is described in "Create".

Parameters

You can use the following parameters. Other parameters might depend on the specific action implementation:

`_prettyPrint=true`

Format the body of the response.

`_fields=`*`field`*`[,`*`field`*`...]`

Return only the specified fields in the body of the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

D.1.12. Query

To query a resource collection (or resource container if you prefer to think of it that way), perform an HTTP GET and accept a JSON response, including at least a `_queryExpression`, `_queryFilter`, or `_queryId` parameter. These parameters cannot be used together:

```
GET /users?_queryFilter=true HTTP/1.1
Host: example.com
Accept: application/json
```

The server returns the result as a JSON object including a "results" array and other fields related to the query string parameters that you specify.

Parameters

You can use the following parameters:

`_queryFilter=`*`filter-expression`*

Query filters request that the server return entries that match the filter expression. You must URL-escape the filter expression.

The string representation is summarized as follows. Continue reading for additional explanation:

```
Expr           = OrExpr
OrExpr         = AndExpr ( 'or' AndExpr ) *
AndExpr        = NotExpr ( 'and' NotExpr ) *
NotExpr        = '!' PrimaryExpr | PrimaryExpr
PrimaryExpr    = '(' Expr ')' | ComparisonExpr | PresenceExpr | LiteralExpr
ComparisonExpr = Pointer OpName JsonValue
PresenceExpr   = Pointer 'pr'
LiteralExpr    = 'true' | 'false'
Pointer        = JSON pointer
OpName         = 'eq' |  # equal to
                 'co' |  # contains
                 'sw' |  # starts with
                 'lt' |  # less than
                 'le' |  # less than or equal to
                 'gt' |  # greater than
                 'ge' |  # greater than or equal to
                 STRING  # extended operator
JsonValue      = NUMBER | BOOLEAN | '"' UTF8STRING '"'
STRING         = ASCII string not containing white-space
UTF8STRING     = UTF-8 string possibly containing white-space
```

JsonValue components of filter expressions follow RFC 7159: *The JavaScript Object Notation (JSON) Data Interchange Format*. In particular, as described in section 7 of the RFC, the escape character in strings is the backslash character. For example, to match the identifier `test\`, use `_id eq 'test\\'`. In the JSON resource, the `\` is escaped the same way: `"_id":"test\\"`.

When using a query filter in a URL, be aware that the filter expression is part of a query string parameter. A query string parameter must be URL encoded as described in RFC 3986: *Uniform Resource Identifier (URI): Generic Syntax* For example, white space, double quotes ("), parentheses, and exclamation characters need URL encoding in HTTP query strings. The following rules apply to URL query components:

```
query       = *( pchar / "/" / "?" )
pchar       = unreserved / pct-encoded / sub-delims / ":" / "@"
unreserved  = ALPHA / DIGIT / "-" / "." / "_" / "~"
pct-encoded = "%" HEXDIG HEXDIG
sub-delims  = "!" / "$" / "&" / "'" / "(" / ")"
                  / "*" / "+" / "," / ";" / "="
```

`ALPHA`, `DIGIT`, and `HEXDIG` are core rules of RFC 5234: *Augmented BNF for Syntax Specifications*:

```
ALPHA       =  %x41-5A / %x61-7A   ; A-Z / a-z
DIGIT       =  %x30-39             ; 0-9
HEXDIG      =  DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
```

As a result, a backslash escape character in a *JsonValue* component is percent-encoded in the URL query string parameter as `%5C`. To encode the query filter expression `_id eq 'test\\'`, use `_id+eq+'test%5C%5C'`, for example.

A simple filter expression can represent a comparison, presence, or a literal value.

For comparison expressions use *json-pointer comparator json-value*, where the *comparator* is one of the following:

`eq` (equals)
`co` (contains)
`sw` (starts with)
`lt` (less than)
`le` (less than or equal to)
`gt` (greater than)
`ge` (greater than or equal to)

For presence, use *json-pointer pr* to match resources where the JSON pointer is present.

Literal values include true (match anything) and false (match nothing).

Complex expressions employ `and`, `or`, and `!` (not), with parentheses, `(expression)`, to group expressions.

`_queryId=identifier`

Specify a query by its identifier.

Specific queries can take their own query string parameter arguments, which depend on the implementation.

`_pagedResultsCookie=`*`string`*

The string is an opaque cookie used by the server to keep track of the position in the search results. The server returns the cookie in the JSON response as the value of `pagedResultsCookie`.

In the request `_pageSize` must also be set and non-zero. You receive the cookie value from the provider on the first request, and then supply the cookie value in subsequent requests until the server returns a `null` cookie, meaning that the final page of results has been returned.

The `_pagedResultsCookie` parameter is supported when used with the `_queryFilter` parameter. The `_pagedResultsCookie` parameter is not guaranteed to work when used with the `_queryExpression` and `_queryId` parameters.

The `_pagedResultsCookie` and `_pagedResultsOffset` parameters are mutually exclusive, and not to be used together.

`_pagedResultsOffset=`*`integer`*

When `_pageSize` is non-zero, use this as an index in the result set indicating the first page to return.

The `_pagedResultsCookie` and `_pagedResultsOffset` parameters are mutually exclusive, and not to be used together.

`_pageSize=`*`integer`*

Return query results in pages of this size. After the initial request, use `_pagedResultsCookie` or `_pageResultsOffset` to page through the results.

`_totalPagedResultsPolicy=`*`string`*

When a `_pageSize` is specified, and non-zero, the server calculates the "totalPagedResults", in accordance with the `totalPagedResultsPolicy`, and provides the value as part of the response. The "totalPagedResults" is either an estimate of the total number of paged results (`_totalPagedResultsPolicy=ESTIMATE`), or the exact total result count (`_totalPagedResultsPolicy=EXACT`). If no count policy is specified in the query, or if `_totalPagedResultsPolicy=NONE`, result counting is disabled, and the server returns value of -1 for "totalPagedResults".

`_sortKeys=[+-]`*`field`*`[,[+-]`*`field...`*`]`

Sort the resources returned based on the specified field(s), either in `+` (ascending, default) order, or in `-` (descending) order.

The `_sortKeys` parameter is not supported for predefined queries (`_queryId`).

`_prettyPrint=true`

Format the body of the response.

`_fields=field[,field...]`

Return only the specified fields in each element of the "results" array in the response.

The `field` values are JSON pointers. For example if the resource is `{"parent":{"child":"value"}}`, `parent/child` refers to the `"child":"value"`.

D.1.13. HTTP Status Codes

When working with a Common REST API over HTTP, client applications should expect at least the following HTTP status codes. Not all servers necessarily return all status codes identified here:

200 OK

The request was successful and a resource returned, depending on the request.

201 Created

The request succeeded and the resource was created.

204 No Content

The action request succeeded, and there was no content to return.

304 Not Modified

The read request included an `If-None-Match` header, and the value of the header matched the revision value of the resource.

400 Bad Request

The request was malformed.

401 Unauthorized

The request requires user authentication.

403 Forbidden

Access was forbidden during an operation on a resource.

404 Not Found

The specified resource could not be found, perhaps because it does not exist.

405 Method Not Allowed

The HTTP method is not allowed for the requested resource.

406 Not Acceptable

The request contains parameters that are not acceptable, such as a resource or protocol version that is not available.

409 Conflict

The request would have resulted in a conflict with the current state of the resource.

410 Gone

The requested resource is no longer available, and will not become available again. This can happen when resources expire for example.

412 Precondition Failed

The resource's current version does not match the version provided.

415 Unsupported Media Type

The request is in a format not supported by the requested resource for the requested method.

428 Precondition Required

The resource requires a version, but no version was supplied in the request.

500 Internal Server Error

The server encountered an unexpected condition that prevented it from fulfilling the request.

501 Not Implemented

The resource does not support the functionality required to fulfill the request.

503 Service Unavailable

The requested resource was temporarily unavailable. The service may have been disabled, for example.

D.2. Common REST and OpenIDM

OpenIDM implements the Common REST API as described in the previous section, with the exception of the following elements:

- The PATCH `transform` action is supported only on the `config` endpoint. Note that this is an optional action and not implemented everywhere across the ForgeRock Identity Platform.

- Common REST supports PATCH operations by list element index, as shown in the example in "Patch Operation: Remove". OpenIDM does not support PATCH by list element index.

D.3. URI Scheme

The URI scheme for accessing a managed object follows this convention, assuming the OpenIDM web application was deployed at /openidm.

```
/openidm/managed/type/id
```

Similar schemes exist for URIs associated with all but system objects. For more information, see "Understanding the Access Configuration Script (access.js)".

The URI scheme for accessing a system object follows this convention:

```
/openidm/system/resource-name/type/id
```

An example of a system object in an LDAP directory might be:

```
/openidm/system/ldap/account/07b46858-56eb-457c-b935-cfe6ddf769c7
```

Note that for LDAP resources, you should not map the LDAP dn to the OpenIDM uidAttribute (_id). The attribute that is used for the _id should be immutable. You should therefore map the LDAP entryUUID operational attribute to the OpenIDM _id, as shown in the following excerpt of the provisioner configuration file:

```
...
"uidAttribute" : "entryUUID",
...
```

D.4. Object Identifiers

Every managed and system object has an identifier (expressed as *id* in the URI scheme) that is used to address the object through the REST API. The REST API allows for client-generated and server-generated identifiers, through PUT and POST methods. The default server-generated identifier type is a UUID. If you create an object by using POST, a server-assigned ID is generated in the form of a UUID. If you create an object by using PUT, the client assigns the ID in whatever format you specify.

Most of the examples in this guide use client-assigned IDs, as it makes the examples easier to read.

D.5. Content Negotiation

The REST API fully supports negotiation of content representation through the Accept HTTP header. Currently, the supported content type is JSON. When you send a JSON payload, you must include the following header:

```
Accept: application/json
```

In a REST call (using the **curl** command, for example), you would include the following option to specify the noted header:

```
--header "Content-Type: application/json"
```

You can also specify the default UTF-8 character set as follows:

```
--header "Content-Type: application/json;charset=utf-8"
```

The `application/json` content type is not needed when the REST call does not send a JSON payload.

D.6. Conditional Operations

The REST API supports conditional operations through the use of the `ETag`, `If-Match` and `If-None-Match` HTTP headers. The use of HTTP conditional operations is the basis of OpenIDM's optimistic concurrency control system. Clients should make requests conditional in order to prevent inadvertent modification of the wrong version of an object. For *managed objects*, if no conditional header is specified, a default of `If-Match: *` is applied.

REST API Conditional Operations

HTTP Header	Operation	Description
`If-Match: <rev>`	PUT	Update the object if the <rev> matches the revision level of the object.
`If-Match: *`	PUT	Update the object regardless of revision level
`If-None-Match: <rev>`		Bad request
`If-None-Match: *`	PUT	Create; fails if the object already exists
When the conditional operations `If-Match`, `If-None-Match` are not used	PUT	Upsert; attempts a create, and then an update; if both attempts fail, return an error

D.7. REST Endpoints and Sample Commands

This section describes the REST endpoints and provides a number of sample commands that show the interaction with the REST interface.

D.7.1. Managing the Server Configuration Over REST

OpenIDM stores configuration objects in the repository, and exposes them under the context path `/openidm/config`. Single instance configuration objects are exposed under `/openidm/config/object-name`.

Multiple instance configuration objects are exposed under `/openidm/config/object-name/instance-name`. The following table outlines these configuration objects and how they can be accessed through the REST interface.

URI	HTTP Operation	Description
/openidm/config	GET	Returns a list of configuration objects
/openidm/config/audit	GET	Returns the current logging configuration
/openidm/config/provisioner.openicf/*provisioner-name*	GET	Returns the configuration of the specified connector
/openidm/config/router	PUT	Changes the router configuration. Modifications are provided with the `--data` option, in JSON format.
/openidm/config/*object*	PATCH	Changes one or more fields of the specified configuration object. Modifications are provided as a JSON array of patch operations.
/openidm/config/*object*	DELETE	Deletes the specified configuration object.

OpenIDM supports REST operations to create, read, update, query, and delete configuration objects.

For command-line examples of managing the configuration over REST, see "Configuring the Server Over REST".

One entry is returned for each configuration object. To obtain additional information on the configuration object, include its `pid` or `_id` in the URL. The following example displays configuration information on the `sync` object, based on a deployment using Sample 1.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/config/sync"
{
  "mappings": [ {
    "target" : "managed/user",
    "correlationQuery" : {
      "type" : "text/javascript",
      "source" : "var query = {'_queryId' : 'for-userName', 'uid' : source.name};query;"
    },
    "properties" : [ {
      "target" : "_id",
      "source" : "_id"
    }, {
      "target" : "description",
      "source" : "description"
    }, {
      "target" : "givenName",
      "source" : "firstname"
    }, {
      "target" : "mail",
      "source" : "email"
    },
 {
...
```

D.7.2. Managing Users Over REST

User objects are stored in the repository and are exposed under the context path `/managed/user`. Many examples of REST calls related to this context path exist throughout this document. The following table lists available functionality associated with the `/managed/user` context path.

URI	HTTP Operation	Description
/openidm/managed/user?_queryId=query-all-ids	GET	List the IDs of all the managed users in the repository
/openidm/managed/user?_queryId=query-all	GET	List all info for the managed users in the repository
/openidm/managed/user?_queryFilter=*filter*	GET	Query the managed user object with the defined filter.
/openidm/managed/user/*_id*	GET	Retrieve the JSON representation of a specific user
/openidm/managed/user/*_id*	PUT	Create a new user
/openidm/managed/user/*_id*	PUT	Update a user entry (replaces the entire entry)
/openidm/managed/user?_action=create	POST	Create a new user
/openidm/managed/user?_action=patch&_queryId=for-userName&uid=*userName*	POST	Update a user (can be used to replace the value of one or more existing attributes)
/openidm/managed/user/*_id*	PATCH	Update specified fields of a user entry
/openidm/managed/user/*_id*	DELETE	Delete a user entry

For a number of sample commands that show how to manage users over REST, see "Working with Managed Users".

D.7.3. Managing System Objects Over REST

System objects, that is, objects that are stored in remote systems, are exposed under the `/openidm/system` context. OpenIDM provides access to system objects over REST, as listed in the following table.

URI	HTTP Operation	Description
/openidm/system?_action=*action-name*	POST	`_action=availableConnectors` returns a list of the connectors that are available in `openidm/connectors` or in `openidm/bundle`. `_action=createCoreConfig` takes the supplied connector reference (`connectorRef`) and adds the configuration properties required for that connector. This generates a core connector

URI	HTTP Operation	Description
		configuration that you can use to create a full configuration with the `createFullConfig` action. `_action=createFullConfig` generates a complete connector configuration, using the configuration properties from the `createCoreConfig` action, and retrieving the object types and operation options from the resource, to complete the configuration. `_action=test` returns a list of all remote systems, with their status, and supported object types. `_action=testConfig` validates the connector configuration provided in the POST body. `_action=liveSync` triggers a liveSync operation on the specified source object. `_action=authenticate` authenticates to the specified system with the credentials provided.
/openidm/system/*system-name*?_action=*action-name*	POST	`_action=test` tests the status of the specified system.
/openidm/system/*system-name*/*system-object*?_action=*action-name*	POST	`_action=liveSync` triggers a liveSync operation on the specified system object. `_action=script` runs the specified script on the system object. `_action=authenticate` authenticates to the specified system object, with the provided credentials. `_action=create` creates a new system object.
/openidm/system/*system-name*/*system-object*?_queryId=query-all-ids	GET	Lists all IDs related to the specified system object, such as users, and groups.
/openidm/system/*system-name*/*system-object*?_queryFilter=*filter*	GET	Lists the item(s) associated with the query filter.
/openidm/system/*system-name*/*system-object*/*id*	PUT	Creates a system object, or updates the system object, if it exists (replaces the entire object).
/openidm/system/*system-name*/*system-object*/*id*	PATCH	Updates the specified fields of a system object

URI	HTTP Operation	Description
/openidm/system/*system-name*/*system-object*/*id*	DELETE	Deletes a system object

Note

When you create a system object with a PUT request (that is, specifying a client-assigned ID), you should specify the ID in the URL only and not in the JSON payload. If you specify a different ID in the URL and in the JSON payload, the request will fail, with an error similar to the following:

```
{
    "code":500,
    "reason":"Internal Server Error",
    "message":"The uid attribute is not single value attribute."
}
```

A `POST` request with a `patch` action is not currently supported on system objects. To patch a system object, you must send a `PATCH` request.

Returning a list of the available connector configurations

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system?_action=availableConnectors"
```

Returning a list of remote systems, and their status

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system?_action=test"
[
  {
    "ok": true,
    "displayName": "LDAP Connector",
    "connectorRef": {
      "bundleVersion": "[1.4.0.0,1.5.0.0)",
      "bundleName": "org.forgerock.openicf.connectors.ldap-connector",
      "connectorName": "org.identityconnectors.ldap.LdapConnector"
    },
    "objectTypes": [
      "__ALL__",
      "group",
      "account"
    ],
    "config": "config/provisioner.openicf/ldap",
    "enabled": true,
    "name": "ldap"
  }
]
```

Two options for running a liveSync operation on a specified system object

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system?_action=liveSync&source=system/ldap/account"
{
  "_rev": "1",
  "_id": "SYSTEMLDAPACCOUNT",
  "connectorData": {
    "nativeType": "integer",
    "syncToken": 0
  }
}
```

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system/ldap/account?_action=liveSync"

{
  "_rev": "2",
  "_id": "SYSTEMLDAPACCOUNT",
  "connectorData": {
    "nativeType": "integer",
    "syncToken": 0
  }
}
```

Running a script on a system object

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system/ldap/account?_action=script&_scriptId=addUser"
```

Authenticating to a system object

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/system/ldap/account?
_action=authenticate&username=bjensen&password=Passw0rd"
{
  "_id": "fc252fd9-b982-3ed6-b42a-c76d2546312c"
}
```

Creating a new system object

```
$ curl \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --data '{
    "cn":"James Smith",
    "dn":"uid=jsmith,ou=people,dc=example,dc=com",
    "uid":"jsmith",
    "sn":"Smith",
    "givenName":"James",
    "mail": "jsmith@example.com",
    "description":"Created by OpenIDM REST"}' \
 --request POST \
 "http://localhost:8080/openidm/system/ldap/account?_action=create"
{
    "telephoneNumber":null,
    "description":"Created by OpenIDM REST",
    "mail":"jsmith@example.com",
    "givenName":"James",
    "cn":"James Smith",
    "dn":"uid=jsmith,ou=people,dc=example,dc=com",
    "uid":"jsmith",
    "ldapGroups":[],
    "sn":"Smith",
    "_id":"07b46858-56eb-457c-b935-cfe6ddf769c7"
}
```

Renaming a system object

You can rename a system object simply by supplying a new naming attribute value in a PUT request. The PUT request replaces the entire object. The naming attribute depends on the external resource.

The following example renames an object on an LDAP server, by changing the DN of the LDAP object (effectively performing a modDN operation on that object).

The example renames the user created in the previous example.

```
$ curl \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "If-Match: *" \
 --data '{
    "cn":"James Smith",
    "dn":"uid=jimmysmith,ou=people,dc=example,dc=com",
    "uid":"jimmysmith",
    "sn":"Smith",
    "givenName":"James",
    "mail": "jsmith@example.com"}' \
 --request PUT \
 "http://localhost:8080/openidm/system/ldap/account/07b46858-56eb-457c-b935-cfe6ddf769c7"
{
  "mail":"jsmith@example.com",
  "cn":"James Smith",
  "sn":"Smith",
  "dn":"uid=jimmysmith,ou=people,dc=example,dc=com",
  "ldapGroups":[],
  "telephoneNumber":null,
  "description":"Created by OpenIDM REST",
  "givenName":"James",
  "uid":"jimmysmith",
  "_id":"07b46858-56eb-457c-b935-cfe6ddf769c7"
}
```

List the IDs associated with a specific system object

```
$ curl \
 --header "X-OpenIDM-Password: openidm-admin" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/system/ldap/account?_queryId=query-all-ids"
{
  "remainingPagedResults": -1,
  "pagedResultsCookie": null,
  "resultCount": 3,
  "result": [
        {
           "dn": "uid=jdoe,ou=People,dc=example,dc=com",
           "_id": "1ff2e78f-4c4c-300c-b8f7-c2ab160061e0"
       },
       {
           "dn": "uid=bjensen,ou=People,dc=example,dc=com",
           "_id": "fc252fd9-b982-3ed6-b42a-c76d2546312c"
       },
       {
           "dn": "uid=jimmysmith,ou=people,dc=example,dc=com",
           "_id": "07b46858-56eb-457c-b935-cfe6ddf769c7"
       }
  ]
}
```

D.7.4. Managing Workflows Over REST

Workflow objects are exposed under the `/openidm/workflow` context path. OpenIDM provides access to the workflow module over REST, as listed in the following table.

URI	HTTP Operation	Description
/openidm/workflow/processdefinition?_queryId=*id*	GET	Lists workflow definitions based on filtering criteria
/openidm/workflow/processdefinition/*id*	GET	Returns detailed information about the specified process definition
/openidm/workflow/processdefinition/*id*/taskdefinition	GET	Returns detailed information about the task definition, when you include an *id* or a query for all IDs, `?_queryId=query-all-ids`
/openidm/workflow/processinstance?_queryId=query-all-ids	GET	Lists the available running workflows, by their ID
/openidm/workflow/processinstance/*id*	GET	Provides detailed information of a running process instance
/openidm/workflow/processinstance?_queryId=filtered-query&*filter*	GET	Returns a list of workflows, based on the specified query filter. The parameters on which this list can be filtered include: `businessKey`, `deleteReason`, `durationInMillis`, `endActivityId`, `endTime`, `processDefinitionId`, `processInstanceId`, `processVariables`, `queryVariables`, `startActivityId`, `startTime`, `startUserId`, `superProcessInstanceId`, `tenantId`, `processDefinitionResourceName`
/openidm/workflow/processinstance/history?_queryId=query-all-ids	GET	Lists running and completed workflows, by their ID
/openidm/workflow/processinstance/history?_queryId=filtered-query&*filter*	GET	Returns a list of running or completed workflows, based on the specified query filter. The parameters on which this list can be filtered include: `businessKey`, `deleteReason`, `durationInMillis`, `endActivityId`, `endTime`, `processDefinitionId`, `processInstanceId`, `processVariables`, `queryVariables`, `startActivityId`, `startTime`, `startUserId`, `superProcessInstanceId`, `tenantId`, `processDefinitionResourceName`,
/openidm/workflow/processinstance?_action=create	POST	Start a new workflow. Parameters are included in the request body.
/openidm/workflow/processinstance/*id*	DELETE	Stops a process instance
/openidm/workflow/taskinstance?_queryId=query-all-ids	GET	Lists all active tasks

URI	HTTP Operation	Description
/openidm/workflow/taskinstance?_queryId=filtered-query&*filter*	GET	Lists the tasks according to the specified filter. The parameters on which this list can be filtered include: `taskId`, `activityInstanceVariables`, `cachedElContext`, `category`, `createTime`, `delegationState`, `delegationStateString`, `deleted` (boolean), `description`, `dueDate`, `eventName`, `executionId`, `name`, `owner`, `parentTaskId`, `priority`, `processDefinitionId`, `processInstanceId`, `processVariables`, `queryVariables`, `suspended` (boolean), `suspensionState`, `taskDefinitionKey`, `taskLocalVariables`, `tenantId`, `assignee`
/openidm/workflow/taskinstance/*id*	PUT	Update task data
/openidm/workflow/taskinstance/*id*?_action=*action*	POST	Perform the specified action on that task. Parameters are included in the request body. Supported actions include `claim`, and `complete`
/openidm/workflow/taskinstance/history?_queryId=query-all-ids	GET	Lists the running and completed tasks, by their ID
/openidm/workflow/taskinstance/history?_queryId=filtered-query&*filter*	GET	Returns a list of running or completed tasks, based on the specified query filter. The parameters on which this list can be filtered include: `taskId`, `assignee`, `category`, `claimTime`, `deleteReason`, `description`, `dueDate`, `durationInMillis`, `endTime`, `executionId`, `formKey`, `name`, `owner`, `parentTaskId`, `priority`, `processDefinitionId`, `processInstanceId`, `processVariables`, `queryVariables`, `startTime`, `taskDefinitionKey`, `taskLocalVariables`, `tenantId`, `time`, `workTimeInMillis`

The following examples list the defined workflows. For a workflow to appear in this list, the corresponding workflow definition must be in the `openidm/workflow` directory.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request GET \
 "http://localhost:8080/openidm/workflow/processdefinition?_queryId=query-all-ids"
```

Depending on the defined workflows, the output will be something like the following:

```
{
"result":[ {
     "tenantId" : "",
     "candidateStarterGroupIdExpressions" : [ ],
     "candidateStarterUserIdExpressions" : [ ],
     "participantProcess" : null,
...
 } ],
     "resultCount" : 1,
     "pagedResultsCookie" : null,
     "remainingPagedResults" : -1
}
```

The following example invokes a workflow named "myWorkflow". The `foo` parameter is given the value `bar` in the workflow invocation.

```
$ curl \
 --header "Content-Type: application/json" \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 --data '{
      "_key":"contractorOnboarding",
      "foo":"bar"
  }' \
 "http://localhost:8080/openidm/workflow/processinstance?_action=create"
```

D.7.5. Managing Schedules Over REST

OpenIDM provides a scheduler service that enables you to manage and monitor scheduled jobs. For more information about the scheduler service, see "*Scheduling Tasks and Events*".

You can access the scheduler service over REST, as indicated in the following table:

URI	HTTP Operation	Description
/openidm/scheduler?_action=validateQuartzCronExpression	POST	Validates a cron expression.
/openidm/scheduler/job/*id*	PUT	Creates or updates a schedule with the specified ID.
/openidm/scheduler/job/*id*	GET	Obtains the details of the specified schedule.
/openidm/scheduler/job/*id*	DELETE	Deletes the specified schedule.
/openidm/scheduler/job?_action=create	POST	Creates a schedule with a system-generated ID.
/openidm/scheduler/job?_queryFilter=*query*	GET	Queries the existing defined schedules.
/openidm/scheduler/job?_action=listCurrentlyExecutingJobs	POST	Returns a list of the jobs that are currently running.

URI	HTTP Operation	Description
/openidm/scheduler/job?_action=pauseJobs	POST	Suspends all scheduled jobs.
/openidm/scheduler/job?_action=resumeJobs	POST	Resumes all suspended scheduled jobs.
/openidm/scheduler/trigger?_queryFilter=*query*	GET	Queries the existing triggers.
/openidm/scheduler/trigger/*id*	GET	Obtains the details of the specified trigger.
/openidm/scheduler/acquiredTriggers	GET	Returns an array of the triggers that have been acquired, per node.
/openidm/scheduler/waitingTriggers	GET	Returns an array of the triggers that have not yet been acquired.

D.7.6. Managing Scanned Tasks Over REST

OpenIDM provides a task scanning mechanism that enables you to perform a batch scan for a specified date in OpenIDM data, on a scheduled interval, and then to execute a task when this date is reached. For more information about scanned tasks, see "Scanning Data to Trigger Tasks".

OpenIDM provides REST access to the task scanner, as listed in the following table.

URI	HTTP Operation	Description
/openidm/taskscanner	GET	Lists the all scanning tasks, past and present.
/openidm/taskscanner/*id*	GET	Lists details of the given task.
/openidm/taskscanner?_action=execute&name=*name*	POST	Triggers the specified task scan run.
/openidm/taskscanner/*id*?_action=cancel	POST	Cancels the specified task scan run.

D.7.7. Accessing Log Entries Over REST

You can interact with the audit logs over REST, as shown in the following table. Queries on the audit endpoint must use `queryFilter` syntax. Predefined queries (invoked with the `_queryId` parameter) are not supported.

URI	HTTP Operation	Description
/openidm/audit/recon?_queryFilter=true	GET	Displays the reconciliation audit log
/openidm/audit/recon/*id*	GET	Reads a specific reconciliation audit log entry
/openidm/audit/recon/*id*	PUT	Creates a reconciliation audit log entry
/openidm/audit/recon?_queryFilter=/reconId+eq +"*reconId*"	GET	Queries the audit log for a particular reconciliation operation

URI	HTTP Operation	Description
/openidm/audit/recon?_queryFilter=/reconId+eq+"*reconId*"+and+situation+eq+"*situation*"	GET	Queries the reconciliation audit log for a specific reconciliation situation
/openidm/audit/sync?_queryFilter=true	GET	Displays the synchronization audit log
/openidm/audit/sync/*id*	GET	Reads a specific synchronization audit log entry
/openidm/audit/sync/*id*	PUT	Creates a synchronization audit log entry
/openidm/audit/activity?_queryFilter=true	GET	Displays the activity log
/openidm/audit/activity/*id*	GET	Returns activity information for a specific action
/openidm/audit/activity/*id*	PUT	Creates an activity audit log entry
/openidm/audit/activity?_queryFilter=transactionId=*id*	GET	Queries the activity log for all actions resulting from a specific transaction
/openidm/audit/access?_queryFilter=true	GET	Displays the full list of auditable actions.
/openidm/audit/access/*id*	GET	Displays information on the specific audit item
/openidm/audit/access/*id*	PUT	Creates an access audit log entry
/openidm/audit/authentication?_queryFilter=true	GET	Displays a complete list of authentication attempts, successful and unsuccessful
/openidm/audit/authentication?_queryFilter=/principal+eq+"*principal*"	GET	Displays the authentication attempts by a specified user
/openidm/audit?_action=availableHandlers	POST	Returns a list of audit event handlers
openidm/audit/config?_queryFilter=true	GET	Lists changes made to the configuration

D.7.8. Managing Reconciliation Operations Over REST

You can interact with the reconciliation engine over REST, as shown in the following table.

URI	HTTP Operation	Description
/openidm/recon	GET	Lists all completed reconciliation runs
/openidm/recon?_action=recon&mapping=*mapping-name*	POST	Launches a reconciliation run with the specified mapping
/openidm/recon/*id*?_action=cancel	POST	Cancels the specified reconciliation run
/openidm/system/*datastore*/account?_action=liveSync	POST	Calls a liveSync operation.

The following example runs a reconciliation action, with the mapping `systemHrdb_managedUser`, defined in the `sync.json` file.

```
$ curl \
 --header "X-OpenIDM-Username: openidm-admin" \
 --header "X-OpenIDM-Password: openidm-admin" \
 --request POST \
 "http://localhost:8080/openidm/recon?_action=recon&mapping=systemHrdb_managedUser"
```

D.7.9. Managing the Security Service Over REST

You can interact with the security service over REST, as shown in the following table:

URI	HTTP Operation	Description
/openidm/security/keystore	GET	Lists the certificates in the keystore
/openidm/security/keystore/cert/*alias*	PUT	Imports a signed certificate into the keystore
/openidm/security/keystore?_action=generateCert	POST	Generates a self-signed certificate and imports it into the keystore
/openidm/security/keystore?_action=generateCSR	POST	Generates a certificate signing request, for submission to a certificate authority
/openidm/security/truststore	GET	Lists the public keys and certificate in the truststore

For sample REST commands, see "Accessing the Security Management Service".

D.7.10. Managing the Repository Over REST

You can interact with the repository engine over REST, as shown in the following table.

URI	HTTP Operation	Description
/openidm/repo/synchronisation/deadLetterQueue/*resource*?_queryId=query-all-ids	GET	Lists any failed synchronisation records for that resource, that have been placed in the dead letter queue.
/openidm/repo/link?_queryId=query-all-ids	GET	Lists entries in the links table
/openidm/repo/internal/user?_queryId=query-all-ids	GET	Lists the internal users
/openidm/repo/internal/user/*username*	PUT	Enables you to change the username or password of an internal user
/openidm/repo?_action=updateDbCredentials	POST	Enables you to change the database username and password, in the case of an OrientDB repository

For examples of queries on the `repo/` endpoint, see "Interacting With the Repository Over REST".

D.7.11. Managing Updates Over REST

You can interact with the updates engine over REST, as shown in the following table.

URI	HTTP Operation	Description
/openidm/maintenance/update?_action=available	POST	Lists update archives in the `project-dir/openidm/bin/update/` directory
/openidm/maintenance/update?_action=preview&archive=*patch*.zip	POST	Lists file states of the current installation, relative to the *patch*.zip archive, using checksums
openidm/maintenance/update?_action=listMigrations&archive=*patch*.zip	POST	Gets a list of repository migrations for a given update type
/openidm/maintenance/update?_action=getLicense&archive=*patch*.zip	POST	Retrieves the license from the *patch*.zip archive
/openidm/maintenance/update?_action=listRepoUpdates&archive=*patch*.zip	POST	Get a list of repository update archives; use the *path* in the output for the endpoint with repo files
/openidm/maintenance/update/archives/*patch*.zip/*path*?_field=contents&_mimeType=text/plain	POST	Get files for the specific repository update, defined in the *path*.
/openidm/maintenance?_action=enable	POST	Activates maintenance mode; you should first run the commands in "Pausing Scheduled Jobs".
/openidm/maintenance?_action=disable	POST	Disables maintenance mode; you can then re-enable scheduled tasks as noted in "Resuming All Scheduled Jobs".
/openidm/maintenance?_action=status	POST	Returns current maintenance mode information
/openidm/maintenance/update?_action=update&archive=*patch*.zip	POST	Start an update with the *patch*.zip archive
/openidm/maintenance/update?_action=installed	POST	Retrieve a summary of all installed updates
/openidm/maintenance/update?_action=restart	POST	Restart OpenIDM
/openidm/maintenance/update?_action=lastUpdateId	POST	Returns the `_id` value of the last successful update
/openidm/maintenance/update?_action=markComplete&updateId=*id_string*	POST	For an update with `PENDING_REPO_UPDATES` for one or more repositories, mark as complete. Replace *id_string* with the value of `_id` for the update archive.
/openidm/maintenance/update/log/*_id*	GET	Get information about an update, by *_id* (status, dates, file action taken)
/openidm/maintenance/update/log/?_queryFilter=true	GET	Get information about all past updates, by repository

Update Status Message

Status	Description
IN_PROGRESS	Update has started, not yet complete
PENDING_REPO_UPDATES	OpenIDM update is complete, updates to the repository are pending
COMPLETE	Update is complete
FAILED	Update failed, not yet reverted

D.7.12. Managing Social ID Providers Over REST

You can manage social ID providers over REST, as shown in the following table. For more information, see "*Configuring Social ID Providers*".

URI	HTTP Operation	Description
/openidm/identityProviders	GET	Returns JSON details for all configured social ID providers
/openidm/authentication	GET	Returns JSON details for all configured social ID providers, if the `SOCIAL_PROVIDERS` module is enabled
/openidm/managed/*social ID provider*	multiple	Supports access to social ID provider information
/openidm/managed/user/*social ID provider*	GET	Supports a list of users associated with a specific social ID provider
/openidm/managed/user/*User UUID*/idps	multiple	Supports management of social ID providers by UUID

Appendix E. Scripting Reference

This appendix lists the functions supported by the script engine, the locations in which scripts can be triggered, and the variables available to scripts. For more information about scripting in OpenIDM, see "*Extending Functionality By Using Scripts*".

E.1. Function Reference

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

The following sections describe the functions supported by the script engine:

E.1.1. 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);
```

E.1.2. 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

JSON object

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 `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}]);
```

E.1.3. 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"])
```

E.1.4. 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);
```

E.1.5. 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)
```

E.1.6. openidm.query(resourceName, params, fields)

This function performs a query on the specified resource object. For more information, see "Constructing Queries".

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`, `_queryId`, or `_queryExpression`. 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". For more information about paging, see "Paging and Counting Query Results".

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 OpenIDM 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 } );
```

E.1.7. 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 enables you to 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 described in "*REST API Reference*".

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

The following list outlines the supported actions, for each resource. The actions listed here are also supported over the REST interface, and are described in detail in "*REST API Reference*".

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

patch, triggerSyncCheck

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 repository (`repo`)

command, updateDbCredentials

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 synchronization resource (`sync`)

performAction,

For example:

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

Actions supported on the reconciliation resource (`recon`)

recon, cancel

For example:

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

Actions supported on the script resource (`script`)

eval

For example:

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

Actions supported on the policy resource (`policy`)

validateObject, validateProperty

For example:

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

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

On `workflow/processinstance` create

For example:

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

On `workflow/taskinstance` claim, complete

For example:

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

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

execute, cancel

Actions supported on the external email resource (`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);
}
```

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.

E.1.8. 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/ECB/PKCS5Padding`.

alias

string

The key alias in the keystore with which to encrypt the node.

Returns

The value, encrypted with the specified cipher and key.

Throws

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

E.1.9. 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.

E.1.10. 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.

E.1.11. 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".

Returns

The value, calculated with the specified hash algorithm.

Throws

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

E.1.12. 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.

E.1.13. 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.

E.1.14. Logging Functions

OpenIDM 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 OpenIDM 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 following sections describe the logging functions available to the script engine.

E.1.14.1. 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.

E.1.14.2. 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.

E.1.14.3. 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.

E.1.14.4. 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.

E.1.14.5. 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.

E.2. Places to Trigger Scripts

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 the mapping (`conf/sync.json`) file

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.

E.3. Variables Available to Scripts

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 (see "Object-Mapping Objects"), a property object (see "Property Objects"), or a policy object (see "Policy Objects").

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

E.3.1. 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".

managed object configuration object	
Trigger	**Variable**
`onCreate`, `postCreate`	• **object**: Represents the content of the object being created. • **newObject**: Represents the object after the create operation is complete. • **context**: Context of the current request. • **resourceName**: The resource path of the object the query is performed upon. For example, if you create a managed user with ID `42f8a60e-2019-4110-a10d-7231c3578e2b`, resourceName would return `managed/user/42f8a60e-2019-4110-a10d-7231c3578e2b`. • **request**: Value of the request object.
`onUpdate`, `postUpdate` *Returns JSON object*	• **object**: Represents the object being updated. • **oldObject**: Represents the state of the object prior to invoking the update operation. • **newObject**: Reflects the changes that will be applied to the object being updated. This can continue to be modified when used with the `onUpdate` trigger. • **context**: Context of the current request. • **resourceName**: The resource path of the object the query is performed upon. • **request**: Value of the request object.
`onDelete`, `onRetrieve`, `onRead` *Returns JSON object*	• **object**: Represents the object being operated upon • **context**: Context of the current request • **resourceName**: The resource path of the object the query is performed upon • **request**: Value of the request object
`postDelete` *Returns JSON object*	• **oldObject**: Represents the deleted object • **context**: Context of the current request • **resourceName**: The resource path of the object the query is performed upon • **request**: Value of the request object

managed object configuration object	
Trigger	**Variable**
`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**: Context of the current request. • **request**: Value of the request object. • **resourceName**: An object representing the resource path the query is performed upon. • **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**: Context of the current request • **resourceName**: The resource path of the object the query is performed upon • **request**: Value of the request object

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**: Context of the current request • **resourceName**: The resource path of the object the query is performed upon • **request**: Value of the request object
`onValidate` *Returns JSON object*	• **property**: The value of the property being validated • **context**: Context of the current request

property object	
Trigger	**Variable**
	• **resourceName**: The resource path of the object the query is performed upon • **request**: Value of the request object

E.3.2. Script Triggers Defined in `sync.json`

For information about how managed objects in `sync.json` are handled and what script triggers are available, see "Object-Mapping Objects".

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**: The context related to the current sync
`linkQualifier` *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**: The context related to the current sync.
`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**: The context related to the current sync operation • **sourceId**: The object ID for the source object • **targetId**: The object ID for the target object

object-mapping object	
Trigger	**Variable**
	- **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**: The context related to the current sync - **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**: The context 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**: The context related to the current operation - **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
`validTarget` *Returns boolean*	- **target**: Represents the target object

object-mapping object	
Trigger	**Variable**
	- **linkQualifier**: The link qualifier associated with the current sync operation

property object	
Trigger	**Variable**
`condition` *Returns boolean*	- **object**: The current object being mapped. - **context**: Represents the associated context. - **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

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 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). - **recon**: Represents the reconciliation operation.

policy object	
Trigger	**Variable**
	- 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.
condition *Returns JSON object*	- **object**: Represents the source object. - **linkQualifier**: The link qualifier used for this operation (default if no other link qualifier is specified).

E.3.3. Script Triggers Defined in router.json

Trigger	Variable
onFailure	exception

Trigger	Variable
onRequest	request
onResponse	response

E.3.4. 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. For more details about writing custom endpoint scripts, see "Writing 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**: The context of the request. **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. • **queryExpression**: A string containing a native query used to query a data source 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

E.3.5. 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 when those assignments are applied to a user, or removed from a user. For more information on role assignments, see "Creating an Assignment".

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.

E.3.6. 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 OpenIDM 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()
  ```

E.3.7. The identityServer Variable

OpenIDM provides an additional variable, named identityServer, to scripts. You can use this variable in several ways. The ScriptRegistryService, described in "Validating 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 "Using Property Value Substitution In the Configuration".

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

  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 OpenIDM installation path, such as /path/to/openidm. May be superseded by an absolute path.

- getProjectLocation

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

For more information on the project location, see "Specifying the Startup Configuration".

- `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 "Specifying the Startup Configuration".

Appendix F. Router Service Reference

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

F.1. Configuration

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

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

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.

F.1.1. Filter Objects

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.

F.1.1.1. 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 repository objects.

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

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

F.1.2. 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",
                "file" : "script/router-authz.js"
            }
        },
        {
            "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
```

F.1.3. 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 (see "Function Reference").

Provides access to OpenIDM 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 "Understanding the 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 follows.

```
{
  "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.

F.2. 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"
            }
        }
    ]
}
```

F.3. Understanding the 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.

Appendix G. Embedded Jetty Configuration

OpenIDM includes an embedded Jetty web server.

To configure the embedded Jetty server, edit `openidm/conf/jetty.xml`. OpenIDM delegates most of the connector configuration to `jetty.xml`. OSGi and PAX web specific settings for connector configuration therefore do not have an effect. This lets you take advantage of all Jetty capabilities, as the web server is not configured through an abstraction that might limit some of the options.

The Jetty configuration can reference configuration properties (such as port numbers and keystore details) from OpenIDM's `boot.properties` configuration file.

G.1. Using OpenIDM Configuration Properties in the Jetty Configuration

OpenIDM exposes a `Param` class that you can use in `jetty.xml` to include OpenIDM configuration. The `Param` class exposes Bean properties for common Jetty settings and generic property access for other, arbitrary settings.

G.1.1. Accessing Explicit Bean Properties

To retrieve an explicit Bean property, use the following syntax in `jetty.xml`.

```
<Get class="org.forgerock.openidm.jetty.Param" name="<bean property name>"/>
```

For example, to set a Jetty property for keystore password:

```
<Set name="password">
    <Get class="org.forgerock.openidm.jetty.Param" name="keystorePassword"/>
</Set>
```

Also see the bundled `jetty.xml` for further examples.

The following explicit Bean properties are available.

port

Maps to `openidm.port.http`

port

Maps to `openidm.port.https`

port

Maps to `openidm.port.mutualauth`

keystoreType

Maps to `openidm.keystore.type`

keystoreProvider

Maps to `openidm.keystore.provider`

keystoreLocation

Maps to `openidm.keystore.location`

keystorePassword

Maps to `openidm.keystore.password`

keystoreKeyPassword

Maps to `openidm.keystore.key.password`, or the keystore password, if not set

truststoreLocation

Maps to `openidm.truststore.location`, or the keystore location, if not set

truststorePassword

Maps to `openidm.truststore.password`, or the keystore password, if not set

G.1.2. Accessing Generic Properties

```
<Call class="org.forgerock.openidm.jetty.Param" name="getProperty">
  <Arg>org.forgerock.openidm.some.sample.property</Arg>
</Call>
```

G.2. Jetty Default Settings

By default the embedded Jetty server uses the following settings.

- The HTTP, SSL, and Mutual Authentication ports defined in OpenIDM
- The same keystore and truststore settings as OpenIDM
- Trivial sample realm, `openidm/security/realm.properties` to add users

The default settings are intended for evaluation only. Adjust them according to your production requirements.

G.3. Registering Additional Servlet Filters

You can register generic servlet filters in the embedded Jetty server to perform additional filtering tasks on requests to or responses from OpenIDM. For example, you might want to use a servlet filter to protect access to OpenIDM with an access management product. Servlet filters are configured in files named `openidm/conf/servletfilter-name.json`. These servlet filter configuration files define the filter class, required libraries, and other settings.

A sample servlet filter configuration is provided in the `servletfilter-cors.json` file in the `/path/to/openidm/conf` directory.

The sample servlet filter configuration file is shown below:

```
{
    "classPathURLs" : [ ],
    "systemProperties" : { },
    "requestAttributes" : { },
    "scriptExtensions" : { }.
    "initParams" : {
       "allowedOrigins" : "https://localhost:&{openidm.port.https}",
       "allowedMethods" : "GET,POST,PUT,DELETE,PATCH",
       "allowedHeaders" : "accept,x-openidm-password,x-openidm-nosession,
                           x-openidm-username,content-type,origin,
                           x-requested-with",
       "allowCredentials" : "true",
       "chainPreflight" : "false"
    },
    "urlPatterns" : [
       "/*"
    ],
    "filterClass" : "org.eclipse.jetty.servlets.CrossOriginFilter"
}
```

The sample configuration includes the following properties:

`classPathURLs`

The URLs to any required classes or libraries that should be added to the classpath used by the servlet filter class `systemProperties`

Any additional Java system properties required by the filter

`requestAttributes`

The HTTP Servlet request attributes that will be set by OpenIDM when the filter is invoked. OpenIDM expects certain request attributes to be set by any module that protects access to it, so this helps in setting these expected settings.

`scriptExtensions`

Optional script extensions to OpenIDM. Currently only `"augmentSecurityContext"` is supported. A script that is defined in `augmentSecurityContext` is executed by OpenIDM after a successful authentication request. The script helps to populate the expected security context in OpenIDM. For example, the login module (servlet filter) might select to supply only the authenticated user name, while the associated roles and user ID can be augmented by the script.

Supported script types include `"text/javascript"` and `"groovy"`. The script can be provided inline (`"source":`*`script source`*) or in a file (`"file":`*`filename`*). The sample filter extends the filter interface with the functionality in the script `script/security/populateContext.js`.

`filterClass`

The servlet filter that is being registered

The following additional properties can be configured for the filter:

`httpContextId`

The HTTP context under which the filter should be registered. The default is `"openidm"`.

`servletNames`

A list of servlet names to which the filter should apply. The default is `"OpenIDM REST"`.

`urlPatterns`

A list of URL patterns to which the filter applies. The default is `["/*"]`.

`initParams`

Filter configuration initialization parameters that are passed to the servlet filter `init` method. For more information, see http://docs.oracle.com/javaee/5/api/javax/servlet/FilterConfig.html.

G.4. Disabling and Enabling Secure Protocols

Secure communications are important. To that end, the embedded Jetty web server enables a number of different protocols. To review the list of enabled protocols, use a command such as the following:

```
nmap --script ssl-enum-ciphers -p 8443 localhost
```

You can modify the list of enabled protocols in the `jetty.xml` file in the `conf/` subdirectory for your project. Based on the following excerpt, `SSLv3` and `TLSv1` are excluded from the list of enabled protocols:

```
...
    <Array id="excludedProtocols" type="java.lang.String">
       <Item>SSLv3</Item>
       <Item>TLSv1</Item>
    </Array>
...
```

Important

Disable TLSv1. Include it in the list of `excludedProtocols` in the `jetty.xml` file for your project.

Note

As noted in the following *Security Advisory*, "SSL 3.0 [RFC6101] is an obsolete and insecure protocol."

Support for the `TLSv1.0` protocol has been removed. For more information, see the following PDF: *Migrating from SSL and Early TLS* from the *PCI Security Standards Council*.

To exclude another protocol from the `Enabled` list, just add it to the `"ExcludeProtocols"` XML block. For example, if you included the following line in that XML block, your instance of Jetty would also exclude TLSv1.1:

```
<Item>TLSv1.1</Item>
```

You can reverse the process by removing the protocol from the `"ExcludeProtocols"` block.

Appendix H. Authentication and Session Module Configuration Details

This appendix includes configuration details for authentication modules described here: "Supported Authentication and Session Modules".

Authentication modules, as configured in the `authentication.json` file, include a number of properties. Except for the "OPENAM_SESSION Module Configuration Options", Those properties are listed in the following tables:

Session Module

Authentication Property	Property as Listed in the Admin UI	Description
`keyAlias`	(not shown)	Used by the Jetty Web server to service SSL requests.
`privateKeyPassword`	(not shown)	Defaults to `openidm.keystore.password` in `boot.properties`.
`keystoreType`	(not shown)	Defaults to `openidm.keystore.type` in `boot.properties`.
`keystoreFile`	(not shown)	Defaults to `openidm.keystore.location` in `boot.properties`.
`keystorePassword`	(not shown)	Defaults to `openidm.keystore.password` in `boot.properties`
`maxTokenLifeMinutes`	Max Token Life (in seconds)	Maximum time before a session is cancelled. Note the different units for the property and the UI.

Authentication Property	Property as Listed in the Admin UI	Description
`tokenIdleTimeMinutes`	Token Idle Time (in seconds)	Maximum time before an idle session is cancelled. Note the different units for the property and the UI.
`sessionOnly`	Session Only	Whether the session continues after browser restarts.

Static User Module

Authentication Property	Property as Listed in the Admin UI	Description
`enabled`	Module Enabled	Does OpenIDM use the module
`queryOnResource`	Query on Resource	Endpoint hard coded to user `anonymous`
`username`	Static User Name	Default for the static user, `anonymous`
`password`	Static User Password	Default for the static user, `anonymous`
`defaultUserRoles`	Static User Role	Normally set to `openidm-reg` for self-registration

The following table applies to several authentication modules:

`Managed User`
`Internal User`
`Client Cert`
`Passthrough`
`IWA`

The IWA module includes several Kerberos-related properties listed at the end of the table.

Common Module Properties

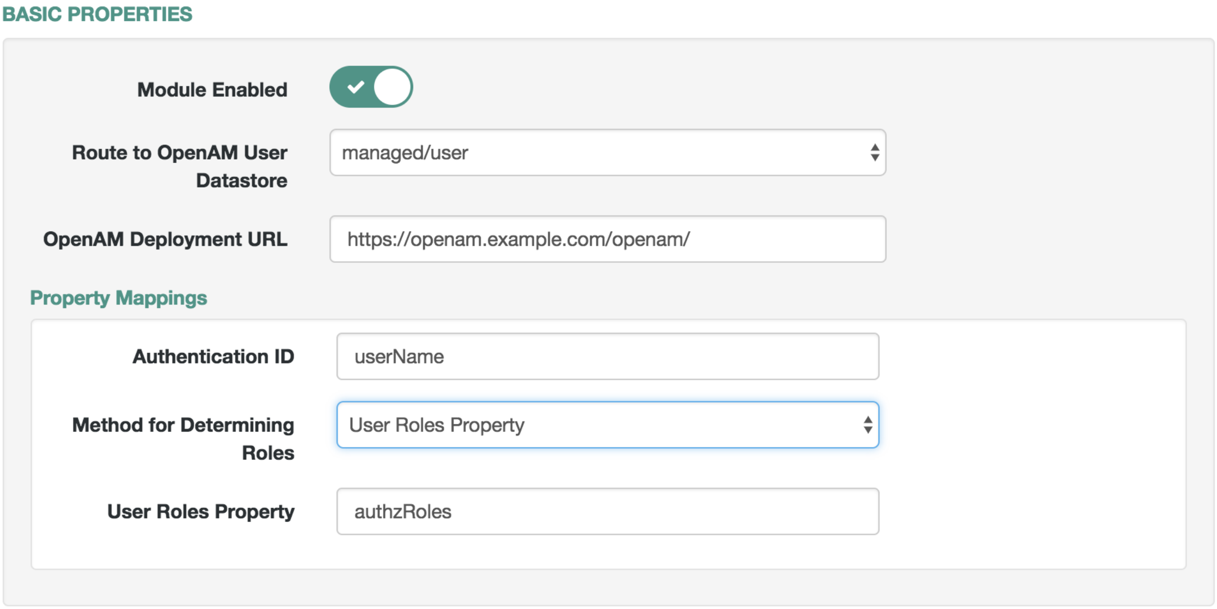
Authentication Property	Property as Listed in the Admin UI	Description
`enabled`	Module Enabled	Does OpenIDM use the module
`queryOnResource`	Query on Resource	Endpoint to query
`queryId`	Use Query ID	A defined `queryId` searches against the `queryOnResource` endpoint. An undefined `queryId` against `queryOnResource` with `action=reauthenticate`
`defaultUserRoles`	Default User Roles	Normally blank for managed users
`authenticationId`	Authentication ID	Defines how account credentials are derived from a `queryOnResource` endpoint

Authentication Property	Property as Listed in the Admin UI	Description
`userCredential`	User Credential	Defines how account credentials are derived from a `queryOnResource` endpoint; if required, typically `password` or `userPassword`
`userRoles`	User Roles	Defines how account roles are derived from a `queryOnResource` endpoint
`groupMembership`	Group Membership	Provides more information for calculated roles
`groupRoleMapping`	Group Role Mapping	Provides more information for calculated roles
`groupComparisonMethod`	Group Comparison Method	Provides more information for calculated roles
`managedUserLink`	Managed User Link	Applicable mapping (Passthrough module only)
`augmentSecurityContext`	Augment Security Context	Includes a script that is executed only after a successful authentication request.
`servicePrincipal`	Kerberos Service Principal	(IWA only) For more information, see "Configuring IWA Authentication"
`keytabFileName`	Keytab File Name	(IWA only) For more information, see "Configuring IWA Authentication"
`kerberosRealm`	Kerberos Realm	(IWA only) For more information, see "Configuring IWA Authentication"
`kerberosServerName`	Kerberos Server Name	(IWA only) For more information, see "Configuring IWA Authentication"

H.1. OPENAM_SESSION Module Configuration Options

The `OPENAM_SESSION` module uses OpenAM-based authentication to protect an full stack deployment.

The options shown in the screen are subdivided into basic and advanced properties. You may need to choose Advanced Properties to review those details.



BASIC PROPERTIES

Module Enabled ✓

Route to OpenAM User Datastore: managed/user

OpenAM Deployment URL: https://openam.example.com/openam/

Property Mappings

Authentication ID: userName

Method for Determining Roles: User Roles Property

User Roles Property: authzRoles

The following table describes the label that you see in the Admin UI, the default value (if any), a brief description, and the associated configuration file. If you need the property name, examine the `authentication.json` configuration file.

OPENAM_SESSION Module Basic Properties

Admin UI Label	Default	Description
Module Enabled	enabled	Whether to enable the module
Route to OpenAM User Datastore	managed/user	External repository with OpenAM Data Store Information
OpenAM Deployment URL	https://openam.example.com/openam/	Full URL of the deployed instance of OpenAM
Authentication ID	userName	User identifier
Method for Determining Roles	User Roles Property	May also be Group Membership
User Roles Property (if selected)	authzRoles	Authorization Roles

OPENAM_SESSION Module Advanced Properties

Admin UI Label	Default	Description
Default User Roles	openidm-authorized	OpenIDM assigns such roles to the security context of a user
OpenAM SSO Token Cookie Name	blank	The standard OpenAM SSO Token Cookie name is `iPlanetDirectoryPro`

Appendix I. Social ID Provider Configuration Details

This appendix includes a list of configuration details for each supported social identity provider.

I.1. Google Social ID Provider Configuration Details

You can set up the Google social identity provider either through the Admin UI or in the `identityProvider-google.json` file in your project's `conf/` subdirectory. The following table includes the information shown in the Admin UI Google Provider pop-up window, along with associated information in the `identityProvider-google.json` file.

OpenIDM generates the `identityProvider-google.json` file only when you configure and enable the Google social identity provider in the Admin UI.

Google Social ID Provider Configuration Properties

Property (UI)	Property (JSON file)	Description
Client ID	`client_id`	The client identifier for your Google Identity Platform project
Client Secret	`client_secret`	Used with the Client ID to access the configured Google API
Scope	`scope`	An array of strings that allows access to user data; see Google's documentation on *Authorization Scopes*

Property (UI)	Property (JSON file)	Description
Authorization Endpoint	`authorization_endpoint`	Per *RFC 6749*, "used to interact with the resource owner and obtain an authorization grant". For Google's implementation, see *Forming the URL*.
Token Endpoint	`token_endpoint`	Endpoint that receives a one-time authorization grant, and returns an access and ID token
User Info Endpoint	`userinfo_endpoint`	Endpoint that receives an access token, and returns information about the user
Well-Known Endpoint	`well-known`	Access URL for Google's *Discovery Document*
Sign-In Button HTML	`icon`	Image location and text for the sign-in button
Not in the Admin UI	`name`	Name of the Social ID provider
Not in the Admin UI	`type`	Authentication module
Not in the Admin UI	`authenticationId`	Authentication identifier, as returned from the User Info Endpoint for each Social ID Provider
Not in the Admin UI	`propertyMap`	Mapping between Google and OpenIDM

I.2. LinkedIn Social ID Provider Configuration Details

You can set up the LinkedIn social identity provider either through the Admin UI or in the `identityProvider-linkedIn.json` file in your project's `conf/` subdirectory. The following table includes the information shown in the Admin UI LinkedIn Provider pop-up window, along with associated information in the `identityProvider-linkedIn.json` file.

OpenIDM generates the `identityProvider-linkedIn.json` file only when you configure and enable the Linkedin social identity provider in the Admin UI.

Linkedin Social ID Provider Configuration Properties

Property (UI)	Property (JSON file)	Description
Client ID	`client_id`	The client identifier for your Linkedin Application
Client Secret	`client_secret`	Used with the Client ID to access the applicable Linkedin API
Scope	`scope`	An array of strings that allows access to user data; see Linkedin's documentation on *Basic Profile Fields*.
Authorization Endpoint	`authorization_endpoint`	Per *RFC 6749*, "used to interact with the resource owner and obtain an authorization grant". For Linkedin's implementation, see their documentation on *Authenticating with OAuth 2.0*.
Token Endpoint	`token_endpoint`	Endpoint that receives a one-time authorization code, and returns an access token. For Linkedin's

Property (UI)	Property (JSON file)	Description
		implementation, see their documentation on *Authenticating with OAuth 2.0*.
User Info Endpoint	`userinfo_endpoint`	Endpoint that transmits scope-related fields through Linkedin's API. The default endpoint includes the noted field properties in parenthesis, as defined in Linkedin's documentation on *Basic Profile Fields*.
Well-Known Endpoint	`well-known`	Not used for Linkedin
Sign-In Button HTML	`icon`	Image location and text for the sign-in button
Not in the Admin UI	`name`	Name of the Social ID provider
Not in the Admin UI	`type`	Authentication module
Not in the Admin UI	`authenticationId`	Authentication identifier, as returned from the User Info Endpoint for each Social ID Provider
Not in the Admin UI	`propertyMap`	Mapping between LinkedIn and OpenIDM

I.3. Facebook Social ID Provider Configuration Details

You can set up the Facebook social identity provider either through the Admin UI or in the `identityProvider-facebook.json` file in your project's `conf/` subdirectory. The following table includes the information shown in the Admin UI Facebook Provider pop-up window, along with associated information in the `identityProvider-facebook.json` file.

OpenIDM generates the `identityProvider-facebook.json` file only when you configure and enable the Facebook social identity provider in the Admin UI. Alternatively, you can create that file manually.

Facebook Social ID Provider Configuration Properties

Property (UI)	Property (JSON file)	Description
App ID	`client_id`	The client identifier for your Facebook App
App Secret	`client_secret`	Used with the App ID to access the applicable Facebook API
Scope	`scope`	An array of strings that allows access to user data; see Facebook's *Permissions Reference* Documentation.
Authorization Endpoint	`authorization_endpoint`	For Facebook's implementation, see their documentation on how they *Manually Build a Login Flow*.
Token Endpoint	`token_endpoint`	Endpoint that receives a one-time authorization code, and returns an access token. For Facebook's implementation, see their documentation on how they *Manually Build a Login Flow*.

Property (UI)	Property (JSON file)	Description
User Info Endpoint	`userinfo_endpoint`	Endpoint that transmits scope-related fields through Linkedin's API. The default endpoint includes the noted field properties as a list, as defined in Facebook's *Permissions Reference*.
Sign-In Button HTML	`icon`	Image location and text for the sign-in button
Not in the Admin UI	`name`	Name of the Social ID provider
Not in the Admin UI	`type`	Authentication module
Not in the Admin UI	`authenticationId`	Authentication identifier, as returned from the User Info Endpoint for each Social ID Provider
Not in the Admin UI	`propertyMap`	Mapping between Facebook and OpenIDM

I.4. Custom Social ID Provider Configuration Details

When you set up a custom social identity provider, starting with "Preparing OpenIDM For a Custom Social ID Provider", you'll see configuration details in the `identityProviders.json` file, in your project's `conf/` subdirectory. The following table includes the information shown in the relevant Admin UI pop-up window.

OpenIDM generates the `identityProvider-custom.json` file only when you configure and enable the custom social identity provider in the Admin UI. Alternatively, you can create that file manually.

Custom Social ID Provider Configuration Properties

Property (UI)	Property (JSON file)	Description
Client ID	`client_id`	The client identifier for your social ID provider
Client Secret	`client_secret`	Used with the Client ID
Scope	`scope`	An array of strings that allows access to user data; varies by provider.
Authorization Endpoint	`authorization_endpoint`	Every social ID provider should have an authorization endpoint to authenticate end users.
Token Endpoint	`token_endpoint`	Endpoint that receives a one-time authorization code, and returns an access token.
User Info Endpoint	`userinfo_endpoint`	Endpoint that transmits scope-related fields.
Sign-In Button HTML	`icon`	Image location and text for the sign-in button
Not in the Admin UI	`name`	Name of the Social ID provider
Not in the Admin UI	`type`	Authentication module
Not in the Admin UI	`authenticationId`	Authentication identifier, as returned from the User Info Endpoint for each Social ID Provider

Property (UI)	Property (JSON file)	Description
Not in the Admin UI	`propertyMap`	Mapping between the social ID provider and OpenIDM

Appendix J. Audit Log Reference

J.1. Audit Log Schema

The following tables depict the schema for the six audit event topics. For the JSON audit event handler, each audit topic is logged to a distinct JSON file, with the topic in the filename. Files are created in the `openidm/audit` directory by default:

- `access.audit.json`: see "Access Event Topic Properties"
- `activity.audit.json`: see "Activity Event Topic Properties"
- `authentication.audit.json`: see "Authentication Event Topic Properties"
- `config.audit.json`: see "Configuration Event Topic Properties"
- `recon.audit.json`: see "Reconciliation Event Topic Properties"
- `sync.audit.json`: see "Synchronization Event Topic Properties"

You can parse the files in the `openidm/audit` directory using a JSON processor, such as `jq`. For example:

```
$ tail -f authentication.audit.json | jq .
{
  "context": {
    "component": "repo/internal/user",
    "roles": [
      "openidm-admin",
      "openidm-authorized"
    ],
    "ipAddress": "0:0:0:0:0:0:0:1",
    "id": "openidm-admin",
    "moduleId": "INTERNAL_USER"
  },
  "entries": [
    {
      "moduleId": "JwtSession",
      "result": "SUCCESSFUL",
      "info": {
        "org.forgerock.authentication.principal": "openidm-admin"
      }
    }
  ],
  "principal": [
    "openidm-admin"
  ]
,
...
```

The JSON properties that correspond to each audit topic are described in the following tables:

J.1.1. Audit Event Topics

Reconciliation Event Topic Properties

Event Property	Description
`_id`	UUID for the message object, such as `"0419d364-1b3d-4e4f-b769-555c3ca098b0"`
`transactionId`	The UUID of the transaction; you may see the same ID in different audit event topics.
`timestamp`	The time that the message was logged, in UTC format; for example `"2015-05-18T08:48:00.160Z"`
`eventName`	The name of the audit event: `recon` for this log
`userId`	User ID
`trackingIds`	A unique value for an object being tracked
`action`	Reconciliation action, depicted as a CREST action. For more information, see "Synchronization Actions"
`exception`	The stack trace of the exception
`linkQualifier`	The link qualifier applied to the action; For more information, see "Mapping a Single Source Object to Multiple Target Objects"

Event Property	Description
mapping	The name of the mapping used for the synchronization operation, defined in conf/sync.json.
message	Description of the synchronization action
messageDetail	Details from the synchronization run, shown as CREST output
situation	The synchronization situation described in "How OpenIDM Assesses Synchronization Situations"
sourceObjectId	The object ID on the source system, such as managed/user/jdoe
status	Reconciliation result status, such as SUCCESS or FAILURE
targetObjectId	The object ID on the target system, such as system/xmlfile/account/bjensen
reconciling	What is currently being reconciled, source for the first phase, target for the second phase.
ambiguousTargetObjectIds	When the situation is AMBIGUOUS or UNQUALIFIED, and OpenIDM cannot distinguish between more than one target object, the object IDs are logged, to help figure out what was ambiguous.
reconAction	The reconciliation action, typically recon or null
entryType	The type of reconciliation log entry, such as start, entry, or summary.
reconId	UUID for the reconciliation operation

Synchronization Event Topic Properties

Event Property	Description
_id	UUID for the message object, such as "0419d364-1b3d-4e4f-b769-555c3ca098b0"
transactionId	The UUID of the transaction; you may see the same ID in different audit event topics.
timestamp	The time that the message was logged, in UTC format; for example "2015-05-18T08:48:00.160Z"
eventName	The name of the audit event: sync for this log
userId	User ID
trackingIds	A unique value for an object being tracked
action	Synchronization action, depicted as a CREST action. For more information, see "Synchronization Actions"
exception	The stack trace of the exception
linkQualifier	The link qualifier applied to the action; For more information, see "Mapping a Single Source Object to Multiple Target Objects"
mapping	The name of the mapping used for the synchronization operation, defined in conf/sync.json.

Event Property	Description
`message`	Description of the synchronization action
`messageDetail`	Details from the reconciliation run, shown as CREST output
`situation`	The synchronization situation described in "How OpenIDM Assesses Synchronization Situations"
`sourceObjectId`	The object ID on the source system, such as `managed/user/jdoe`
`status`	Reconciliation result status, such as SUCCESS or FAILURE
`targetObjectId`	The object ID on the target system, such as `uid=jdoe,ou=People,dc=example,dc=com`

J.1.2. Commons Audit Event Topics

Access Event Topic Properties

Event Property	Description
`_id`	UUID for the message object, such as `"0419d364-1b3d-4e4f-b769-555c3ca098b0"`
`timestamp`	The time that OpenIDM logged the message, in UTC format; for example `"2015-05-18T08:48:00.160Z"`
`eventName`	The name of the audit event: `access` for this log
`transactionId`	The UUID of the transaction; you may see the same transaction for the same event in different audit event topics
`userId`	User ID
`trackingIds`	A unique value for an object being tracked
`server.ip`	IP address of the OpenIDM server
`server.port`	Port number used by the OpenIDM server
`client.ip`	Client IP address
`client.port`	Client port number
`request.protocol`	Protocol for request, typically CREST
`request.operation`	Typically a CREST operation
`request.detail`	Typically details for an ACTION request
`http.request.secure`	Boolean for request security
`http.request.method`	HTTP method requested by the client
`http.request.path`	Path of the HTTP request
`http.request.queryParameters`	Parameters sent in the HTTP request, such as a key/value pair
`http.request.headers`	HTTP headers for the request (optional)
`http.request.cookies`	HTTP cookies for the request (optional)

Event Property	Description
`http.response.headers`	HTTP response headers (optional)
`response.status`	Normally, SUCCESSFUL, FAILED, or null
`response.statusCode`	SUCCESS in `response.status` leads to a null `response.statusCode`; FAILURE leads to a 400-level error
`response.detail`	Message associated with `response.statusCode`, such as Not Found or Internal Server Error
`response.elapsedTime`	Time to execute the access event
`response.elapsedTimeUnits`	Units for response time
`roles`	OpenIDM roles associated with the request

Activity Event Topic Properties

Event Property	Description
`_id`	UUID for the message object, such as `"0419d364-1b3d-4e4f-b769-555c3ca098b0"`
`timestamp`	The time that OpenIDM logged the message, in UTC format; for example `"2015-05-18T08:48:00.160Z"`
`eventName`	The name of the audit event: `activity` for this log
`transactionId`	The UUID of the transaction; you may see the same transaction for the same event in different audit event topics.
`userId`	User ID
`trackingIds`	A unique value for the object being tracked
`runAs`	User to run the activity as; may be used in delegated administration
`objectId`	Object identifier, such as `/managed/user/jdoe`
`operation`	Typically a CREST operation
`before`	JSON representation of the object prior to the activity
`after`	JSON representation of the object after the activity
`changedFields`	Fields that were changed, based on "Watched Fields: Defining Fields to Monitor"
`revision`	Object revision number
`status`	Result, such as SUCCESS
`message`	Human readable text about the action
`passwordChanged`	True/False entry on changes to the password

Authentication Event Topic Properties

Event Property	Description
`_id`	UUID for the message object, such as `"0419d364-1b3d-4e4f-b769-555c3ca098b0"`
`timestamp`	The time that OpenIDM logged the message, in UTC format; for example `"2015-05-18T08:48:00.160Z"`
`eventName`	The name of the audit event: `authentication` for this log
`transactionId`	The UUID of the transaction; you may see the same transaction for the same event in different audit event topics.
`userId`	User ID
`trackingIds`	A unique value for an object being tracked
`result`	The result of the transaction, either "SUCCESSFUL", or "FAILED"
`principal`	An array of the accounts used to authenticate, such as ["openidm-admin"]
`context`	The complete security context of the authentication operation, including the authenticating ID, targeted endpoint, authentication module, any roles applied, and the IP address from which the authentication request was made.
`entries`	The JSON representation of the authentication session

Configuration Event Topic Properties

Event Property	Description
`_id`	UUID for the message object, such as `"0419d364-1b3d-4e4f-b769-555c3ca098b0"`
`timestamp`	The time that OpenIDM logged the message, in UTC format; for example `"2015-05-18T08:48:00.160Z"`
`eventName`	The name of the audit event: `config` for this log
`transactionId`	The UUID of the transaction; you may see the same transaction for the same event in different audit event topics.
`userId`	User ID
`trackingIds`	A unique value for an object being tracked
`runAs`	User to run the activity as; may be used in delegated administration
`objectId`	Object identifier, such as `ui`
`operation`	Typically a CREST operation
`before`	JSON representation of the object prior to the activity
`after`	JSON representation of the object after to the activity

Event Property	Description
`changedFields`	Fields that were changed, based on "Watched Fields: Defining Fields to Monitor"
`revision`	Object revision number

J.2. Audit Event Handler Configuration

To configure an audit event handler, set the `config` properties for that handler in your project's `conf/audit.json` file.

To configure these properties from the Admin UI, click Configure > System Preferences > Audit, and click the edit icon associated with your event handler.

The tables shown in this section reflect the order in which properties are shown in the Admin UI. That order differs when you review the properties in your project's `audit.json` file.

Common Audit Event Handler Property Configuration

UI Label / Text	`audit.json` File Label	Description
Name	`name`	`config` sub-property. The name of the audit event handler
Audit Events	`topics`	`config` sub-property; the list of audit topics that are logged by this audit event handler, for example, `access`, `activity`, and `config`
Use for Queries	`handlerForQueries`	Specifies whether this audit event handler manages the queries on audit logs
Enabled	`enabled`	`config` sub-property; specifies whether the audit event handler is enabled. An audit event handler can be configured, but disabled, in which case it will not log events.
n/a	`config`	The JSON object used to configure the handler; includes several sub-properties
Shown only in `audit.json`	`class`	The class name in the Java file(s) used to build the handler

The following table lists the configurable properties specific to the JSON audit event handler:

JSON Audit Event Handler `config` Properties

Property	Description
`fileRotation`	Groups the file rotation configuration parameters.
`rotationEnabled`	Specifies whether file rotation is enabled. Boolean, true or false.
`maxFileSize`	The maximum size of an audit file, in bytes, before rotation is triggered.

Property	Description
`rotationFilePrefix`	The prefix to add to the start of an audit file name when it is rotated.
`rotationTimes`	Specifies a list of times at which file rotation should be triggered. The times must be provided as durations, offset from midnight. For example, a list of `10 minutes, 20 minutes, 30 minutes` will cause files to rotate at 10, 20 and 30 minutes after midnight.
`rotationFileSuffix`	The suffix appended to rotated audit file names. This suffix should take the form of a timestamp, in simple date format. The default suffix format, if none is specified, is `-yyyy.MM.dd-HH.mm.ss`.
`rotationInterval`	The interval to trigger a file rotation, expressed as a duration. For example, `5 seconds`, `5 minutes`, `5 hours`. A value of `0` or `disabled` disables time-based file rotation. Note that you can specify a list of `rotationTimes` and a `rotationInterval`. The audit event handler checks all rotation and retention policies on a periodic basis, and assesses whether each policy should be triggered at the current time, for a particular audit file. The first policy to meet the criteria is triggered.
`fileRetention`	Groups the file retention configuration parameters. The retention policy specifies how long audit files remain on disk before they are automatically deleted.
`maxNumberOfHistoryFiles`	The maximum number of historical audit files that can be stored. If the total number of audit files exceed this maximum, older files are deleted. A value of `-1` disables purging of old log files.
`maxDiskSpaceToUse`	The maximum disk space, in bytes, that can be used for audit files. If the total space occupied by the audit files exceed this maximum, older files are deleted. A negative or zero value indicates that this policy is disabled, that is, that unlimited disk space can be used for historical audit files.
`minFreeSpaceRequired`	The minimum free disk space, in bytes, required on the system that houses the audit files. If the free space drops below this minimum, older files are deleted. A negative or zero value indicates that this policy is disabled, that is, that no minimum space requirements apply.
`rotationRetentionCheckInterval`	Interval for periodically checking file rotation and retention policies. The interval must be a duration, for example, `5 seconds`, `5 minutes`, or `5 hours`.
`logDirectory`	Directory with JSON audit files
`elasticsearchCompatible`	Enable ElasticSearch JSON format compatibility. Boolean, true or false. Set this property to `true`, for example, if you are using Logstash to feed into ElasticSearch. For more information, see the ElasticSearch documentation.
`buffering`	Configuration for event buffering
`maxSize`	The maximum number of events that can be buffered (default/minimum: 100000)
`writeInterval`	The delay after which the file-writer thread is scheduled to run after encountering an empty event buffer (units of 'ms' are recommended). Default: 100 ms.

The following table lists the configurable properties specific to the CSV audit event handler:

CSV Audit Event Handler `config` Properties

UI Label / Text	`audit.json` File Label	Description
File Rotation	`fileRotation`	Groups the file rotation configuration parameters.
rotationEnabled	`rotationEnabled`	Specifies whether file rotation is enabled. Boolean, true or false.
maxFileSize	`maxFileSize`	The maximum size of an audit file, in bytes, before rotation is triggered.
rotationFilePrefix	`rotationFilePrefix`	The prefix to add to the start of an audit file name when it is rotated.
Rotation Times	`rotationTimes`	Specifies a list of times at which file rotation should be triggered. The times must be provided as durations, offset from midnight. For example, a list of `10 minutes, 20 minutes, 30 minutes` will cause files to rotate at 10, 20 and 30 minutes after midnight.
File Rotation Suffix	`rotationFileSuffix`	The suffix appended to rotated audit file names. This suffix should take the form of a timestamp, in simple date format. The default suffix format, if none is specified, is `-yyyy.MM.dd-HH.mm.ss`.
Rotation Interval	`rotationInterval`	The interval to trigger a file rotation, expressed as a duration. For example, `5 seconds`, `5 minutes`, `5 hours`. A value of `0` or `disabled` disables time-based file rotation. Note that you can specify a list of `rotationTimes` and a `rotationInterval`. The audit event handler checks all rotation and retention policies on a periodic basis, and assesses whether each policy should be triggered at the current time, for a particular audit file. The first policy to meet the criteria is triggered.
File Retention	`fileRetention`	Groups the file retention configuration parameters. The retention policy specifies how long audit files remain on disk before they are automatically deleted.
Maximum Number of Historical Files	`maxNumberOfHistoryFiles`	The maximum number of historical audit files that can be stored. If the total number of audit files exceed this maximum, older files are deleted. A value of `-1` disables purging of old log files.
Maximum Disk Space	`maxDiskSpaceToUse`	The maximum disk space, in bytes, that can be used for audit files. If the total space occupied by the audit files exceed this maximum, older files are deleted. A negative or zero value indicates that this policy is disabled, that is, that unlimited disk space can be used for historical audit files.
Minimum Free Space Required	`minFreeSpaceRequired`	The minimum free disk space, in bytes, required on the system that houses the audit files. If the free space drops below this minimum, older files are

UI Label / Text	audit.json File Label	Description
		deleted. A negative or zero value indicates that this policy is disabled, that is, that no minimum space requirements apply.
rotationRetentionCheckInterval	rotationRetentionCheckInterval	Interval for periodically checking file rotation and retention policies. The interval must be a duration, for example, 5 seconds, 5 minutes, or 5 hours.
Log Directory	logDirectory	Directory with CSV audit files
CSV Output Formatting	formatting	
quoteChar	quoteChar	Formatting: Character used around a CSV field
delimiterChar	delimiterChar	Formatting: Character between CSV fields
End of Line Symbols	endOfLineSymbols	Formatting: end of line symbol, such as \n or \r
Security: CSV Tamper Evident Configuration	security	Uses keystore-based signatures
Enabled	enabled	CSV Tamper Evident Configuration: true or false
Filename	filename	CSV Tamper Evident Configuration: Path to the Java keystore
Password	password	CSV Tamper Evident Configuration: Password for the Java keystore
Keystore Handler	keystoreHandlerName	CSV Tamper Evident Configuration: Keystore name. The value of this property must be openidm. This is the name that the audit service provides to the ForgeRock Common Audit Framework for the configured OpenIDM keystore.
Signature Interval	signatureInterval	CSV Tamper Evident Configuration: Signature generation interval. Default = 1 hour. Units described in "Minimum Admin UI CSV Audit Handler Configuration Requirements".
Buffering	buffering	Configuration for optional event buffering
enabled	enabled	Buffering: true or false
autoFlush	autoFlush	Buffering: avoids flushing after each event

Except for the common properties shown in "Common Audit Event Handler Property Configuration", the Repository and Router audit event handlers share one unique property: resourcePath:

```
{
    "class" : "org.forgerock.openidm.audit.impl.RouterAuditEventHandler",
    "config" : {
        "name" : "router",
        "topics" : [ "access", "activity", "recon", "sync", "authentication", "config" ],
        "resourcePath" : "system/auditdb"
    }
  },
```

Repository / Router Audit Event Handler Unique `config` Properties

UI Label / Text	`audit.json` File Label	Description
resourcePath	`resourcePath`	Path to the repository resource

Note

Take care when reading JMS properties in the `audit.json` file. They include the standard ForgeRock audit event topics, along with JMS-unique topics:

JMS Audit Event Handler Unique `config` Properties

UI Label / Text	`audit.json` File Label	Description
Delivery Mode	`deliveryMode`	For messages from a JMS provider; may be `PERSISTENT` or `NON_PERSISTENT`
Session Mode	`sessionMode`	Acknowledgement mode, in sessions without transactions. May be `AUTO`, `CLIENT`, or `DUPS_OK`.
Batch Configuration Settings	`batchConfiguration`	Options when batch messaging is enabled
Batch Enabled	`batchEnabled`	Boolean for batch delivery of audit events
Capacity	`capacity`	Maximum event count in the batch queue; additional events are dropped
Thread Count	`threadCount`	Number of concurrent threads that pull events from the batch queue
Maximum Batched Events	`maxBatchedEvents`	Maximum number of events per batch
Insert Timeout (Seconds)	`insertTimeoutSec`	Waiting period (seconds) for available capacity, when a new event enters the queue
Polling Timeout (Seconds)	`pollTimeoutSec`	Worker thread waiting period (seconds) for the next event, before going idle
Shutdown Timeout (Seconds)	`shutdownTimeoutSec`	Application waiting period (seconds) for worker thread termination
JNDI Configuration	`jndiConfiguration`	Java Naming and Directory Interface (JNDI) Configuration Settings
JNDI Context Properties	`contextProperties`	Settings to populate the JNDI initial context with
JNDI Context Factory	`java.naming.factory.initial`	Initial JNDI context factory, such as `com.tibco.tibjms.naming.TibjmsInitialContextFactory`
JNDI Provider URL	`java.naming.provider.url`	Depends on provider; options include `tcp://localhost:61616` and `tibjmsnaming://192.168.1.133:7222`
JNDI Topic	`topic.audit`	Relevant JNDI topic; default=audit

UI Label / Text	`audit.json` File Label	Description
JNDI Topic Name	`topicName`	JNDI lookup name for the JMS topic
Connection Factory	`connectionFactoryName`	JNDI lookup name for the JMS connection factory

The ForgeRock Syslog audit event handler is based on a widely-used logging protocol. When you configure Syslog on OpenIDM, you will see many of the following properties in the UI and your project's `audit.json` file.

Syslog Audit Event Handler Unique `config` Properties

UI Label / Text	`audit.json` File Label	Description
protocol	`protocol`	Transport protocol for Syslog messages; may be `TCP` or `UDP`
host	`host`	Host name or IP address of the receiving Syslog server
port	`port`	The TCP/IP port number of the receiving Syslog server
connectTimeout	`connectTimeout`	Timeout for connecting to the Syslog server (seconds)
facility	`facility`	Options shown in the Admin UI, `KERN`, `USER`, `MAIL`, `DAEMON`, `AUTH`, `SYSLOG`, `LPR`, `NEWS`, `UUCP`, `CRON`, `AUTPRIV`, `FTP`, `NTP`, `LOGAUDIT`, `LOGALERT`, `CLOCKD`, `LOCAL0`, `LOCAL1`, `LOCAL2`, `LOCAL3`, `LOCAL4`, `LOCAL5`, `LOCAL6`, `LOCAL7` correspond directly to facility values shown in RFC 5424, *The Syslog Protocol*.
SeverityFieldMappings	`severityFieldMappings`	Sets the correspondence between audit event fields and Syslog severity values
topic	`topic`	Severity Field Mappings: the audit event topic to which the mapping applies
field	`field`	Severity Field Mappings: the audit event field to which the mapping applies; taken from the JSON schema for the audit event content
Value Mappings	`valueMappings`	Severity Field Mappings: The map of audit event values to Syslog severities. Syslog severities may be: EMERGENCY, ALERT, CRITICAL, ERROR, WARNING, NOTICE, INFORMATIONAL, or DEBUG, in descending order of importance
Buffering	`buffering`	Disabled by default; all messages written immediately to the log

The Elasticsearch audit event handler is relatively complex, with `config` subcategories for `connection`, `indexMapping`, `buffering`, and `topics`.

Elasticsearch Audit Event Handler Unique `config` Properties

UI Label / Text	`audit.json` File Label	Description
Connection	`connection`	Elasticsearch audit event handler
useSSL	`useSSL`	Connection: Use SSL/TLS to connect to Elasticsearch
host	`host`	Connection: Hostname or IP address of Elasticsearch (default: localhost)
port	`port`	Connection: Port used by Elasticsearch (default: 9200)
username	`username`	Connection: Username when Basic Authentication is enabled via Elasticsearch Shield
password	`password`	Connection: Password when Basic Authentication is enabled via Elasticsearch Shield
Index Mapping	`indexMapping`	Defines how an audit event and its fields are stored and indexed
indexName	`indexName`	Index Mapping: Index Name (default=audit). Change if 'audit' conflicts with an existing Elasticsearch index
Buffering	`buffering`	Configuration for buffering events and batch writes (increases write-throughput)
enabled	`enabled`	Buffering: recommended
maxSize	`maxSize`	Buffering: Fixed maximum number of events that can be buffered (default: 10000)
Write Interval	`writeInterval`	The delay after which the file-writer thread is scheduled to run after encountering an empty event buffer (units of 'ms' are recommended). Default: 100 ms.
maxBatchedEvents	`maxBatchedEvents`	Buffering: Maximum number of events per batch-write to Elasticsearch for each Write Interval (default: 500)

The following table lists the configurable properties specific to the Splunk audit event handler:

Splunk Audit Event Handler `config` Properties

Property	Description
`useSSL`	Specifies whether OpenIDM should connect to the Splunk instance over SSL. Boolean, true or false.
`host`	The hostname or IP address of the Splunk instance. If no hostname is specified, `localhost` is assumed.
`port`	The dedicated Splunk port for HTTP input. Default: 8088.
`buffering`	Configuration for event buffering

Property	Description
`maxSize`	The maximum number of events that can be buffered. Default/minimum: 10000.
`writeInterval`	The delay after which the file-writer thread is scheduled to run after encountering an empty event buffer (units of 'ms' or 's' are recommended). Default: 100 ms.
`maxBatchedEvents`	The maximum number of events per batch-write to Splunk for each Write Interval. Default: 500.
`authzToken`	The authorization token associated with the Splunk configured HTTP event collector.

Appendix K. Release Levels & Interface Stability

This appendix includes ForgeRock definitions for product release levels and interface stability.

K.1. ForgeRock Product Release Levels

ForgeRock defines Major, Minor, Maintenance, and Patch product release levels. The release level is reflected in the version number. The release level tells you what sort of compatibility changes to expect.

Release Level Definitions

Release Label	Version Numbers	Characteristics
Major	Version: x[.0.0] (trailing 0s are optional)	• Bring major new features, minor features, and bug fixes • Can include changes even to Stable interfaces • Can remove previously Deprecated functionality, and in rare cases remove Evolving functionality that has not been explicitly Deprecated • Include changes present in previous Minor and Maintenance releases
Minor	Version: x.y[.0] (trailing 0s are optional)	• Bring minor features, and bug fixes

Release Label	Version Numbers	Characteristics
		• Can include backwards-compatible changes to Stable interfaces in the same Major release, and incompatible changes to Evolving interfaces • Can remove previously Deprecated functionality • Include changes present in previous Minor and Maintenance releases
Maintenance, Patch	Version: x.y.z[.p] The optional `.p` reflects a Patch version.	• Bring bug fixes • Are intended to be fully compatible with previous versions from the same Minor release

K.2. ForgeRock Product Interface Stability

ForgeRock products support many protocols, APIs, GUIs, and command-line interfaces. Some of these interfaces are standard and very stable. Others offer new functionality that is continuing to evolve.

ForgeRock acknowledges that you invest in these interfaces, and therefore must know when and how ForgeRock expects them to change. For that reason, ForgeRock defines interface stability labels and uses these definitions in ForgeRock products.

Interface Stability Definitions

Stability Label	Definition
Stable	This documented interface is expected to undergo backwards-compatible changes only for major releases. Changes may be announced at least one minor release before they take effect.
Evolving	This documented interface is continuing to evolve and so is expected to change, potentially in backwards-incompatible ways even in a minor release. Changes are documented at the time of product release. While new protocols and APIs are still in the process of standardization, they are Evolving. This applies for example to recent Internet-Draft implementations, and also to newly developed functionality.
Deprecated	This interface is deprecated and likely to be removed in a future release. For previously stable interfaces, the change was likely announced in a previous release. Deprecated interfaces will be removed from ForgeRock products.
Removed	This interface was deprecated in a previous release and has now been removed from the product.
Technology Preview	Technology previews provide access to new features that are evolving new technology that are not yet supported. Technology preview features may be functionally incomplete and the function as implemented is subject to

Stability Label	Definition
	change without notice. DO NOT DEPLOY A TECHNOLOGY PREVIEW INTO A PRODUCTION ENVIRONMENT. Customers are encouraged to test drive the technology preview features in a non-production environment and are welcome to make comments and suggestions about the features in the associated forums. ForgeRock does not guarantee that a technology preview feature will be present in future releases, the final complete version of the feature is liable to change between preview and the final version. Once a technology preview moves into the completed version, said feature will become part of the ForgeRock platform. Technology previews are provided on an “AS-IS” basis for evaluation purposes only and ForgeRock accepts no liability or obligations for the use thereof.
Internal/Undocumented	Internal and undocumented interfaces can change without notice. If you depend on one of these interfaces, contact ForgeRock support or email info@forgerock.com to discuss your needs.

Glossary

correlation query

A correlation query specifies an expression that matches existing entries in a source repository to one or more entries on a target repository. While a correlation query may be built with a script, it is *not* a correlation script.

As noted in "Correlating Source Objects With Existing Target Objects", you can set up a query definition, such as`_queryId`, `_queryFilter`, or`_queryExpression`, possibly with the help of a`linkQualifier`.

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.

JSON

JavaScript Object Notation, a lightweight data interchange format based on a subset of JavaScript syntax. For more information, see the JSON site.

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 OpenIDM, the JWT is associated with the `JWT_SESSION` authentication module.

managed object

An object that represents the identity-related data managed by OpenIDM. Managed objects are configurable, JSON-based data structures that OpenIDM 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. While OpenIDM services are designed to run in any OSGi container, currently only Apache Felix 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

OpenIDM includes two different types of provisioning roles and authorization roles. For more information, see "Working With Managed Roles".

source object

In the context of reconciliation, a source object is a data object on the source system, that OpenIDM scans before attempting to find a corresponding object on the target system. Depending on the defined mapping, OpenIDM 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 OpenIDM for the period during which OpenIDM 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 OpenIDM scans after locating its corresponding object on the source system. Depending on the defined mapping, OpenIDM then adjusts the target object to match the corresponding source object.

Index