FORGEROCK®

Security Guide
/ ForgeRock Identity Management 7.1

Latest update: 7.1.0

Copyright © 2016-2021 ForgeRock AS.

Abstract

Guide to securing ForgeRock® Identity Management deployments.

Table of Contents

Overview

Out of the box, IDM is set up for ease of development and deployment. When you deploy IDM in production, there are specific precautions you should take to minimize security breaches. This guide describes the IDM security mechanisms and strategies you can use to reduce risk and mitigate threats to IDM security.

Quick Start

![key icon] Certificates and Keys Manage secrets, certificates and keys.	![lock icon] Passwords Store and manage passwords securely.
![cloud icon] Network Secure network connections to IDM resources.	![data icon] Data Secure IDM stored data.

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

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

Chapter 1
Secret Stores, Certificates and Keys

Encryption makes it possible to protect sensitive data. IDM depends on encryption to negotiate secure network connections, and to keep sensitive data confidential. Encryption in turn depends on keys. IDM stores keys in *secret stores*. This chapter describes the supported secret stores and the features available for managing keys.

As a general precaution in production environments, avoid using self-signed certificates and certificates associated with insecure ciphers.

IDM supports the following secret store types:

• File-based keystores

• Hardware Security Modules (HSM)

Configuring Secret Stores

Secret stores are configured in your project's `conf/secrets.json` file. The `secrets.json` file has the following configuration by default:

```
{
  "stores": [
    {
      "name": "mainKeyStore",
      "class": "org.forgerock.openidm.secrets.config.FileBasedStore",
      "config": {
        "file": "&{openidm.keystore.location|&{idm.install.dir}/security/keystore.jceks}",
        "storetype": "&{openidm.keystore.type|JCEKS}",
        "providerName": "&{openidm.keystore.provider|SunJCE}",
        "storePassword": "&{openidm.keystore.password|changeit}",
        "mappings": [
          {
            "secretId" : "idm.default",
            "types": [ "ENCRYPT", "DECRYPT" ],
            "aliases": [ "&{openidm.config.crypto.alias|openidm-sym-default}" ]
          },
          ...
        ]
      }
    },
    {
      "name": "mainTrustStore",
      "class": "org.forgerock.openidm.secrets.config.FileBasedStore",
      "config": {
```

```
        "file": "&{openidm.truststore.location|&{idm.install.dir}/security/truststore}",
        "storetype": "&{openidm.truststore.type|JKS}",
        "providerName": "&{openidm.truststore.provider|SUN}",
        "storePassword": "&{openidm.truststore.password|changeit}",
        "mappings": [
        ]
      }
    }
  ],
  "populateDefaults": true
}
```

The `mainKeyStore` and `mainTrustStore` properties configure the default secret stores. IDM requires these properties in order to start up. Do not change the property names because they are also provided to third-party products that need a single keystore and a single truststore.

mainKeyStore

The main keystore references a Java Cryptography Extension Keystore (JCEKS) located at `/path/to/openidm/security/keystore.jceks`.

mainTrustStore

The main truststore references a file-based truststore located at `/path/to/openidm/security/truststore`.

populateDefaults

When IDM first starts up, it checks the secrets configuration. If `"populateDefaults": true`, IDM writes a number of encryption keys to the keystore, required to encrypt specific data.

You can manage these keystores and truststores using the **keytool** command, included in your Java installation. For information about the **keytool** command, see https://docs.oracle.com/en/java/javase/11/tools/keytool.html.

Each configured store has a `name` and `class` and the following configuration properties:

file

For file-based secret stores, this property references the path to the store file, for example, `&{idm.install.dir}/security/keystore.jceks}`. Hardware security modules do not have a `file` property.

storetype

The type of secret store. IDM supports a number of store types, including JCEKS, JKS, PKCS #11, and PKCS #12.

providerName

Sets the name of the cryptographic service provider, for example, `SunPKCS11` or `softHSM`. If no provider is specified, the JRE default is used.

storePassword

The password to the secret store. For the default IDM keystore and trustore, the password is `changeit`. You should change this password in a production deployment, as described in "Changing the Default Keystore Password".

mappings

This object enables you to map keys and certificates in the secret stores to specific encryption and decryption functionality in IDM. A secrets mapping object has the following structure:

```
{
    "secretId" : "idm.config.encryption",
    "types": [ "ENCRYPT", "DECRYPT" ],
    "aliases": [ "&{openidm.config.crypto.alias|openidm-sym-default}" ]
}
```

- `secretId` enables you to map a secret to one or more aliases and gives an indication of the secret's purpose. For example, `idm.config.encryption` indicates the aliases that are used to encrypt and decrypt sensitive configuration properties.

- `types` indicates what the keys are used for, for example, encryption and decryption of sensitive property values.

- `aliases` are the key aliases in the secret store that are used for this purpose. You can add as many aliases as necessary. The first alias in the list determines which alias is the active one. Active secrets are used for signature generation and encryption.

  The aliases in the default keystore are described in "Working With the Default Keystore".

The default secret IDs and the aliases to which they are mapped are listed in "Mapping SecretIDs to Key Aliases".

> **Note**
>
> All these properties have a resolvable property value by default, for example `&{openidm.keystore.location}`, that enables you to use *property value substitution*. If no *configuration expression* has been set for that specific property, the value following the vertical bar is used. In the following property, the password is `changeit` unless you have set a configuration expression in one of the property resolver locations:
>
> ```
> "storePassword": "&{openidm.keystore.password|changeit}"
> ```
>
> For more information, see "Property Value Substitution" in the *Setup Guide*.

Mapping SecretIDs to Key Aliases

secretId	alias	Description
idm.default	openidm-sym-default	Encryption keystore for legacy JSON objects that do not contain a `purpose` value in their `$crypto` block

secretId	alias	Description
idm.config.encryption	openidm-sym-default	Encrypts configuration information
idm.password.encryption	openidm-sym-default	Encrypts managed user passwords
idm.jwt.session.module.encryption	openidm-localhost	Encrypts JWT session tokens
idm.jwt.session.module.signing	openidm-jwtsessionhmac-key	Signs JWT session tokens using HMAC
idm.selfservice.signing	selfservice	Signs JWT session tokens using RSA
idm.selfservice.encryption	openidm-selfservice-key	Encrypts JWT self-service tokens

Working With the Default Keystore

IDM generates a number of encryption keys in a JCEKS keystore the first time the server starts up. These keys map to the secrets defined in "Mapping SecretIDs to Key Aliases". Note that the keystore, and the keys, are generated at startup and are not prepackaged. The keys are generated *only* if they do not already exist. You cannot specify custom aliases for these default keys.

To use a different keystore type, such as PKCS #12, create the keystore and generate the keys before you start IDM. This prevents IDM from generating the keys on startup. You can also convert the existing JCEKS keystore to a PKCS #12 keystore. If you use a different keystore type, you must edit the openidm.keystore.type property (in the conf/secrets.json file) to match the new type.

Use the **keytool** command to list the default encryption keys, as follows:

```
keytool \
-list \
-keystore /path/to/openidm/security/keystore.jceks \
-storepass changeit \
-storetype JCEKS
Keystore type: JCEKS
Keystore provider: SunJCE

Your keystore contains 5 entries

openidm-sym-default, Nov 5, 2019, SecretKeyEntry,
openidm-jwtsessionhmac-key, Nov 5, 2019, SecretKeyEntry,
selfservice, Nov 5, 2019, PrivateKeyEntry,
Certificate fingerprint (SHA-256): E9:0B:BA:FB:58:73:02:FC...:7B
openidm-selfservice-key, Nov 5, 2019, SecretKeyEntry,
openidm-localhost, Nov 5, 2019, PrivateKeyEntry,
Certificate fingerprint (SHA-256): 21:50:6C:90:C7:A7:F7:32...:1B
```

> **Note**
>
> If you are using IDM in a cluster, you must share these keys among all nodes in the cluster. The easiest way to do this is to generate a keystore with the appropriate keys and share the keystore in some way, for example by using a filesystem that is shared between the nodes.

Changing the Default Keystore Password

The default keystore password is `changeit`. You should change this password in a production environment.

Change the Default Keystore Password

1. Shut down the server if it is running.

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 \
-storepass changeit
New keystore password: newPassword
Re-enter new keystore password: newPassword
```

3. Change the passwords of the default encryption keys.

   IDM uses a number of encryption keys, listed in "Mapping SecretIDs to Key Aliases", whose passwords are also `changeit` by default. The passwords of each of these keys must match the password of the keystore.

   To get the list of 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 5 entries

openidm-sym-default, May 4, 2021, SecretKeyEntry,
selfservice, May 4, 2021, PrivateKeyEntry, Certificate fingerprint (SHA-256): fingerprint
openidm-jwtsessionhmac-key, May 4, 2021, SecretKeyEntry,
openidm-localhost, May 4, 2021, PrivateKeyEntry, Certificate fingerprint (SHA-256): fingerprint
openidm-selfservice-key, May 4, 2021, SecretKeyEntry,
```

   Change the passwords of each default encryption key as follows:

```
keytool \
-keypasswd \
-alias openidm-localhost \
-keystore /path/to/openidm/security/keystore.jceks \
-storetype jceks \
-storepass 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 \
-storepass 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 \
-storepass 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 selfservice \
-keystore /path/to/openidm/security/keystore.jceks \
-storetype jceks \
-storepass newPassword
Enter key password for <selfservice> changeit
New key password for <selfservice>: newPassword
Re-enter new key password for <selfservice>: newPassword
```

```
keytool \
-keypasswd \
-alias openidm-jwtsessionhmac-key \
-keystore /path/to/openidm/security/keystore.jceks \
-storetype jceks \
-storepass 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. Configure a new expression resolver file in the *Setup Guide* to store just the keystore password.

   a. Create a new directory in `/path/to/openidm/resolver/` that will contain only the properties file for keystore passwords. For example:

   ```
   mkdir /path/to/openidm/resolver/keystore
   ```

> **Important**
>
> Substituted properties are not encrypted by default. You *must* therefore secure access to this directory, using the appropriate permissions.

b. Set the `IDM_ENVCONFIG_DIRS` environment variable to include the new directory:

```
export IDM_ENVCONFIG_DIRS=/path/to/openidm/resolver/,/path/to/openidm/resolver/keystore
```

c. Create a `.json` or `.properties` file in that secure directory, that contains the new keystore password as a resolvable IDM property. For example, add one of the following files to that directory:

keystorepwd.properties

```
openidm.keystore.password=newPassword
```

keystorepwd.json

```
{
    "openidm" : {
        "keystore" : {
            "password" : "newPassword"
        }
    }
}
```

5. Restart IDM.

> **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.

Using CA-Signed Certificates

You can use existing CA-signed certificates to secure connections and data by importing the certificates into the keystore, and referencing them your `boot.properties` file. Use the **keytool** command to import an existing certificate into the keystore.

The following process imports a CA-signed certificate into the keystore, with the alias *example-com*. Replace this alias with the alias of your certificate:

1. Stop the server if it is running.

2. Back up your existing `openidm/security/keystore` and `openidm/security/truststore` files.

3. Use the **keytool** command to import your existing certificate into the keystore.

Substitute the following in this command:

- `example-cert.p12` with the name of your certificate file.

- `srcstorepass` with the password that you set to open your certificate.

- `example-com` with the existing certificate alias.

- `destination keystore password` with the password you set for the keystore.

  If you have not changed the default keystore password, it is `changeit`. In a production environment, you *should* change the default keystore password. For more information, see "Changing the Default Keystore Password".

```
keytool \
-importkeystore \
-srckeystore example-cert.p12 \
-srcstoretype PKCS12 \
-srcstorepass changeit \
-srcalias example-com \
-destkeystore keystore.jceks \
-deststoretype JCEKS \
-destalias openidm-localhost
Importing keystore example-cert.p12 to keystore.jceks...
Enter destination keystore password: changeit
```

The keytool command creates a trusted certificate entry with the specified alias and associates it with the imported certificate. The certificate is imported into the keystore with the alias `openidm-localhost`. If you want to use a different alias, you must modify your `resolver/boot.properties` file to reference that alias, as shown in the following step.

> **Note**
>
> The certificate entry password must be the same as the IDM keystore password. If the source certificate entry password is different from the target keystore password, use the `-destkeypass` option with the same value as the `-deststorepass` option to make the certificate password match the target keystore password. If you do not make these passwords the same, no error is generated when you import the certificate (or when you read the certificate entry in the destination keystore), but IDM will fail to start with the following exception:
>
> ```
> java.security.UnrecoverableKeyException: Given final block not properly padded.
> ```

4. If you specified an alias other than `openidm-localhost` for the new certificate, change the value of `openidm.https.keystore.cert.alias` in your `resolver/boot.properties` file to that alias. For example, if your new certificate alias is `example-com`, change the `boot.properties` file as follows:

   ```
   openidm.https.keystore.cert.alias=example-com
   ```

5. Restart the server for the new certificate to be taken into account.

Deleting Certificates

If you are using CA-signed certificates for encryption, it is best practice to delete the unused default certificates from the keystore and the truststore. You can delete certificates from a keystore using the **keytool** command.

The following example deletes the `openidm-localhost` certificate from the keystore:

```
keytool \
-delete \
-alias openidm-localhost \
-keystore /path/to/openidm/security/keystore.jceks \
-storetype JCEKS \
-storepass changeit
```

The following example deletes the `openidm-localhost` certificate from the truststore:

```
keytool \
-delete \
-alias openidm-localhost \
-keystore /path/to/openidm/security/truststore \
-storepass changeit
```

You can use similar commands to delete custom certificates from the keystore and truststore, specifying the certificate alias in the request.

Repeat these steps to delete all the default certificate aliases that you are not using in your deployment.

Removing Unused CA Certificates

The Java and IDM truststore files include a number of root CA certificates. Although the probability of a compromised root CA certificate is low, it is best practice to delete root CA certificates that are not used in your deployment.

To review the list of root CA certificates in the IDM truststore, run the following command:

```
keytool \
-list \
-keystore /path/to/openidm/security/truststore \
-storepass changeit
```

On UNIX/Linux systems, you can find additional lists of root CA certificates in files named `cacerts`. These include root CA certificates associated with your Java environment, such as Oracle JDK or OpenJDK. You should be able to find that file in `${JAVA_HOME}/jre/lib/security/cacerts`.

Before changing Java environment keystore files, make sure that the Java-related `cacerts` files are up to date and verify that you have a supported Java version installed:

Supported Java Versions

Vendor	Versions
OpenJDK, including OpenJDK-based distributions: • AdoptOpenJDK/Eclipse Adoptium • Amazon Corretto • Azul Zulu • Red Hat OpenJDK ForgeRock tests most extensively with AdoptOpenJDK/Eclipse Adoptium. ForgeRock recommends using the HotSpot JVM.	11
Oracle Java	11

You can remove root CA certificates with the **keytool** command. For example, the following command removes the hypothetical `examplecomca2` certificate from the truststore:

```
keytool \
-delete \
-keystore /path/to/openidm/security/truststore \
-storepass changeit \
-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 Working With Certificates in the Microsoft documentation.

Changing and Rotating Encryption Keys

Most regulatory requirements mandate that the keys used to decrypt sensitive data be rotated out and replaced with new keys on a regular basis. The main purpose of rotating encryption keys is to reduce the amount of data encrypted with that key, so that the potential impact of a security breach with a specific key is reduced. You can update encryption keys in several ways, including the following:

• "Rotating Encryption Keys Manually"

• "Using Scheduled Tasks to Rotate Keys"

• "Changing the Active Alias for Managed Object Encryption"

Rotating Encryption Keys Manually

IDM evaluates keys in `secrets.json` sequentially. For example, assume that you have added a new key named `my-new-key` to the keystore, as described in "Using CA-Signed Certificates".

To use this new key to encrypt passwords, you would include `my-new-key` as the *first alias* in the `idm.password.encryption` secret, as follows:

```
{
    "secretId" : "idm.password.encryption",
    "types": [ "ENCRYPT", "DECRYPT" ],
    "aliases": [ "my-new-key", "&{openidm.config.crypto.alias|openidm-sym-default}" ]
}
```

The properties that use this key (in this case, passwords) are re-encrypted with the new key the next time the managed object is *updated*. You do not need to restart the server.

> **Important**
>
> If you rotate an encryption key, the *active* encryption key might not be the correct key to use for decryption of properties that have already been encrypted with a previous key.
>
> You must therefore keep all applicable keys in `secrets.json` until every object that is encrypted with old keys have been updated with the latest key.

You can force 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 "Accept-API-Version: resource=1.0" \
--cacert ca-cert.pem \
--header "Content-Type: application/json" \
--request POST \
"https://localhost:8443/openidm/managed/user/?_action=triggerSyncCheck"
{
    "status": "OK",
    "countTriggered": 10
}
```

In a large managed object set, the `triggerSyncCheck` action can take a 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" \
--header "Accept-API-Version: resource=1.0" \
--cacert ca-cert.pem \
"https://localhost:8443/openidm/managed/user?_queryFilter=true&_fields=_id"
{
  "result": [
    {
      "_id": "9dce06d4-2fc1-4830-a92b-bd35c2f6bcbb",
      "_rev": "000000004988917b"
    },
    {
      "_id": "55ef0a75-f261-47e9-a72b-f5c61c32d339",
      "_rev": "00000000dd89d671"
    },
    {
      "_id": "998a6181-d694-466a-a373-759a05840555",
      "_rev": "000000006fea54ad"
    },
    ...
  ]
}
```

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--cacert ca-cert.pem \
--header "Content-Type: application/json" \
--request POST \
"https://localhost:8443/openidm/managed/user/9dce06d4-2fc1-4830-a92b-bd35c2f6bcbb?
_action=triggerSyncCheck"
```

In large data sets, the most efficient way to achieve key rotation is to use the scheduler service to launch these commands. The following section shows how to use 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 & lt; 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" : {
        "purpose" : "idm.password.encryption"
    },
    "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" \
--header "Accept-API-Version: resource=1.0" \
--cacert ca-cert.pem \
--request GET \
"https://localhost:8443/openidm/managed/user/ccd92204-aee6-4159-879a-46eeb4362807"
{
  "_id" : "ccd92204-aee6-4159-879a-46eeb4362807",
  "_rev" : "0000000009441230",
  "preferences" : {
    "updates" : false,
    "marketing" : false
  },
  "mail" : "bjensen@example.com",
  "sn" : "Jensen",
  "givenName" : "Babs",
  "userName" : "bjensen",
  "password" : {
    "$crypto" : {
      "type" : "x-simple-encryption",
      "value" : {
        "cipher" : "AES/CBC/PKCS5Padding",
        "stableId" : "openidm-sym-default",
        "salt" : "CVrKDuzfzunXfTDbCwU1Rw==",
        "data" : "1I5tWT5aRH/12hf5DgofXA==",
        "keySize" : 16,
        "purpose" : "idm.password.encryption",
        "iv" : "LGE+jnC3ZtyvrE5pfuSvtA==",
        "mac" : "BEXQ1mftxA63dXhJO6dDZQ=="
      }
    }
  },
  "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 IDM keystore:

```
keytool \
-genseckey \
-alias my-new-key \
-keyalg AES \
-keysize 128 \
-keystore /path/to/openidm/security/keystore.jceks \
-storetype JCEKS
```

d. Shut down the server for keystore to be reloaded.

e. Change your project's `conf/managed.json` file to change the encryption purpose for managed user passwords:

```
"password" : {
    ...
    "encryption" : {
        "purpose" : "idm.password.encryption2"
    },
    "scope" : "public",
    ...
}
```

f. Add the corresponding `purpose` to the `secrets.json` file in the `mainKeyStore` code block:

```
"idm.password.encryption2": {
  "types": [ "ENCRYPT", "DECRYPT" ],
  "aliases": [
    {
      "alias": "my-new-key"
    }
  ]
}
```

g. Restart the server and wait one minute for the first scheduled task to fire.

h. 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" \
--header "Accept-API-Version: resource=1.0" \
--cacert ca-cert.pem \
--request GET \
"https://localhost:8443/openidm/managed/user/ccd92204-aee6-4159-879a-46eeb4362807"
{
  "_id" : "ccd92204-aee6-4159-879a-46eeb4362807",
  "_rev" : "0000000009441230",
  "preferences" : {
    "updates" : false,
    "marketing" : false
  },
  "mail" : "bjensen@example.com",
  "sn" : "Jensen",
  "givenName" : "Babs",
  "userName" : "bjensen",
```

```
    "password" : {
      "$crypto" : {
        "type" : "x-simple-encryption",
        "value" : {
          "cipher" : "AES/CBC/PKCS5Padding",
          "stableId" : "my-new-key",
          "salt" : "CVrKDuzfzunXfTDbCwU1Rw==",
          "data" : "1I5tWT5aRH/12hf5DgofXA==",
          "keySize" : 16,
          "purpose" : "idm.password.encryption2",
          "iv" : "LGE+jnC3ZtyvrE5pfuSvtA==",
          "mac" : "BEXQ1mftxA63dXhJO6dDZQ=="
        }
      }
    },
    "accountStatus" : "active",
    "effectiveRoles" : [ ],
    "effectiveAssignments" : [ ]
}
```

Notice that the user password is now encrypted with `my-new-key`.

Changing the Active Alias for Managed Object Encryption

This example describes how you can configure and then change the managed object encryption key with a scheduled task. You'll create a new key, set up a managed user, add the key to `secrets.json`, restart IDM, run a `triggerSyncCheck`, and review the result.

1. Create a new key for the IDM keystore in the `security/keystore.jceks` file:

```
keytool \
-genseckey \
-alias my-new-key \
-keyalg AES \
-keysize 128 \
-keystore /path/to/openidm/security/keystore.jceks \
-storetype JCEKS
```

2. Solely for the purpose of this example, in `managed.json`, set `"scope" : "public"` to expose the applied password encryption key.

3. Create a managed user:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--cacert ca-cert.pem \
--header "Content-Type: application/json" \
--request PUT \
--data '{
  "userName": "rsutter",
  "sn": "Sutter",
  "givenName": "Rick",
  "mail": "rick@example.com",
  "telephoneNumber": "6669876987",
  "description": "Another user",
  "country": "USA",
  "password": "Passw0rd"
}' \
"https://localhost:8443/openidm/managed/user/ricksutter"
```

4. Add the newly created `my-new-key` alias to your `conf/secrets.json` file, in the `idm.password.encryption` code block:

```
"idm.password.encryption": {
  "types": [ "ENCRYPT", "DECRYPT" ],
  "aliases": [ "my-new-key", "&{openidm.config.crypto.alias|openidm-sym-default}" ]
}
```

5. To apply the new key to your configuration, shut down and restart IDM.

6. Force IDM to update the key for your users with the `triggerSyncCheck` action:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--cacert ca-cert.pem \
--header "Content-Type: application/json" \
--request POST \
"https://localhost:8443/openidm/managed/user/?_action=triggerSyncCheck"
```

7. Review the result for the newly created user, `ricksutter`:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Accept-API-Version: resource=1.0" \
--cacert ca-cert.pem \
--request GET  \
"https://localhost:8443/openidm/managed/user/ricksutter"
```

8. In the output, you should see the new `my-new-key` encryption key applied to that user's password:

```
...
 "password": {
   "$crypto": {
     "type": "x-simple-encryption",
     "value": {
       "cipher": "AES/CBC/PKCS5Padding",
       "stableId": "my-new-key",
       "salt": "bGyKG3PKmwHONOfxerr1Qg==",
       "data": "6vXZiJ3ZNN/UUnsrT7dTQw==",
       "keySize": 16,
       "purpose": "idm.password.encryption",
       "iv": "doAdtxfWfFbrPIIfubGi5g==",
       "mac": "OML6xd9qvDtD5AvMc1Tc3A=="
     }
   }
 },
...
```

Configuring IDM For a Hardware Security Module (HSM) Device

This section demonstrates how to use a PKCS #11 device, such as a hardware security module (HSM), to store the keys used to secure communications. IDM supports retrieval of secrets from HSMs either locally or over the network.

> **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.

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` 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.

Populating the Default Encryption Keys

When IDM 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. The secret keys must use an HMAC algorithm. The following steps set up the required encryption keys.

> **Note**
>
> This procedure assumes that your HSM configuration file is located at `/path/to/hsm/hsm.conf`:

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 AES \
   -keysize 128 \
   -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 AES \
   -keysize 128 \
   -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, see "JWT_SESSION" in the *Authentication and Authorization Guide*. 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 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. The `selfservice` certificate secures requests from the End User UI. Generate that certificate with a command similar to the following:

```
keytool \
-genkey \
-alias selfservice \
-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 Service 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,O=OpenIDM Self Service Certificate,OU=None,L=None,ST=None,C=None?
  [no]:  yes
```

6. If you are *not* using the HSM provider for the truststore, you must add the certificates generated in the previous two steps to the default IDM truststore.

   If you *are* using the HSM provider for the truststore, you can skip this step.

   To add the `openidm-localhost` certificate to the IDM 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*.

Configuring IDM to Support an HSM Provider

To enable IDM to use an HSM provider, make the following configuration changes:

In your secret store configuration (`conf/secrets.json`)

Change the `mainKeyStore` and `mainTrustStore` to reference the HSM. For example:

```
{
  "stores": [
    {
      "name": "mainKeyStore",
      "class": "org.forgerock.openidm.secrets.config.HsmBasedStore",
      "config": {
        "storetype": "&{openidm.keystore.type|PKCS11}",
        "providerName": "&{openidm.keystore.provider|SunPKCS11-softHSM}",
        "storePassword": "&{openidm.keystore.password|changeit}",
        "mappings": [
          {
            "secretId" : "idm.default",
            "types": [ "ENCRYPT", "DECRYPT" ],
            "aliases": [ "&{openidm.config.crypto.alias|openidm-sym-default}" ]
          },
          {
            "secretId" : "idm.config.encryption",
            "types": [ "ENCRYPT", "DECRYPT" ],
            "aliases": [ "&{openidm.config.crypto.alias|openidm-sym-default}" ]
          },
          {
            "secretId" : "idm.password.encryption",
            "types": [ "ENCRYPT", "DECRYPT" ],
            "aliases": [ "&{openidm.config.crypto.alias|openidm-sym-default}" ]
          },
          {
            "secretId" : "idm.jwt.session.module.encryption",
            "types": [ "ENCRYPT", "DECRYPT" ],
            "aliases": [ "&{openidm.https.keystore.cert.alias|openidm-localhost}" ]
          },
```

```
          {
            "secretId" : "idm.jwt.session.module.signing",
            "types": [ "SIGN", "VERIFY" ],
            "aliases": [ "&{openidm.config.crypto.jwtsession.hmackey.alias|openidm-jwtsessionhmac-
key}" ]
          },
          {
            "secretId" : "idm.selfservice.signing",
            "types": [ "SIGN", "VERIFY" ],
            "aliases": [ "selfservice" ]
          },
          {
            "secretId" : "idm.selfservice.encryption",
            "types": [ "ENCRYPT", "DECRYPT" ],
            "aliases": [ "&{openidm.config.crypto.selfservice.sharedkey.alias|openidm-selfservice-
key}" ]
          }
        ]
      }
    },
    {
      "name": "mainTrustStore",
      "class": "org.forgerock.openidm.secrets.config.HsmBasedStore",
      "config": {
        "storetype": "&{openidm.keystore.type|PKCS11}",
        "providerName": "&{openidm.keystore.provider|SunPKCS11-softHSM}",
        "storePassword": "&{openidm.keystore.password|changeit}",
        "mappings": [
        ]
      }
    }
  ],
  "populateDefaults": false
}
```

> **Note**
>
> The `"populateDefaults": false` turns off the default key generation. This setting is *required* for an HSM
> key provider.

In the IDM Java security file (`conf/java.security`)

Specify the location of your PKCS #11 configuration file. For example:

```
security.provider.14=SunPKCS11 /path/to/pkc11/config/pkcs11.conf
```

Templates for the `pkcs11.conf` file are included in your PKCS package.

You should now be able to start IDM with the keys in the HSM provider.

Chapter 2
Secure Passwords

IDM provides password management features that help you enforce password policies, limit the number of passwords users must remember, and allow users to reset and change their passwords.

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.

IDM enforces password policy rules as part of the general policy service. 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 change these validation requirements, or include additional requirements, by configuring the policy for passwords.

Passwords are validated in several situations:

Password change and password reset

Password *change* refers to users changing their own passwords. Password *reset* refers to an administrator setting a user or account password on behalf of a user.

By default, IDM validates password values as they are provisioned.

Password recovery

Password recovery involves recovering a password or setting a new password when the password has been forgotten.

Password history

You can add validation to prevent reuse of previous password values. For more information, see "Creating a Password History Policy".

Password expiration

You can use workflows to ensure that users are able to change expiring passwords or to reset expired passwords.

Creating a Password History Policy

The sample described in "*Store Multiple Passwords For Managed Users*" in the *Samples Guide* shows how to set up a password history policy in a scenario where users have multiple different passwords across resources. You can use the scripts provided in that sample to set up a simple password history policy that prevents managed users from setting the same password that they used previously.

To create a password history policy based on the scripts in the multiple passwords sample, make the following changes to your project:

1. Copy the `pwpolicy.js` script from the multiple passwords sample to your project's `script` directory:

   ```
   cp /path/to/openidm/samples/multiple-passwords/script/pwpolicy.js /path/to/openidm/my-project-dir/
   script/
   ```

   The `pwpolicy.js` script contains an `is-new` policy definition that compares a new field value with the list of historical values for that field.

   The `is-new` policy takes a `historyLength` parameter that specifies the number of historical values on which the policy should be enforced. This number must not exceed the `historySize` that you set in `conf/managed.json` to be passed to the `onCreate` and `onUpdate` scripts.

2. Copy the `onCreate-user-custom.js` and `onUpdate-user-custom.js` scripts to your project's `script` directory:

   ```
   cp samples/multiple-passwords/script/onCreate-user-custom.js /my-project-dir/script/
   cp samples/multiple-passwords/script/onUpdate-user-custom.js /my-project-dir/script/
   ```

   These scripts validate the password history policy when a managed user is created or updated.

3. Update your policy configuration (`conf/policy.json`) to reference the new policy definition by adding the policy script to the `additionalFiles` array:

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

4. Update your project's `conf/managed.json` file as follows:

    *   Add a `fieldHistory` property to the managed user object:

        ```
        "fieldHistory" : {
            "title" : "Field History",
            "type" : "object",
            "viewable" : false,
            "searchable" : false,
            "userEditable" : false,
            "scope" : "private"
        }
        ```

        The value of this field is a map of field names to a list of historical values for that field. These lists of values are used by the `is-new` policy to determine if a new value has already been used.

    *   Update the managed user object to call the scripts when a user is created, updated:

        ```
        "name" : "user",
        "onCreate" : {
            "type" : "text/javascript",
            "file" : "script/onCreate-user-custom.js",
            "historyFields" : [
                "password"
            ],
            "historySize" : 4
        },
        "onUpdate" : {
            "type" : "text/javascript",
            "file" : "script/onUpdate-user-custom.js",
            "historyFields" : [
                "password"
            ],
            "historySize" : 4
        },
        ...
        ```

        > **Important**
        >
        > If you have any other script logic that is executed on these events, you must update the scripts to include that logic, or add the password history logic to your current scripts.

    *   Add the `is-new` policy to the list of policies enforced on the `password` property of a managed user. Specify the number of historical values that the policy should check in `historyLength` property:

```
"password" : {
    ...
    "policies" : [
        {
            "policyId" : "at-least-X-capitals",
            "params" : {
                "numCaps" : 1
            }
        },
        ...
        {
            "policyId" : "is-new",
            "params" : {
                "historyLength" : 4
            }
        },
        ...
    ]
}
```

You should now be able to test the password history policy by creating a new managed user, and having that user update their password. If the user specifies the same password used within the previous four passwords, the update request is denied with a policy error.

Storing Separate Passwords Per Linked Resource

You can store multiple passwords in a single managed user entry to enable synchronization of different passwords on different external resources.

To store multiple passwords, 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.

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 "*Store Multiple Passwords For Managed Users*" in the *Samples Guide*. That sample can also help you set up password history policies.

Generating Random Passwords

In certain situations, you might want to generate a random password when users are created.

You can 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/onCreateUser.js` script. The following section of that file

(commented out by default) means that users created through 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);

}
```

Note that changes made to scripts take effect after the time set in the `recompile.minimumInterval`, described in "*Script Configuration*" in the *Scripting Guide*.

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". Note that synchronizing hashed passwords is not supported.

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.

Modifying the `password` Property

To use a property other than the default `password` property to store passwords, you must change the following files:

policy.json

> If you want to enforce password validation rules on a different property, change the `password` property in this file.

managed.json

> Modify the `password` object in this file, which also includes password complexity policies.

sync.json

> If you change the `password` property, make sure that you limit the change to the appropriate system, designated as `source` or `target`.

selfservice-reset.json

> If you are setting up self-service password reset, as described in "*Password Reset*" in the *Self-Service Reference*, change the value of `identityPasswordField` from `password` to the desired new property.

Every UI file that includes `password` as a property name

Whenever there's a way for a user to enter a password, the associated `HTML` page will include a password entry. For example, the `LoginTemplate.html` file includes the `password` property. A full list of default files with the `password` property include:

- `_passwordFields.html`

- `_resetPassword.html`

- `ConfirmPasswordDialogTemplate.html`

- `EditPasswordPageView.html`

- `LoginTemplate.html`

- `MandatoryPasswordChangeDialogTemplate.html`

- `resetStage-initial.html`

- `UserPasswordTab.html`

This list does not include any custom UI files that you might have created.

Rate Limiting Emails

No rate limiting is applied to password reset emails, or any emails sent by the IDM server. This means that an attacker can potentially spam a known user account with an infinite number of emails, filling that user's inbox. In the case of password reset, the spam attack can obscure an actual password reset attempt.

In a production environment, you must configure email rate limiting through the network infrastructure in which IDM runs. Configure the network infrastructure to detect and prevent frequent repeated requests to publicly accessible web pages, such as the password reset page. You can also handle rate limiting within your email server.

Chapter 3

Secure Network Connections

This chapter explains how to secure incoming connections and ports. As a general precaution in production environments, avoid communication over insecure HTTP.

- "Use TLS/SSL"

- "Restrict REST Access to the HTTPS Port"

- "Protect Sensitive REST Interface URLs"

- "Enable HTTP Strict-Transport-Security"

- "Restrict the HTTP Payload Size"

- "Deploy Securely Behind a Load Balancer"

- "Connect to IDM Through a Proxy Server"

Use TLS/SSL

Use TLS/SSL to access IDM, 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 truststore and keystore of each application, provides a level of confidence for trusting application access.

Restrict REST Access to the HTTPS Port

In a production environment, you should restrict REST access to a secure port:

1. Edit your project's `conf/jetty.xml` file:

   Comment out or delete the `<Call name="addConnector">` code block that includes the `openidm.port.http` property.

> **Note**
>
> Do not delete the `<Call name="addConnector">` code blocks that contain the `openidm.port.https` and `openidm.port.mutualauth` properties.

2. Edit `resolver/boot.properties`:

   - Set the `openidm.port.https` port number.

   - Set the `openidm.port.mutualauth` port number.

   - Add the property `openidm.https.enabled=true`.

Use a certificate to secure REST access over HTTPS. You can use self-signed certificates in a test environment. In production, all certificates should be signed by a certificate authority. The examples in this guide assume a CA-signed certificate named `ca-cert.pem`.

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 "Configure Access Control in `access.json`" in the *Authentication and Authorization Guide*.

Enable HTTP Strict-Transport-Security

HTTP Strict-Transport-Security (HSTS) is a web security policy that forces browsers to make secure HTTPS connections to specified web applications. HSTS can protect websites against passive eavesdropper and active man-in-the-middle attacks.

IDM provides an HSTS configuration but it is disabled by default. To enable HSTS, locate the following excerpt in your `conf/jetty.xml` file:

```
<New id="tlsHttpConfig" class="org.eclipse.jetty.server.HttpConfiguration">
    ...
    <Call name="addCustomizer">
        <Arg>
            <New class="org.eclipse.jetty.server.SecureRequestCustomizer">
                <!-- Enable SNI Host Check when true -->
                <Arg name="sniHostCheck" type="boolean">true</Arg>
                <!-- Enable Strict-Transport-Security header and define max-age when >= 0 seconds -->
                <Arg name="stsMaxAgeSeconds" type="long">-1</Arg>
                <!-- If enabled, add includeSubDomains to Strict-Transport-Security header when true -->
                <Arg name="stsIncludeSubdomains" type="boolean">false</Arg>
            </New>
        </Arg>
    </Call>
...
```

Set the following arguments:

stsMaxAgeSeconds

> This parameter sets the length of time, in seconds, that the browser should remember that a site can only be accessed using HTTPS.
>
> For example, the following setting applies the HSTS policy and remains in effect for an hour:
>
> ```
> <Arg name="stsMaxAgeSeconds" type="long">3600</Arg>
> ```

stsMaxAgeSeconds

> If this parameter is `true>`, the HSTS policy is applied to the domain of the issuing host as well as its subdomains:
>
> ```
> <Arg name="stsIncludeSubdomains" type="boolean">true</Arg>
> ```

For more information about HSTS, read this article.

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 "Register Additional Servlet Filters" in the *Installation Guide*.

Deploy Securely Behind a Load Balancer

IDM prevents URL-hijacking, with the following code block in the `conf/jetty.xml` file:

```
<Call name="addCustomizer">
  <Arg>
    <New class="org.eclipse.jetty.server.SecureRequestCustomizer">
      <!-- Enable SNI Host Check when true -->
      <Arg name="sniHostCheck" type="boolean">true</Arg>
      <!-- Enable Strict-Transport-Security header and define max-age when >= 0 seconds -->
      <Arg name="stsMaxAgeSeconds" type="long">-1</Arg>
      <!-- If enabled, add includeSubDomains to Strict-Transport-Security header when true -->
      <Arg name="stsIncludeSubdomains" type="boolean">false</Arg>
    </New>
  </Arg>
</Call>
```

If you are deploying IDM behind a system such as a load balancer, firewall, or a reverse proxy, you must uncomment the next section in `jetty.xml`, so that Jetty honors `X-Forwarded-Host` headers:

```
  <Call name="addCustomizer">
    <Arg>
        <New class="org.eclipse.jetty.server.ForwardedRequestCustomizer">
            <Set name="forcedHost">
                <Call class="org.forgerock.openidm.jetty.Param" name="getProperty">
                    <Arg>openidm.host</Arg>
                </Call>:<Call class="org.forgerock.openidm.jetty.Param" name="getProperty">
                    <Arg>openidm.port.https</Arg>
                </Call>
            </Set>
        </New>
    </Arg>
</Call>
```

Connect to IDM Through a Proxy Server

To configure IDM to communicate through a proxy server:

1. Add the following JVM parameters to the value of `OPENIDM_OPTS` in your startup script (`startup.sh` or `startup.bat`):

   -Dhttps.proxyHost

   Hostname or IP address of the proxy server; for example, `proxy.example.com` or `192.168.0.1`.

   -Dhttps.proxyPort

   Port number used by IDM; for example, `8443` or `9443`.

   For example:

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

2. Enable the `ForwardedRequestCustomizer` class so that Jetty honors `X-Forwarded-` headers.

   To enable the class, uncomment the following excerpt in your `conf/jetty.xml` file:

   ```
   <Call name="addCustomizer">
       <Arg>
           <New class="org.eclipse.jetty.server.ForwardedRequestCustomizer">
               <Set name="forcedHost">
                   <Call class="org.forgerock.openidm.jetty.Param" name="getProperty">
                       <Arg>openidm.host</Arg>
                   </Call>:
                   <Call class="org.forgerock.openidm.jetty.Param" name="getProperty">
                       <Arg>openidm.port.https</Arg>
                   </Call>
               </Set>
           </New>
       </Arg>
   </Call>
   ```

   For more information on this class, see the Jetty documentation.

Chapter 4
Protect IDM Data

Beyond relying on end-to-end availability of TLS/SSL to protect data, IDM 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.

IDM also supports encryption of data stored in the repository, using the symmetric keys specified in `conf/secrets.json`. This protects against some attacks on the data store. Explicit table mapping is supported for encrypted string values.

IDM automatically encrypts sensitive data (such as passwords) in configuration files, and replaces clear text values when the system first reads the configuration file. Take care with configuration files that contain clear text values that IDM has not yet read and encrypted.

Encoding Attribute Values

There are two ways to encode attribute values for managed objects—reversible encryption and 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 "*Secure Passwords*".

You configure attribute value encoding, per schema property, in the managed object configuration *(You can edit the managed object configuration over REST at the config/managed endpoint, or directly in the conf/managed.json file.)*. The following sections show how to use reversible encryption and salted hash algorithms to encode attribute values.

Encoding Attribute Values With Reversible Encryption

The following managed object configuration *(You can edit the managed object configuration over REST at the config/managed endpoint, or directly in the conf/managed.json file.)* encrypts and decrypts the `password` attribute using the default symmetric key:

```
{
    "objects" : [
        {
            "name" : "user",
            ...
            "schema" : {
                ...
                "properties" : {
                    ...
                    "password" : {
                        "title" : "Password",
                        ...
                        "encryption" : {
                            "purpose" : "idm.password.encryption"
                        },
                        "scope" : "private",
                    }
                ...
            }
        ]
}
```

The settings for reversible encryption depend on the encryption capabilities of the underlying JVM.
See the explanations in javax.crypto.Cipher. You can accept the default settings, or specify the `cipher`
and the `keySize`, for example:

```
...
    "encryption" : {
        "purpose": "idm.password.encryption",
        "cipher": "AES/GCM/NoPadding",
        "keySize": 128
},
```

The syntax for the `cipher` is *algorithm/mode/padding*, for example, `"cipher" : "AES/CBC/PKCS5Padding"`:

• The cipher algorithm defines how the plaintext is encrypted and decrypted.

  The default algorithm is the Advanced Encryption Standard (AES).

• The cipher mode defines how a block cipher algorithm transforms data larger than a single block.

  The default cipher mode is cipher block chaining (CBC).

• The cipher padding defines how to pad the plaintext to reach the appropriate size for the algorithm.

  The default cipher padding is PKCS#5 padding.

• The cipher key size determines the encryption strength, where longer key lengths strengthen
  encryption at the cost of lower performance.

  The default `keySize` is `16`.

> **Note**
>
> If you change the default cipher, you must specify the algorithm, mode, and padding. If the algorithm does not require a mode, use `NONE`. If the algorithm does not require padding, use `NoPadding`.

To encrypt attribute values from the command-line, see "**encrypt**" in the *Setup Guide*.

+ *To Configure Encryption Through the UI*

1. Select Configure > Managed Objects, and select 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.

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 define the hashing configuration. The configuration depends on the algorithm that you choose.

If you do not specify an algorithm, `SHA-256` is used by default. MD5 and SHA-1 are supported for legacy reasons but you should use a more secure algorithm in production environments.

The following list shows the supported hash algorithms and their configurations:

SHA-256

```
"secureHash" : {
    "algorithm" : "SHA-256",
    "saltLength" : 16
}
```

SHA-384

```
"secureHash" : {
    "algorithm" : "SHA-384",
    "saltLength" : 16
}
```

SHA-512

```
"secureHash" : {
    "algorithm" : "SHA-512",
    "saltLength" : 16
}
```

Bcrypt

```
"secureHash" : {
    "algorithm" : "BCRYPT",
    "cost" : 16
}
```

Scrypt

```
"secureHash" : {
    "algorithm" : "SCRYPT",
    "hashLength" : 16,
    "saltLength" : 16,
    "n" : 32768,
    "r" : 8,
    "p" : 1
}
```

Password-Based Key Derivation Function 2 (PBKDF2)

```
"secureHash" : {
    "algorithm" : "PBKDF2",
    "hashLength" : 16,
    "saltLength" : 16,
    "iterations" : 10,
    "hmac" : "SHA-256"
}
```

> **Warning**
>
> Some one-way hash functions are designed to be computationally *expensive*. Functions such as PBKDF2, Bcrypt, and Scrypt are designed to be relatively slow even on modern hardware. This makes them generally less susceptible to brute force attacks. *However*, computationally expensive functions can dramatically increase response times. If you use these functions, be aware of the performance impact and perform extensive testing before deploying your service in production. Do not use functions like PBKDF2 and Bcrypt for any accounts that are used for frequent, short-lived connections.
>
> Hashing is a one-way operation, such that the original value cannot be recovered. 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.

The following excerpt of a managed object configuration shows that values of the `password` attribute are hashed using the `SHA-256` algorithm:

```
{
    "objects" : [
        {
            "name" : "user",
            ...
            "schema" : {
                ...
                "properties" : {
                    ...
                    "password" : {
                        "title" : "Password",
                        ...
                        "secureHash" : {
                            "algorithm" : "SHA-256"
                        },
                        "scope" : "private",
                    }
                ...
            }
        ]
}
```

To hash attribute values from the command-line, see **"secureHash"** in the *Setup Guide*.

+ *To Configure Hashing Through the UI*

You can configure hashing of properties through the Admin UI, but the functionality is limited to setting the hash algorithm. Not all algorithms are supported in the UI, and none of the enhanced configuration options are supported. To configure attribute hashing in the UI:

1. Select Configure > Managed Objects, and select 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, select Privacy & Encryption, then select the Hashed checkbox.

3. Select the algorithm that should be used to hash the property value.

Structure of an Encrypted Object

Encrypted objects and properties, such as passwords, include a `$crypto` object, that has the following structure:

```
"password": {
  "$crypto": {
    "type": "x-simple-encryption",
    "value": {
      "cipher": "AES/CBC/PKCS5Padding",
      "stableId": "openidm-sym-default",
      "salt": "Gwi+AGrn+VBOTmyq+TTuuw==",
      "data": "+9i7XAXpWZBXYTVEOBkM+w==",
      "keySize": 16,
      "purpose": "idm.password.encryption",
      "iv": "4xtI88eFu5tgfm8ooq+yqQ==",
      "mac": "NlzsYo71M/b/G6iLOhNohA=="
    }
  }
}
```

Most of the properties in the encrypted object `value` are self-explanatory and indicate how the property was encrypted. Specific IDM properties include the following:

- The `stableId` indicates the key alias that was used to encrypt the property value.

- The `purpose` refers to the secret ID used to encrypt the property value. For more information about secret IDs, see "Configuring Secret Stores".

Encrypting and Decrypting Properties Over REST

The `openidm.encrypt` and `openidm.decrypt` functions of the Resource API enable you to encrypt and decrypt property values. To use these functions over the REST interface, run the `?_action=eval` action on the `script` endpoint.

The following example uses the `openidm.encrypt` function to encrypt a password value:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Content-Type: application/json" \
--cacert ca-cert.pem \
--request POST \
--data '{
  "type": "text/javascript",
  "globals": {
    "val": {
      "myKey": "myPassword"
    }
  },
  "source":"openidm.encrypt(val,null,\"idm.password.encryption\");"
}' \
"https://localhost:8443/openidm/script?_action=eval"
{
  "$crypto": {
    "type": "x-simple-encryption",
    "value": {
      "cipher": "AES/CBC/PKCS5Padding",
      "stableId": "openidm-sym-default",
      "salt": "qAS/eG7zdnFyK5H8lXvqTA==",
      "data": "zewf6hR1yjp34EFJqUGpdnzzFCPJs2IaX4V97jdQlSI=",
      "keySize": 16,
      "purpose": "idm.password.encryption",
      "iv": "A4pIiY6kG6t0uLyLmJAoWQ==",
      "mac": "sFDJqg0Mmp0Ftl+1q1Bjzw=="
    }
  }
}
```

The following example uses the `openidm.decrypt` function to decrypt the password value:

```
curl \
--header "X-OpenIDM-Username: openidm-admin" \
--header "X-OpenIDM-Password: openidm-admin" \
--header "Content-Type: application/json" \
--cacert ca-cert.pem \
--request POST \
--data '{
  "type": "text/javascript",
  "globals": {
    "val": {
      "$crypto": {
        "type": "x-simple-encryption",
        "value": {
          "cipher": "AES/CBC/PKCS5Padding",
          "stableId": "openidm-sym-default",
          "salt": "qAS/eG7zdnFyK5H8lXvqTA==",
          "data": "zewf6hR1yjp34EFJqUGpdnzzFCPJs2IaX4V97jdQlSI=",
          "keySize": 16,
          "purpose": "idm.password.encryption",
          "iv": "A4pIiY6kG6t0uLyLmJAoWQ==",
          "mac": "sFDJqg0Mmp0Ftl+1q1Bjzw=="
        }
      }
    }
  },
  "source":"openidm.decrypt(val);"
}' \
"https://localhost:8443/openidm/script?_action=eval"
{
  "myKey": "myPassword"
}
```

For more information about the `openidm.encrypt` and `openidm.decrypt` functions, see
openidm.encrypt(value, cipher, alias) and openidm.decrypt(value) in the *Scripting Guide*.

Securing the Repository

Configuration data and, in most deployments, user data, are stored in the IDM repository. In
production deployments, you must secure access to the repository, and encrypt sensitive stored data.

For JDBC repositories, use a strong password for the connection to the repository and change at least
the password of the database user (`openidm` by default). When you change the database username and/
or password, update your database connection configuration file (`conf/datasource.jdbc-default.json`).

For a DS repository, change the `bindDN` and `bindPassword` for the directory server user in the
`ldapConnectionFactories` property in the `repo.ds.json` file.

In both cases, the password is encrypted on server startup, using the key specified in the `idm.password.`
`encryption` secret ID in `conf/secrets.json`.

Protecting Sensitive Files and Directories

Protect IDM files from access by unauthorized users. In particular, prevent other users from reading files in at least the `openidm/resolver/` 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 Sensitive Files in Unix

1. Make sure that user and group ownership of the installation and project directories is limited to the user running the IDM 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 Sensitive Files in Windows

1. The IDM process in Windows is normally run by the `Local System` service account.

2. If you are concerned about the security of this account, you can set up a service account that only has permissions for IDM-related directories, then remove User access to the directories noted above. You should also configure the service account to deny local and remote login. For more information, see the User Rights Assignment article in Microsoft's documentation.

Removing or Protecting Development and Debug Tools

Before you deploy IDM 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 IDM.

+ *Remove Tools*

To remove the Felix web console:

1. Remove the web console bundle and all of the plugin bundles related to the web console:

   ```
   rm /path/to/openidm/bundle/org.apache.felix.webconsole*.jar
   rm /path/to/openidm/bundle/openidm-felix-webconsole-7.0.0.jar
   ```

2. Remove the `felix.webconsole.json` configuration file from your project's `conf/` directory:

```
rm /path/to/project-dir/conf/felix.webconsole.json
```

+ *Protect Tools*

To protect access to the Felix web console, change the credentials in your project's `conf/felix.webconsole.json` file. These properties can be set using property substitution in the *Setup Guide*. This file contains the username and password to access the console, by default:

```
{
    "username" : "&{openidm.felix.webconsole.username|admin}",
    "password" : "&{openidm.felix.webconsole.password|admin}"
}
```

Adjusting Log Levels

In production, set log levels to `INFO` to ensure that you capture enough information to help diagnose issues, but do not expose unnecessary information. For more information, see "*Configure Server Logs*" in the *Monitoring Guide*.

At start up and shut down, `INFO` can produce many messages. During stable operation, `INFO` generally results in log messages only when coarse-grain operations such as scheduled reconciliation start or stop.

> **Important**
>
> The default IDM log formatter encodes all control characters (such as newline characters) using URL-encoding, to protect against log forgery. For more information, see "*Configure Server Logs*" in the *Monitoring Guide*.

Securing the API Explorer

The "*REST API Explorer*" serves up interactive REST API documentation. The API Explorer can help you identify endpoints, and run REST calls against those endpoints. To protect production servers from unauthorized API descriptor requests, IDM requires authentication, by default. The property `authEnabled` protects static web resources from public view.

+ *Default ui.context-api.json file*

```
{
    "enabled" : true,
    "authEnabled" : true,
    "urlContextRoot" : "/api",
    "defaultDir" : "&{idm.install.dir}/ui/api/default",
    "extensionDir" : "&{idm.install.dir}/ui/api/extension"
}
```

To completely disable the API Explorer, set the following property in your `resolver/boot.properties` file:

```
openidm.apidescriptor.enabled=false
```

Hide Unused REST Endpoints

The two main use cases for IDM are data synchronization and user self-service.

If you are using IDM *only* to synchronize data sources, do not expose the server externally. In this case, all connections are initiated by IDM.

If you are using IDM *only* for user self-service, ensure that the server is placed behind a firewall or proxy, such as ForgeRock Identity Gateway. At a minimum, hide the `/admin` endpoint in the web interface via the proxy. Use the `conf/access.json` in the *Authentication and Authorization Guide* file as a guide for proxy or firewall rules.

If you are using IDM for data synchronization *and* user self-service, it is preferable to run two IDM servers or clusters, each with its own security model. Because the two use cases have very different load characteristics and security implications, running them on separate servers can help to prevent synchronization activity from impacting the performance on end-user systems.

Disabling Automatic Configuration Updates

By default, IDM 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 IDM reads its configuration only from the repository.

> **Important**
>
> Before you disable automatic polling, you must have started the server at least once to ensure that the configuration has been loaded into the repository. Be aware, if automatic polling is enabled, IDM 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
```

Securing IDM Server Files With a Read-Only Installation

One method of locking down the server is to install IDM on a read-only file system.

This section assumes that you have prepared the read-only volume appropriate for your Linux/UNIX installation environment and that you have set up a regular Linux user named `idm` and a dedicated volume for the `/idm` directory.

Configure the dedicated volume device, `/dev/volume` in the `/etc/fstab` file, as follows:

```
/dev/volume    /idm    ext4    ro,defaults    1,2
```

When you run the **mount -a** command, the `/dev/volume` volume device is mounted on the `/idm` directory.

You can switch between read-write and read-only mode for the `/idm` volume with the following commands:

```
sudo mount -o remount,rw /idm
sudo mount -o remount,ro /idm
```

Confirm the result with the **mount** command, which should show that the `/idm` volume is mounted in read-only mode:

```
/dev/volume on /idm type ext4 (ro)
```

Set up the `/idm` volume in read-write mode:

```
sudo mount -o remount,rw /idm
```

With the following commands, you can unpack the IDM binary in the `/idm` directory, and give user `idm` ownership of all files in that directory:

```
sudo unzip /idm/IDM-7.1.0.zip
sudo chown -R idm.idm /idm
```

When you have installed IDM on a read-only file system, redirect audit and logging data to writable volumes. This procedure assumes a user `idm` with Linux administrative (superuser) privileges.

1. Create an external directory where IDM can send logging, auditing, and internal repository information:

```
sudo mkdir -p /var/log/openidm/audit
sudo mkdir /var/log/openidm/logs
sudo mkdir -p /var/cache/openidm/felix-cache
sudo mkdir /var/run/openidm
```

   Alternatively, route audit data to a remote data store. For an example of how to send audit data to a MySQL repository, see "*Direct Audit Information To MySQL*" in the *Samples Guide*.

2. Give the `idm` user ownership of the newly created directories:

```
sudo chown -R idm.idm /var/log/openidm
sudo chown -R idm.idm /var/cache/openidm
sudo chown -R idm.idm /var/run/openidm
```

3. Modify the following configuration files:

   conf/audit.json

   Make sure the `handlerForQueries` is the JSON audit event handler and change the `logDirectory` property to the `/var/log/openidm/audit` subdirectory:

```
"eventHandlers" : [
    {
        "class" : "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
        "config" : {
            "name" : "json",
            "logDirectory" : "/var/log/openidm/audit",
            ...
        },
        ...
    }
]
```

   conf/logging.properties

   Change the `java.util.logging.FileHandler.pattern` property as follows:

```
java.util.logging.FileHandler.pattern = /var/log/openidm/logs/openidm%u.log
```

   conf/config.properties

   Activate and redirect the `org.osgi.framework.storage` property as follows:

```
# If this value is not absolute, then the felix.cache.rootdir controls
# how the absolute location is calculated. (See buildNext property)
org.osgi.framework.storage=&{felix.cache.rootdir|&{user.dir}}/felix-cache

# The following property is used to convert a relative bundle cache
# location into an absolute one by specifying the root to prepend to
# the relative cache path. The default for this property is the
# current working directory.
felix.cache.rootdir=/var/cache/openidm
```

> **Note**
>
> You might want to set up additional redirection for the following:
>
> - Connectors. Depending on the connector, and the read-only volume, consider configuring connectors to direct output to writable volumes.
>
> - Scripts. If you are using Groovy, examine the `conf/script.json` file for your project. Make sure that output such as to the `groovy.target.directory` is directed to an appropriate location, such as `idm.data.dir`.

Adjust the value of the `OPENIDM_PID_FILE` in the `startup.sh` and `shutdown.sh` scripts.

For RHEL 6 and Ubuntu 14.04 systems, the default shell is bash. You can set the value of `OPENIDM_PID_FILE` for user `idm` by adding the following line to `/home/idm/.bashrc`:

```
export OPENIDM_PID_FILE=/var/run/openidm/openidm.pid
```

If you have set up a different command line shell, adjust your changes accordingly.

When you log in again as user `idm`, your `OPENIDM_PID_FILE` variable should redirect the process identifier file, `openidm.pid` to the `/var/run/openidm` directory, ready for access by the `shutdown.sh` script.

While the volume is still mounted in read-write mode, start IDM normally:

```
./startup.sh -p project-dir
```

The first startup of IDM either processes the signed certificate that you added, or generates a self-signed certificate, and encrypts any passwords in the various configuration files.

Stop IDM if it is running.

You can now mount the `/idm` directory in read-only mode. The configuration in `/etc/fstab` ensures that Linux mounts the `/idm` directory in read-only mode the next time that system is booted.

```
sudo mount -o remount,ro /idm
```

You can now start IDM, configured on a secure read-only volume.

```
./startup.sh -p project-dir
```