OpenICF Developer's Guide

Version 1.5

Lana Frost
László Hordós
Mark Craig

ForgeRock AS
33 New Montgomery St.,
Suite 1500
San Francisco, CA 94105, USA
+1 415-599-1100 (US)
www.forgerock.com

Abstract

Hands-on guide to developing and configuring OpenICF. The Open Identity Connector Framework (OpenICF) provides connectors for a consistent generic layer between applications and target resources.

Table of Contents

Preface

This guide shows you how to use and develop OpenICF connectors, which decouple applications from data resources.

1 Who Should Use this Guide

This guide is written for Java, C#, and web developers who use OpenICF to connect to resources from their applications, and who build their own OpenICF connectors and connector servers.

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 core 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.

Core documentation therefore follows a three-phase review process designed to eliminate errors:

- Product managers and software architects review project documentation design with respect to the readers' software lifecycle needs.

- Subject matter experts review proposed documentation changes for technical accuracy and completeness with respect to the corresponding software.

- Quality experts validate implemented documentation changes for technical accuracy, completeness in scope, and usability for the readership.

The review process helps to ensure that documentation published for a ForgeRock release is technically accurate and complete.

Fully reviewed, published core documentation is available at http://backstage.forgerock.com/. Use this documentation when working with a ForgeRock Identity Platform release.

4 Joining the ForgeRock Community

Visit the Community resource center where you can find information about each project, download trial builds, browse the resource catalog, ask and answer questions on the forums, find community events near you, and find the source code for open source software.

Chapter 1

About the OpenICF Framework and ICF Connectors

The Open Identity Connector Framework (OpenICF) project provides interoperability between identity, compliance and risk management solutions. An OpenICF Connector enables provisioning software, such as OpenIDM, to manage the identities that are maintained by a specific identity provider.

OpenICF connectors provide a consistent layer between identity applications and target resources, and expose a set of operations for the complete lifecycle of an identity. The connectors provide a way to decouple applications from the target resources to which data is provisioned.

OpenICF focuses on provisioning and identity management, but also provides general purpose capabilities, including authentication, create, read, update, delete, search, scripting, and synchronization operations. Connector bundles rely on the OpenICF Framework, but applications remain completely separate from the connector bundles. This enables you to change and update connectors without changing your application or its dependencies.

Many connectors have been built within the OpenICF framework, and are maintained and supported by ForgeRock and by the OpenICF community. However, you can also develop your own OpenICF connector, to address a requirement that is not covered by one of the existing connectors. In addition, OpenICF provides two *scripted connector toolkits*, that enable you to write your own connectors based on Groovy or PowerShell scripts.

Version 1.5 of the OpenICF framework can use OpenIDM, Sun Identity Manager, and Oracle Waveset connectors (version 1.1) and can use ConnID connectors up to version 1.4.

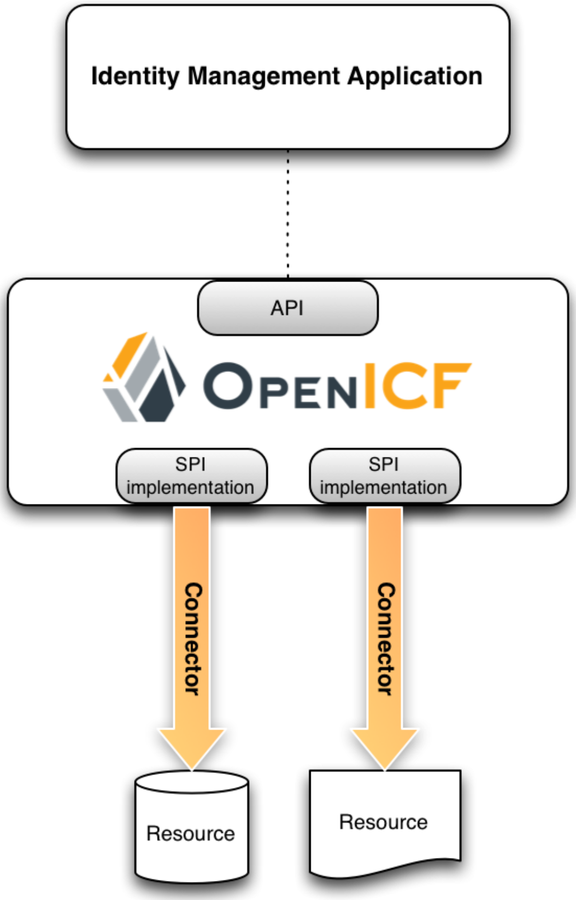
This guide provides the following information:

- An overview of the OpenICF framework and its components
- Information on how to use the OpenICF existing connectors in your application (both locally and remotely)
- Information on how to write your own Java and .NET connectors, scripted Groovy connectors, or scripted PowerShell connectors

1.1 Overview of the OpenICF Architecture

OpenICF is situated between the identity management application and the target resource. The framework provides a generic layer between the application and the connector bundle that accesses the resource. The framework implements an API, that includes a defined set of operations. When you are building a connector, you implement the Service Provider Interface (SPI), and include only those operations that are supported by your target resource. Each connector implements a set of SPI operations. The API operations call the SPI operations that you implement.

The following image shows a high-level overview of an OpenICF deployment.



1.1.1 Understanding the OpenICF Framework Components

When you are building, or modifying, an identity management application to use the OpenICF Framework and its connectors, you use the following interfaces of the API:

- Connector Info Manager Component

  The connector info manager maintains a set of connector info instances, each of which describes an available connector. The OpenICF Framework provides three different types of connector info manager:

  - Local

    A local connector info manager accesses the connector bundle or assembly directly.

  - Remote

    A remote connector info manager accesses the connector bundle or assembly through a remote connector server.

  - OSGi

    An OSGi connector info manager accesses the connector bundle within the OSGi context.

  For more information, see *Creating the Connector Info Manager*.

- Connector Info Component

  The connector info component provides meta information (display name, category, messages, and so forth) for a given connector.

- Connector Key Component

  The connector key component uniquely identifies a specific connector implementation.

- API Configuration

  The API configuration holds the available configuration properties and values from both the API, and the SPI, based on the connector type.

  For more information, see *Implementing the Configuration Interface*.

- Connector Facade Interface

  The connector facade is the main interface through which an application invokes connector operations. The connector facade represents a specific connector instance, that has been configured in a specific way. For more information, see *Creating a Connector Facade*.

When you are building a new connector, you implement the SPI, including the following interfaces:

- The `connector` interface.

  The connector interface handles initialization and disposal of the connector, and determines whether the connector is poolable. For more information, see *Implementing the Connector Interface*

- The `configuration` interface.

  The configuration interface implementation includes all of the required information to enable the connector to connect to the target system, and to perform its operations. The configuration interface implements getters and setters for each of its defined properties. It also provides a validate method that determines whether all the required properties are available, and valid. For more information, see *Implementing the Configuration Interface*.

  The OpenICF framework uses the configuration interface implementation to build the *configuration properties* inside the API configuration.

  When the configuration interface is implemented, it becomes available to the default API configuration.

- Any `operations` that the target resource can support, such as `CreateOp`, `UpdateOp`, `DeleteOp` and so forth. For more information, see *Implementing the Operation Interfaces*.

1.2 Overview of a Remote Connector Implementation

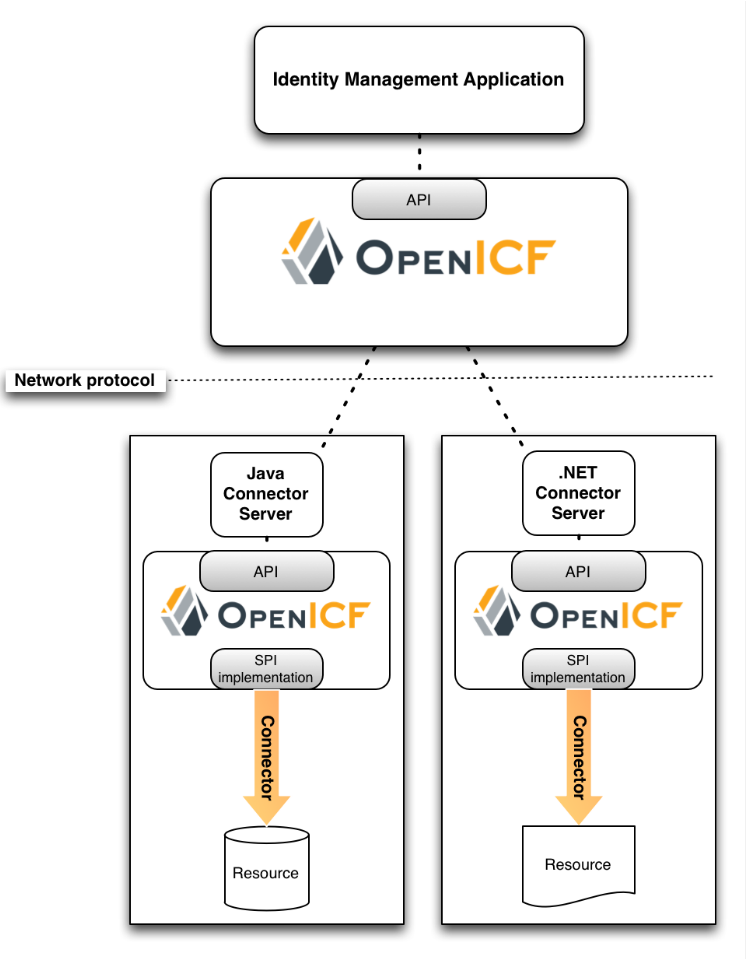
Connectors can run locally (on the same host as your application) or remotely (on a host that is remote to your application). Connectors that run remotely require a *connector server*, running on the same host as the connector. Applications access the connector implementation *through* the connector server.

Note

The OpenICF framework can support both local and remote connector implementations simultaneously.

Connector servers also enable you to run connector bundles that are written in C# on a .NET platform, and to access them over the network from a Java or .NET application.

The following image shows a high-level overview of an OpenICF deployment, including a remote connector server.



For more information about connector servers, and how to use them in your application, see *Using OpenICF Connector Servers to Run Connectors Remotely*.

1.3 Overview of OpenICF Functionality

OpenICF provides many capabilities, including the following:

- Connector pooling
- Timeouts on all operations

- Search filtering
- Search and synchronization buffering and result streaming
- Scripting with Groovy, JavaScript, shell, and PowerShell
- Classloader isolation
- An independent logging API/SPI
- Java and .NET platform support
- Opt-in operations that support both simple and advanced implementations for the same CRUD operation
- A logging proxy that captures all API calls
- A Maven connector archetype to create connectors

Chapter 2
Using the OpenICF API

This chapter describes how to use the OpenICF API, which enables you to call OpenICF connector implementations from your application. The chapter demonstrates creating a connector facade, required for applications to access connectors, and then how to call the various OpenICF operations from your application.

2.1 Before You Start Using the OpenICF API

Before you can use an OpenICF connector in your application, you must download the OpenICF framework libraries, and the required connector bundles.

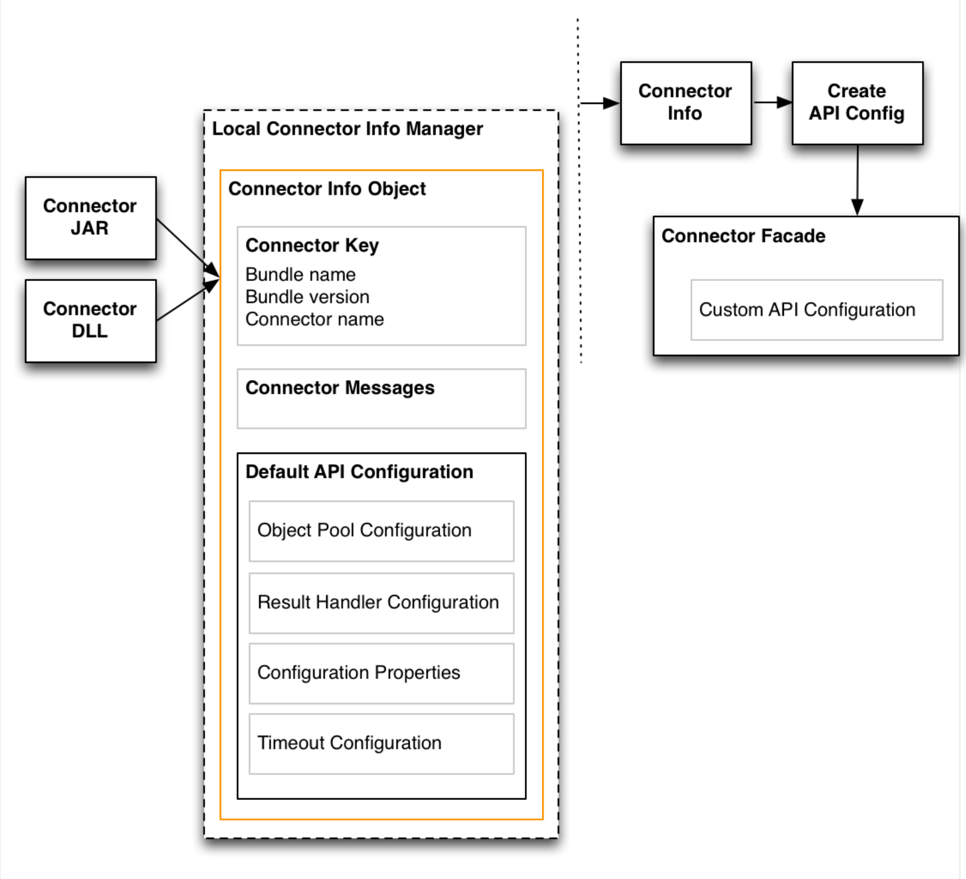
The easiest way to start using the OpenICF framework, from Java, is to use the sample Maven project file as a starting point. This sample project includes comprehensive comments about its use.

To use a .NET connector remotely, you must install the .NET remote connector server, as described in *Installing, Configuring and Running a .NET Connector Server*. You must also download and install the specific connector assemblies that you want to use, from the ForgeRock Resource Catalog page.

You can now start integrating the connector with your application.

2.2 About the Connector Facade

An application interacts with a connector through an instance of the `ConnectorFacade` class. The following diagram shows the creation and configuration of the connector facade. The components shown here are described in more detail in the sections that follow.



The connector facade is instantiated and configured in the following steps:

1. The application creates a `LocalConnectorInfoManager` instance (or instances) and adds the individual connector bundles (or assemblies).

   The `LocalConnectorInfoManager` processes these bundles or assemblies to instantiate a `ConnectorInfo` object.

To be processed by the connector info manager, the connector bundle or assembly must have the following characteristics:

Java Connector Bundle

The `META-INF/MANIFEST.MF` file *must* include the following entries:

`ConnectorBundle-FrameworkVersion` - Minimum required OpenICF Framework version (either 1.1, 1.4, or 1.5)
`ConnectorBundle-Name` - Unique name of the connector bundle
`ConnectorBundle-Version` - Version of the connector bundle

The combination of the `ConnectorBundle-Name` and the `ConnectorBundle-Version` must be unique.

The connector bundle JAR must contain at least one class, that has the `ConnectorClass` annotation and implements the `Connector` interface.

.NET Connector Assembly

The `AssemblyInfo.cs` is used to determine the bundle version, from the `AssemblyVersion` property.

The bundle name is derived from the `Name` property of the assembly. For more information, see the corresponding Microsoft documentation.

Warning

If you change the name of your assembly, you must adjust the `bundleName` property in your connector configuration file, accordingly.

The connector assembly DLL must contain at least one class, that has the `ConnectorClassAttribute` attribute and implements the `Connector` interface.

2. For each connector, the `LocalConnectorInfoManager` processes the `MessageCatalog`, which contains the localized help and description messages for the configuration, and any log or error messages for the connector.

   Your application can use this information to provide additional help during the connector configuration process.

3. For each connector, the `LocalConnectorInfoManager` then processes the `ConfigurationClass`, to build the configuration properties for the connector.

4. Your application finds the connector info by its *connector key*. When the application has the connector info, it creates an API Configuration object that customises the following components:

   - Object pool configuration
   - Result handler configuration
   - Configuration properties
   - Timeout configuration

   The API Configuration object is described in more detail in Section 2.4, "The API Configuration Object".

5. The `ConnectorFacade` takes this customized API configuration object, determines which connector to use and how to configure it, and implements all of the OpenICF API operations.

2.3 The Connector Messages Object

The Connector Messages interface sets the message catalog for each connector, and enables messages to be localized. The interface has one method (`format()`), which formats a given message key in the current locale.

For more information, see the corresponding Javadoc.

2.4 The API Configuration Object

The API Configuration Object holds the runtime configuration of the connector facade instance. The OpenICF framework creates a default API Configuration Object inside the Connector Info Object. The application creates a copy of the API Configuration Object and customises it according to its requirements. The API Configuration Object includes the following components:

Object Pool Configuration
> The object pool configuration specifies the pool configuration for poolable connectors only. Non-poolable connectors ignore this parameter. The object pool configuration includes the following parameters:
>
> 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.

Results Handler Configuration

The results handler configuration defines how the OpenICF framework chains together the different results handlers to filter search results.

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 this property allows you to enable case insensitive filtering. When case insensitive filtering is not enabled, 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 that 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, it is recommended that you set this property to `false` for local connectors, and to `true` for remote connectors.

Configuration Properties

The Configuration Properties object is built and populated by the framework as it parses the connectors configuration class.

Timeout Configuration

The timeout configuration enables you to configure timeout values per operation type. By default, there is no timeout configured for any operation type.

2.5 Creating the Connector Info Manager

You must initiate a specific connector info manager type, depending on whether your connector is local or remote. The following samples show how to create a local connector info manager and a remote connector info manager.

1. Create a `ConnectorInfoManager` and a `ConnectorKey` for the connector.

   The `ConnectorKey` uniquely identifies the connector instance. The `ConnectorKey` class takes a `bundleName` (the name of the Connector bundle), a `bundleVersion` (the version of the Connector bundle) and a connectorName (the name of the Connector)

   The `ConnectorInfoManager` retrieves a `ConnectorInfo` object for the connector by its connector key.

   Example 2.1. Acquiring a Local Connector Info Object (Java)

   ```
   ConnectorInfoManagerFactory fact = ConnectorInfoManagerFactory.getInstance();
   File bundleDirectory = new File("/connectorDir/bundles/myconnector");
   URL url = IOUtil.makeURL(bundleDirectory,
        "/dist/org.identityconnectors.myconnector-1.0.jar");
   ConnectorInfoManager manager = fact.getLocalManager(url);
   ConnectorKey key = new ConnectorKey("org.identityconnectors.myconnector",
        "1.0", "MyConnector");
   ```

   Example 2.2. Acquiring a Remote Connector Info Object (Java)

   ```
   ConnectorInfoManagerFactory fact = ConnectorInfoManagerFactory.getInstance();
   File bundleDirectory = new File("/connectorDir/bundles/myconnector");
   URL url = IOUtil.makeURL(bundleDirectory,
        "/dist/org.identityconnectors.myconnector-1.0.jar");
   ConnectorInfoManager manager = fact.getLocalManager(url);
   ConnectorKey key = new ConnectorKey("org.identityconnectors.myconnector",
        "1.0", "MyConnector");
   ```

2.6 Creating the Connector Facade

Applications access the connector API through a `ConnectorFacade` class, and interact with the connector through a `ConnectorFacade` instance.

The following steps describe how to create a `ConnectorFacade` in your application.

1. Create a `ConnectorInfoManager` and acquire the `ConnectorInfo` object for your connector, as described in the previous section.

2. From the `ConnectorInfo` object, create the default `APIConfiguration`.

```
APIConfiguration apiConfig = info.createDefaultAPIConfiguration();
```

3. Use the default `APIConfiguration` to set the `ObjectPoolConfiguration`, `ResultsHandlerConfiguration`, `ConfigurationProperties`, and `TimeoutConfiguration`.

```
ConfigurationProperties properties = apiConfig.getConfigurationProperties();
```

4. Set all of the `ConfigurationProperties` that you need for the connector, using `setPropertyValue()`.

```
properties.setPropertyValue("host", SAMPLE_HOST);
properties.setPropertyValue("adminName", SAMPLE_ADMIN);
properties.setPropertyValue("adminPassword", SAMPLE_PASSWORD);
properties.setPropertyValue("useSSL", false);
```

5. Use the `newInstance()` method of the `ConnectorFacadeFactory` to create a new instance of the connector.

```
ConnectorFacade conn = ConnectorFacadeFactory.getInstance()
        .newInstance(apiConfig);
```

6. Validate that you have set up the connector configuration correctly.

```
conn.validate();
```

7. Use the new connector with the supported operations (described in the following sections).

```
conn.[authenticate|create|update|delete|search|...]
```

2.7 Checking the Schema and the Supported Operations

Different connectors support different subsets of the overall set of operations provided by OpenICF. When your connector is ready to use, you can use the `ConnectorFacade` to determine which operations your connector supports.

The quickest way to check whether an operation is supported is to determine whether that specific operation is part of the set of supported operations. The following sample test checks if the `CreateApiOp` is supported:

```
Set<Class< ? extends APIOperation>> ops = conn.getSupportedOperations();
return ops.contains(CreateApiOp.class);
```

Note that a connector might support a particular operation, only for specific object classes. For example, the connector might allow you to *create* a user, but not a group.

To be able to determine the list of supported operations for each object class, you need to check the schema. To determine whether the connector supports an operation for a specific object class, check the object class on which you plan to perform the operation, as shown in the following example.

```
Schema schema = conn.schema();
Set<ObjectClassInfo> objectClasses = schema.getObjectClassInfo();
Set<ObjectClassInfo> ocinfos = schema
        .getSupportedObjectClassesByOperation(CreateApiOp.class);

for(ObjectClassInfo oci : objectClasses) {
    // Check that the operation is supported for your object class.
    if (ocinfos.contains(ocinfo)) {
        // object class is supported
    }
}
```

In addition to determining the supported operations for an object class, your application can check which attributes are *required* and which attributes are *allowed* for a particular object class. The `ObjectClassInfo` class contains this information as a set of `AttributeInfo` objects.

The following example shows how to retrieve the attributes for an object class.

```
Schema schema = conn.schema();
Set<ObjectClassInfo> objectClasses = schema.getObjectClassInfo();
for(ObjectClassInfo oci : objectClasses) {
    Set<AttributeInfo> attributeInfos = oci.getAttributeInfo();
    String type = oci.getType();
    if(ObjectClass.ACCOUNT_NAME.equals(type)) {
        for(AttributeInfo info : attributeInfos) {
            System.out.println(info.toString());
        }
    }
}
```

Using the schema object, you can obtain the following information:

- Object classes and their attributes
- Operation options per operation

The following example shows how to retrieve the schema as a list of `ObjectClass` objects, from the `ObjectClassInfo` class.

```
ObjectClass objectClass = new ObjectClass(objectClassInfo.getType());
```

2.7.1 Operation Options

Operation options provide an extension point to an operation, enabling you to request additional information from the application, for each operation. The connector framework includes a number of predefined operation options for the most common use cases. For example, the option `OP_ATTRIBUTES_TO_GET` enables you to specify a list of attributes that should be returned by an operation. When you write a connector, you must define the operation options that your connector supports in the schema, so that the application knows which operation options are supported.

For a list of the predefined operation options, see the corresponding Javadoc.

2.7.2 OpenICF Special Attributes

OpenICF includes a number of *special* attributes, that all begin and end with __ (for example `__NAME__`, and `__UID__`). These special attributes are essentially functional aliases for specific attributes or object types. The purpose of the special attributes is to enable a connector developer 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.

The special attributes are used extensively in the generic LDAP connector, which 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.

For a list of the special attributes, see the corresponding Javadoc.

2.8 How the OpenICF Framework Manages Connector Instances

The OpenICF framework supports multiple *connector types*, based on the implementation of the `connector` interface, and the `configuration` interface. These two interfaces determine the following:

- Whether the connector instance is obtained from a pool or whether a new instance is created *for each operation*
- Whether the connector configuration instance is retained, and reused for each operation, (stateful configuration) or a new configuration instance is created for each operation (stateless).

Connector developers determine what type of connector to implement, assessing the best match for the resource to which they are connecting. The interaction between the `connector` and `configuration` interface implementations is described in detail in *Deciding on the Connector Type*. This section illustrates how the OpenICF framework manages connector instantiation, depending on the connector type.

2.8.1 Connector Instantiation for a Stateless, Non-Poolable Connector

The most basic connector has a stateless configuration, and is not pooled. A basic connector is initialized as follows:

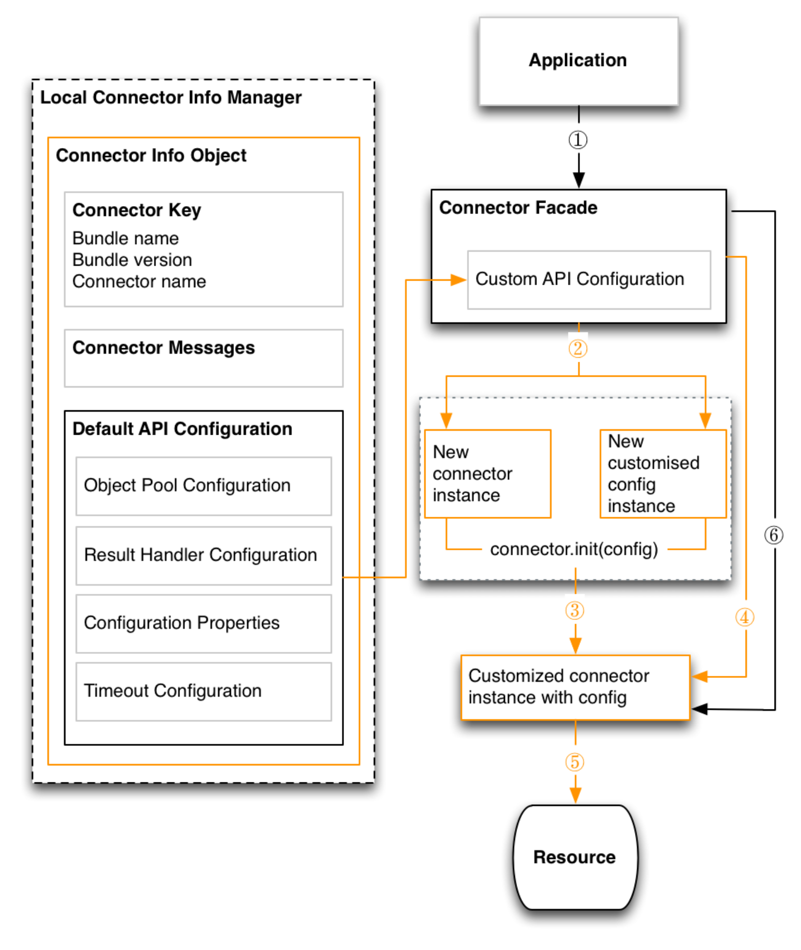
1. The application calls an operation (for example, CREATE) on the connector facade.
2. The OpenICF framework creates a new *configuration instance*, and initializes it with its configuration properties.
3. When the framework has the configuration instance, with all the attributes in the configuration set, the framework creates a new *connector instance*, and initializes it, with the configuration that has been set.

4. The framework executes the operation (for example, CREATE) on the connector instance.

5. The connector instance executes the operation on the resource.

6. The framework calls the `dispose()` method to release all resources that the connector instance was using.

The following illustration shows the initialization process for a basic connector, and references the numbered steps in the preceding list.



2.8.2 Connector Instantiation for a Stateless, Poolable Connector

The second connector type has a stateless configuration, but can be pooled. A stateless, poolable connector is instantiated as follows:

1. The application calls an operation (for example, CREATE) on the connector facade.

2. The OpenICF framework calls on the object pool, to borrow a *live* connector instance to execute the operation.

   If the object pool has an idle connector instance available, the framework *borrows* that one instance (step 5a in the illustration that follows).

   The framework calls the `checkAlive` method on the customized connector instance with its configuration, to check if the instance that was borrowed from the pool is still alive, and ready to execute the operation. If the instance is no longer alive and ready, the framework disposes of the instance and borrows another one.

   The thread that borrows the object has exclusive access to that connector instance, that is, it is thread-safe.

3. If the object pool has no idle connector instances, the pool creates a new connector instance.

4. The framework creates a new *configuration instance*, and initializes it with its configuration properties.

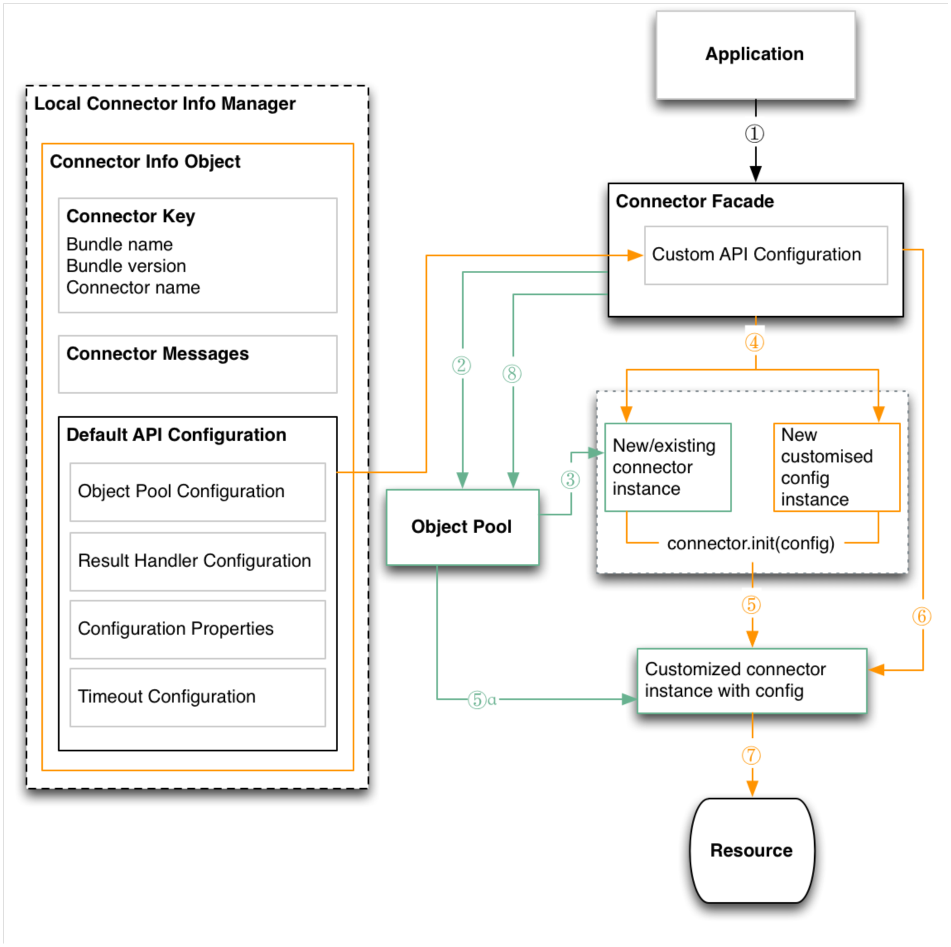
5. The framework initializes the borrowed connector instance, with the configuration that has been set.

6. The framework executes the operation (for example, CREATE) on the connector instance.

7. The connector instance executes the operation on the resource.

8. When the operation is complete, the framework releases the connector instance back into the pool. No `dispose()` method is called.

The following illustration shows the initialization process for a stateless, poolable connector, and references the numbered steps in the preceding list.



2.8.3 Connector Instantiation for a Stateful, Non-Poolable Connector

The third connector type has a stateful configuration, and cannot be pooled. A stateful, non-poolable connector is instantiated as follows:

1. The OpenICF framework creates a new *configuration instance*, initializes it with its configuration properties, and stores it in the connector facade, before any operations are called.

This single configuration instance is shared between multiple threads. The framework does not guarantee isolation, so connector developers must ensure that their implementation is thread-safe.

2. The application calls an operation (for example, CREATE) on the connector facade.

3. The OpenICF framework creates a new connector instance, and calls the `init()` method on that connector instance, with the stored configuration. In other words, the framework initializes the connector, with the single configuration instance stored within the connector facade.

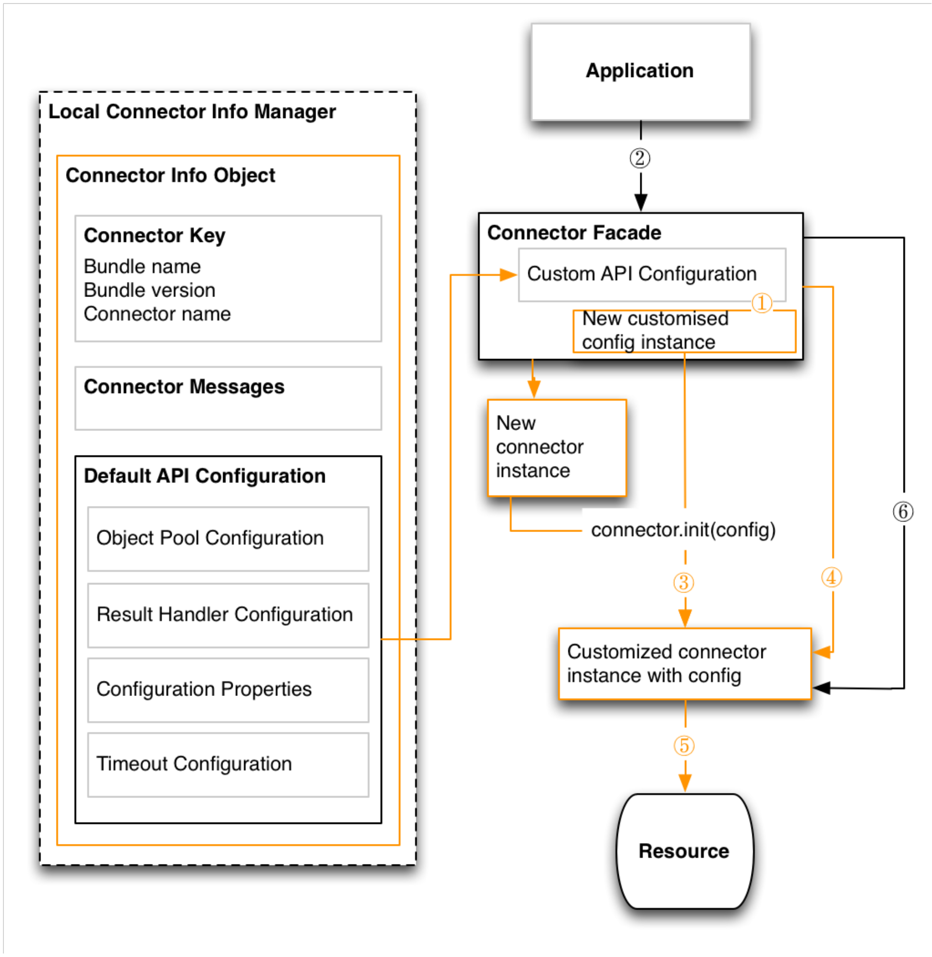
4. The framework executes the operation (for example, CREATE) on the connector instance.

5. The connector instance executes the operation on the resource.

6. The framework calls the `dispose()` method to release all resources that the connector instance was using.

   Note that the customized config instance remains in the connector facade, and is reused for the next operation.

The following illustration shows the initialization process for a non-poolable connector, with a stateful configuration. The illustration references the numbered steps in the preceding list.



2.8.4 Connector Instantiation for a Stateful, Poolable Connector

The fourth connector type has a stateful configuration, and can be pooled. A stateful, poolable connector is instantiated as follows:

1. The OpenICF framework creates a new *configuration instance*, initializes it with its configuration properties, and stores it in the connector facade, before any operations are called.

This single configuration instance is shared between multiple threads. The framework does not guarantee isolation, so connector developers must ensure that their implementation is thread-safe.

2. The application calls an operation (for example, CREATE) on the connector facade.

3. The framework calls on the object pool, to borrow a *live* connector instance to execute the operation.

   If the object pool has an idle connector instance available, the framework *borrows* that one instance (step 5a in the illustration that follows).

   The framework calls the `checkAlive` method on the customized connector instance with its configuration, to check if the instance that was borrowed from the pool is still alive, and ready to execute the operation. If the instance is no longer alive and ready, the framework disposes of the instance and borrows another one.

   The thread that borrows the object has exclusive access to that connector instance, that is, it is thread-safe.

4. If the object pool has no idle connector instances, the pool creates a new connector instance.

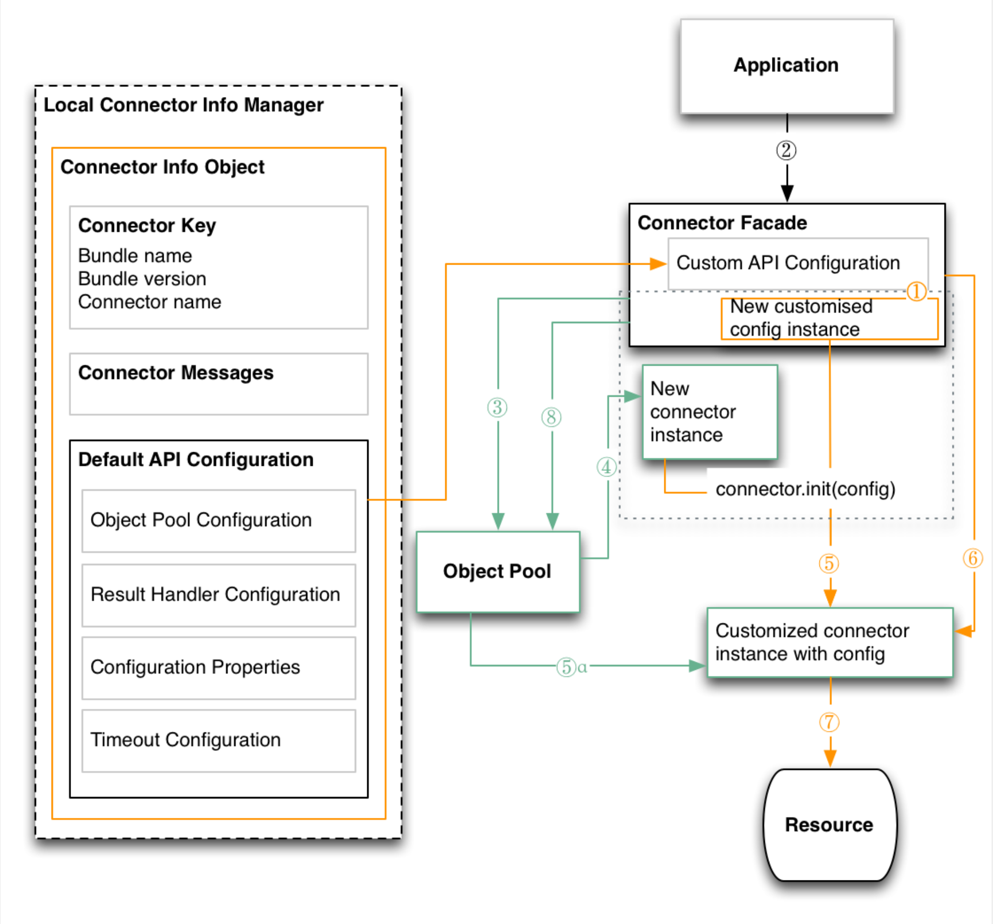
5. The framework initializes the borrowed connector instance, with the stored configuration.

6. The framework executes the operation (for example, CREATE) on the connector instance.

7. The connector instance executes the operation on the resource.

8. When the operation is complete, the framework releases the connector instance back into the pool. No `dispose()` method is called.

The following illustration shows the initialization process for a stateful, poolable connector, and references the numbered steps in the preceding list.


Application
②
Local Connector Info Manager
Connector Info Object
Connector Key
Bundle name
Bundle version
Connector name
Connector Messages
Default API Configuration
Object Pool Configuration
Result Handler Configuration
Configuration Properties
Timeout Configuration
Connector Facade
Custom API Configuration
①
New customised config instance
③
⑧
④
New connector instance
connector.init(config)
Object Pool
⑤
⑥
⑤a
Customized connector instance with config
⑦
Resource

Chapter 3

Using OpenICF Connector Servers to Run Connectors Remotely

A *Connector Server* enables your application to run provisioning operations on a connector bundle that is deployed on a remote system. A key feature of connector servers is the ability to run connector bundles that are written in C# on a .NET platform, and to access them over the network from a Java application.

Connector servers are available for both the Java and .NET platforms.

- A .NET connector server allows Java applications to access C# connector bundles. You deploy the C# connector bundles on the .NET connector server. Your Java application can then communicate with the .NET connector server over the network. The .NET connector server acts as a proxy to provide authenticated access for Java applications to the C# connector bundles that are deployed within the connector server.

- A Java connector server allows you to execute a Java connector bundle in a different JVM from your application. You can also run a Java connector on a different host for performance reasons. You might choose to use a Java connector server with a Java remote connector server to avoid the possibility of crashing an application JVM, due to a fault in a JNI-based connector.

3.1 Installing, Configuring and Running a .NET Connector Server

The .NET connector server requires the .NET framework (version 4.5 or later) and is supported on Windows Server 2008 and 2008 R2.

By default, the connector server outputs log messages to a file named `connectorserver.log`, in the installation directory. To change the location of the log file, set the `initializeData` parameter in the configuration file, *before* before you start 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>
```

For more information about adjusting the logging parameters, see Logging and Troubleshooting.

Procedure 3.1. Installing a .NET Connector Server

1. Download the OpenICF .NET Connector Server from the OpenICF Downloads page.

   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, and move on to the next procedure, Procedure 3.2, "Configuring a .NET Connector Server". Otherwise, follow the steps in this section.

2. Execute the openicf-zip--dotnet.msi installation file and complete the wizard.

   When the wizard has completed, the Connector Server is installed as a Windows Service.

3. Open the Services console and make sure that the Connector Server is listed there.

   The name of the service is OpenICF Connector Server, by default.

Procedure 3.2. Configuring a .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 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)\Identity Connectors\Connector Server"
   ```

3. Run the **ConnectorServer /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`.

   ```
   ConnectorServer /setkey Passw0rd
   Key has been successfully updated.
   ```

   This key is used by clients that connect to the Connector Server.

4. Edit the Connector Server connection settings.

   The Connector Server configuration is saved in a file named `ConnectorServer.exe.Config` (in the directory in which the Connector Server is installed).

   Check and edit this file, as necessary, to reflect your installation. In particular, check the connection properties, under the `<appsettings>` item.

   ```
   <add key="connectorserver.port" value="8759" />
   <add key="connectorserver.usessl" value="false" />
   <add key="connectorserver.certificatestorename" value="ConnectorServerSSLCertificate" />
   <add key="connectorserver.ifaddress" value="0.0.0.0" />
   <add key="connectorserver.key" value="xOS4IeeE6eb/AhMbhxZEC37PgtE=" />
   ```

   The following connection properties are set by default.

   `connectorserver.port`
   : Specifies the port on which the Connector Server listens.

   Note

   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.

`connectorserver.usessl`

Indicates whether client connections to the Connector Server should be over SSL.

Warning

This property is set to `false` by default. In production environments you must set this property to `true` for security reasons, otherwise all messages going over the network are obfuscated rather than encrypted.

To secure connections to the Connector Server, set this property to `true` and store the server certificate in your certificate store, using the following command:

```
ConnectorServer /storeCertificate /storeName <certificate-store-name> /certificateFile <certifica
```

`connectorserver.certificatestorename`

Specifies the name of the certificate store, into which your server certificate has been installed.

This parameter is optional. If you do not set it, the default certificate store name is `ConnectorServerSSLCertificate`.

`connectorserver.ifaddress`

Specifies a single IP address from which connections will be accepted.

If you set a value here (other than the default `0.0.0.0`) connections from all IP addresses other than the one specified are denied.

Record the following information for use when your application connects to the Connector Server.

- Host name or IP address

- Connector server port

   - Connector server key

   - Whether SSL is enabled

5. Check the trace settings, in the same configuration file, under the `<system.diagnostics>` item.

   ```
   <system.diagnostics>
     <trace autoflush="true" indentsize="4">
       <listeners>
         <remove name="Default" />
         <add
             name="myListener"
             type="System.Diagnostics.TextWriterTraceListener"
             initializeData="c:\connectorserver.log"
             traceOutputOptions="DateTime">
             <filter
               type="System.Diagnostics.EventTypeFilter"
               initializeData="Information" />
         </add>
       </listeners>
     </trace>
   </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, you can change the EventTypeFilter's initializeData to "Warning" or "Error". For very verbose logging you can set the value to "Verbose" or "All". The level of logging performed has a direct effect on the performance of the Connector Servers, so take care when setting this level.

   For more information on adjusting the logging and trace settings, see Logging and Troubleshooting.

Procedure 3.3. 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 (`services.msc`).

   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 service user (`System`, by default). In production mode, you must change this to a user with more restrictions.

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 most cases, you should run the Connector Server as a Windows service, but you can run the Connector Server manually, as follows.

   In the Windows Command Prompt, change directory to the location where the Connector Server was installed. The default location is `c:\Program Files (x86)\Identity Connectors\Connector Server`.

   Start the server with the following command:

   ```
   ConnectorServer.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.

Procedure 3.4. Installing Connectors on a .NET Connector Server

When your .NET Connector Server is up and running, you must install the individual connector assemblies that will run on the Connector Server. This section assumes that your connector assemblies are available as .zip files, either downloaded from the ForgeRock Resources Catalog, or packaged after their development.

1. Change to the directory where you installed the connector server.

2. Unzip the connector .zip file.

   The .NET connector server scans the install directory for files named `*.Connector.dll` or `*.Connector-*.dll`.

3. Restart the connector server.

Procedure 3.5. Running Multiple .NET Connector Servers

In some situations, you might want run multiple .NET connector servers on the same host. For example, you might want to run a service with different privileges, or multiple connectors that you are using might be incompatible when run in the same application.

To run multiple connector servers on the same host, adhere to the following steps:

1. Unpack the connector server binaries into different directories
2. Use different port numbers for each server
3. Use different trace files for each server

Each Connector Server can run interactively, or as a Windows Service - they do not need to be run in the same way.

3.2 Installing, Configuring and Running a Remote Java Connector Server

In certain situations, it might be necessary to set up a remote Java Connector Server, rather than run your Java connectors locally. This section provides instructions for setting up a remote Java Connector Server on Unix/Linux and Windows.

Procedure 3.6. Installing the Java Connector Server

1. Download and extract the OpenICF Java Connector Server .zip file from the ForgeRock Downloads page.
2. Start the Java Connector Server from the install directory.

```
$ /install-dir/bin/ConnectorServer.sh

 Usage: ConnectorServer <command> [option], where command is one of the following:
 /run ["-J<java option>"] - Runs the server from the console.
 /setKey [<key>] - Sets the connector server key.
 /setDefaults - Sets the default ConnectorServer.properties.

 example:
 ConnectorServer.sh /run "-J-Djavax.net.ssl.keyStore=mykeystore.jks" "-J-Djavax.net.ssl.keyStorePassword=changeit
 - this will run connector server with SSL

 ConnectorServer.sh jpda /run
 - this will run connector server in debug mode
```

   Your Java Connector Server is now up and running.

Procedure 3.7. Starting the Java Connector Server as a Service

1. For information on starting the Java Connector Server as a Linux service, see *Installing a Remote Java Connector Server for Unix/Linux* in the *OpenIDM Integrator's Guide*.

2. For information on starting the Java Connector Server as a Windows service, see *Installing a Remote Java Connector Server for Windows* in the *OpenIDM Integrator's Guide*.

Procedure 3.8. Configuring the Java Connector Server

The Java Connector Server uses a `key` property to authenticate the connection. The default key value is `changeit`.

1. To change the value of the secret key, run the Connector Server with the `-setKey` option. The following example sets the key value to `Passw0rd`:

```
$ cd /path/to/jconnsrv
$ java \
 -cp "./lib/framework/*" \
 org.identityconnectors.framework.server.Main \
 -setKey
 -key Passw0rd
 -properties ./conf/ConnectorServer.properties
```

2. Review the `ConnectorServer.properties` file in the `./conf` directory, and make any required changes. By default, the configuration file has the following properties:

```
connectorserver.port=8759
connectorserver.bundleDir=bundles
connectorserver.libDir=lib
connectorserver.usessl=false
connectorserver.loggerClass=org.forgerock.openicf.common.logging.slf4j.SLF4JLog
connectorserver.key=xOS4IeeE6eb/AhMbhxZEC37PgtE\=
```

`connectorserver.port`
The port number to listen on

`connectorserver.bundleDir`
The installation directory of the connectors. The path is relative to the connector server installation directory, and is `bundles` by default.

`connectorserver.libDir`
The directory in which runtime libraries that are required by the connectors are located. The default directory is `lib`.

The runtime libraries directory enables you to prevent library conflicts, by using Java classloader isolation.

`connectorserver.usessl`
Indicates whether client connections to the connector server should be over SSL. This property is set to `false` by default. You must set it to `true` in production environments.

To secure connections to the connector server, set this property to `true` and set the following system config properties (on the command line) before you start the connector server:

```
-Djavax.net.ssl.keyStore=mySrvKeystore
-Djavax.net.ssl.keyStoreType (optional)
-Djavax.net.ssl.keyStorePassword
```

`connectorserver.loggerClass`

Specifies the logging implementation.

OpenICF use its own logging API. The default implementation is to log everything to the console with (`org.identityconnectors.common.logging.StdOutLogger`). This implementation is fine in a development environment.

In production environments, you should use either the JDK (`org.identityconnectors.common.logging.impl.JDKLogger`) or the Simple Logging Facade for Java (SLF4J) (`org.forgerock.openicf.common.logging.slf4j.SLF4JLog`) implementations. For more information, see the Javadoc for the JDK Logger and the SLF4J.

`connectorserver.key`

Provides a secure hash of the connector server key. You can set this key by using the `-setkey` flag, as described previously.

`connectorserver.ifaddress`

This optional parameter enables you to specifies a single IP address from which connections will be accepted.

If you set a value here (other than the default `0.0.0.0`) connections from all IP addresses other than the one specified are denied.

Procedure 3.9. Running the Java Connector Server

1. Start the Java Connector Server as follows:

```
$ java \
 -cp "connector-framework.jar:connector-framework-internal.jar:groovy-all.jar" \
 org.identityconnectors.framework.server.Main \
 -run \
 -properties ./conf/connectorserver.properties
```

The connector server is now running, and listening on port 8759, by default.

Log files are available in the `/path/to/jconnsrv/logs` directory.

```
$ ls logs/
Connector.log  ConnectorServer.log  ConnectorServerTrace.log
```

2. If required, stop the Java Connector Server by pressing `q`.

Procedure 3.10. Installing Connectors on a Java Connector Server

When your Java Connector Server is up and running, you must install the individual connector bundles that will run on the Connector Server. This section assumes that your connector bundles are available as JAR files, either downloaded from the OpenICF download page, or bundled after their development.

1. Copy the connector bundle .jar to the `bundles/` folder.
2. Add any required third-party .jar files to the `lib/` folder.
3. Restart the Java Connector Server.

3.3 Accessing Connector Servers Over SSL

In production environments, your application should access either the .NET or remote Java Connector Server over SSL.

To configure secure access to the connector server, follow these steps:

1. Configure the Connector Server to use an SSL certificate.

   Edit the `ConnectorServer.properties` file (in the `/path/to/openicf/conf` folder), to set `connectorserver.usessl=true`.

2. Configure the Connector Server to provide SSL sockets.

   If you use self-signed or other certificates that Java cannot verify automatically, either import these certificates into the `$JAVA_HOME/lib/security/cacerts` directory, or use Java properties to specify your own, properly configured, truststore:

   - `-Djavax.net.ssl.trustStore=/path/to/myApp_cacerts`
   - `-Djavax.net.ssl.trustStorePassword=changeit`

Chapter 4

Implementing the OpenICF SPI

This chapter describes the OpenICF SPI, which enables you to create connectors that are compatible with the OpenICF framework.

The SPI includes a number of interfaces, but you need only implement those that are supported by the target resource to which you are connecting. For information about how to get started with writing connectors, see *Writing Java Connectors* and *Writing Scripted Connectors With the Groovy Connector Toolkit*.

The order in which you implement your connector is as follows:

1. Decide on the connector type (see Section 4.1, "Deciding on the Connector Type".
2. Implement the configuration interface (see Section 4.2, "Implementing the Configuration Interface".
3. Implement the connector interface (see Section 4.3, "Implementing the Connector Interface".
4. Implement the operation interfaces (see Section 4.4, "Implementing the Operation Interfaces".

4.1 Deciding on the Connector Type

OpenICF supports multiple connector types, based on the implementation of the `connector` interface, and the `configuration` interface. These two interfaces determine whether the connector can be pooled, and whether its configuration is stateful. Before you begin developing your connector, decide on the *connector type*, based on the system to which you are connecting. For an overview of how the OpenICF framework manages each connector type, see Section 2.8, "How the OpenICF Framework Manages Connector Instances".

This section outlines the different connector types.

Connector

The basic connector is a *non-poolable* connector. Each operation is executed on a new instance of the connector. OpenICF creates a new instance of the Connector class and uses a new or existing instance of the connector configuration to initialise the instance before the operation is executed. After the execution, OpenICF disposes of the connector instance.

Poolable Connector

Before an operation is executed, an existing connector instance is pulled from the Connector Pool. If there is no existing instance, a new instance is created. After the operation execution, the Connector instance is released and placed back into pool.

The OpenICF framework pools *instances* of a poolable connector, rather than pooling connections within the connector.

Configuration

For a basic (non-stateful) configuration, each time the configuration is used (when an operation is validated or a new connector instance is initialised, a new Configuration instance is created and configured with the Configuration properties.

Stateful Configuration

With a stateful configuration, the configuration instance is created only once and is used until the Facade or Connector Pool that is associated with the Configuration is disposed of.

The following table illustrates how these elements combine to determine the connector type.

Table 4.1. Connector Types

	Connector	**Poolable Connector**
Configuration	Entirely stateless combination. A new Configuration and Connector instance	Connector initialisation is an expensive operation, so it is preferable to keep connector instances

	Connector	Poolable Connector
	are created for each operation.	in a pool. A new configuration is required only when a new connector instance is added to the pool.
Stateful Configuration	The configuration can be used to make the heavy resource initialisation. The less intensive connector instance can then execute the operation.	The configuration must be shared between the instances in the same pool and the connector initialisation is expensive.

For detailed information on how the OpenICF framework manages each of these connector types, see Section 2.8, "How the OpenICF Framework Manages Connector Instances".

4.2 Implementing the Configuration Interface

The OpenICF connector framework uses the configuration interface implementation to build the *configuration properties* inside the API configuration.

The configuration interface implementation includes the required information to enable the connector to connect to the target system, and to perform its operations. The configuration interface implements getters and setters for each of its defined properties. It also provides a validate method that your application can use to check whether all the required properties are available, and valid, before passing them to the connector.

The configuration interface has three methods:

- `setConnectorMessages(ConnectorMessages messages)` sets the message catalog instance, that enables the connector to provide localized messages.

  The message catalog is defined in the file `Messages.properties`, and can be localized as required by appending the locale to the file name, for example, `Messages_fr.properties`.

  For more information on the message catalog, see *The Connector Messages Object*.

- `getConnectorMessages()` returns the message catalog that is set by setConnectorMessages(ConnectorMessages)

- `validate()` checks that all the required properties have been set and that their values are valid

  The purpose of this method is to test that the configuration that the application provides to your connector is valid.

Each property that is declared is not necessarily required. If a property is required, it must be included in the `ConfigurationProperty` annotation.

The `ConfigurationProperty` annotation (Java) or attribute (.NET) enables you to add custom meta information to properties. The OpenICF framework scans the meta information and collects this information to build the `ConfigurationProperties` object inside the `APIConfiguration`. The following meta information can be provided:

Element	**Description**	**Implementation in Java**	**Implementation in C#**
order	The order in which this property is displayed		
helpMessageKey	Enables you to change the default help message key	`*propertyName*.help`	`*help_propertyName*`
displayMessageKey	Enables you to change the default display message key	`*propertyName.display*`	`*display_propertyName*`
groupMessageKey	Enables you to change the default group message key	`*propertyName.group*`	`*group_propertyName*`
confidential	Indicates that this is a confidential property and that its value should be encrypted by the application when persisted		

Element	Description	Implementation in Java	Implementation in C#
required	Boolean, indicates whether the property is required		
operations	The array of operations that require this property		

The following examples show how the meta information is provided, in both Java and C#.

Example 4.1. Stateless Configuration Implementation (Java)

```
public class SampleConfiguration extends AbstractConfiguration  {

    /**
     * {@inheritDoc}
     */
    public void validate() {
    }

    @ConfigurationProperty(
        order = 1,
        helpMessageKey = "passwordFieldName.help",
        displayMessageKey = "passwordFieldName.display",
        groupMessageKey = "authenticateOp.group",
        confidential = false,
        required = false,
        operations = {AuthenticateOp.class,CreateOp.class}
    )
    public String getPasswordFieldName() {
        return passwordFieldName;
    }

    public void setPasswordFieldName(String value) {
        passwordFieldName = value;
    }
}
```

Example 4.2. Stateful Configuration Implementation (Java)

```
public class SampleConfiguration extends AbstractConfiguration
    implements StatefulConfiguration {

    /**
     * {@inheritDoc}
     */
    public void release() {
    }

    /**
     * {@inheritDoc}
     */
    public void validate() {
    }
}
```

Example 4.3. Stateless Configuration Implementation (C#)

```
public class ActiveDirectoryConfiguration : AbstractConfiguration
    {

        [ConfigurationProperty(
            Order = 1,
            HelpMessageKey = "help_PasswordFieldName",
            DisplayMessageKey = "display_PasswordFieldName",
            GroupMessageKey = "group_PasswordFieldName",
            Confidential = false,
            Required = false,
            OperationTypes = new[] { typeof(AuthenticateOp) })
        ]
        public String PasswordFieldName
        { get; set; }

        public override void Validate()
        {
            throw new NotImplementedException();
        }
    }
```

Example 4.4. Stateful Configuration Implementation (C#)

```

public class ActiveDirectoryConfiguration : AbstractConfiguration,
    StatefulConfiguration
    {

        public override void Validate()
        {
            throw new NotImplementedException();
        }

        public void Release()
        {
            throw new NotImplementedException();
        }
    }

```

4.2.1 Validate Operation

The validate operation validates the connector configuration. A valid configuration is one that is *ready to be used* by the connector.

A configuration that is *ready*, has the following characteristics:

- It is complete, that is all required properties are present and have values
- All property values are well-formed, that is, they are in the expected range and have the expected format

4.2.1.1 ValidateApiOp

The validate operation returns a `ConfigurationException` in the following situations:

- The Framework version is not compatible with the connector
- The connector does not have the required attributes in `MANIFEST.MF`
- The `ConfigurationProperties` cannot be merged into the configuration

Example 4.5. Implementation of the valid operation, at the API Level

```
@Test
      public void ValidateTest() {
      logger.info("Running Validate Test");
      final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
      facade.validate();
      }
```

4.2.2 Validate SPI Implementation

The `validate()` method of the configuration operation must return one of the following:

- `RuntimeException` if the configuration is not valid
- `NullPointerException` if a required configuration property is null
- `IllegalArgumentException` if a required configuration property is blank

Example 4.6. Implementation of the validate method

```
public void validate() {
    if (StringUtil.isBlank(host)) {
        throw new IllegalArgumentException("Host User cannot be null or empty.");
    }

    Assertions.blankCheck(remoteUser, "remoteUser");

    Assertions.nullCheck(password, "password");
}
```

4.2.3 Supported Configuration Types

The OpenICF framework supports a limited number of configuration property types. This limitation is necessary, because OpenICF must serialise and deserialize the configuration property values when sending them over the network.

You can use any of the following types, or an array of these types. Lists of types are not supported.

```

    String.class
    long.class
    Long.class
    char.class
    Character.class
    double.class
    Double.class
    float.class
    Float.class
    int.class
    Integer.class
    boolean.class
    Boolean.class
    URI.class
    File.class
    GuardedByteArray.class
    GuardedString.class
    Script.class

    typeof(string),
    typeof(long),
    typeof(long?),
    typeof(char),
    typeof(char?),
    typeof(double),
    typeof(double?),
    typeof(float),
    typeof(float?),
    typeof(int),
    typeof(int?),
    typeof(bool),
    typeof(bool?),
    typeof(Uri),
    typeof(FileName),
    typeof(GuardedByteArray),
    typeof(GuardedString),
    typeof(Script)
```

The framework introspects the implemented configuration class and adds all properties that have a `set/get` method to the `ConfigurationProperties` object.

The `ConfigurationClass` annotation (Java) or attribute (.NET) provides additional information to the OpenICF framework about the configuration class. The following information is provided:

Element	Description
privateProperty	If this is set, the property is hidden from the application, and the application cannot set the property through the `APIConfiguration`.
skipUnsupported	If the type of an added property is not supported, the framework throws an exception. To avoid the exception, set the value of `skipUnsupported` to `true`.

Example 4.7. ConfigurationClass Annotation in Java

```
@ConfigurationClass(ignore = { "privateProperty", "internalProperty" }, skipUnsupported = true)
```

Example 4.8. ConfigurationClass Attribute in C#

```
[ConfigurationClass(Ignore = { "privateProperty", "internalProperty" }, SkipUnsupported = true)]
```

4.3 Implementing the Connector Interface

The connector interface declares a connector, and manages its life cycle. You *must* implement the `connector` interface. A typical connector lifecycle is as follows:

- The connector creates a connection to the target system.
- Any operations implemented in the connector are called.
- The connector discards the connection and disposes of any resources it has used.

The `connector` interface has only three methods:

- `init(Configuration)` initializes the connector with its configuration
- `getConfiguration()` returns the configuration that was passed to `init(Configuration)`
- `dispose()` disposes of any resources that the connector uses.

The `ConnectorClass`, which is the implementation of the connector interface, must have the `ConnectorClass` annotation (Java) or attribute (.NET) so that the OpenICF framework can find the connector class. The following table shows the elements within the connector class.

Element	Description		
configurationClass	The configuration class for the connector.		
displayNameKey	A key in the message catalog that holds a human readable name for the connector.		
categoryKey	The category to which the connector belongs, such as LDAP, or DB.		

Element	Description		
messageCatalogPaths	The resource path(s) to the message catalog. If multiple paths are provided, the message catalogs are collated. By default, if no path is specified, the `connector-package.Messages.properties` is used		

The following examples show the connector interface implementation, in Java and C#.

Example 4.9. Connector Interface Implementation in Java

```
@ConnectorClass(
    displayNameKey = "Sample.connector.display",
    configurationClass = SampleConfiguration.class)
public class SampleConnector implements Connector...
```

Example 4.10. Connector Interface Implementation in C#

```
[ConnectorClass(
    "connector_displayName",
    typeof (SampleConfiguration)
    ]
public class SampleConnector : Connector ...
```

4.3.1 Implementing a Poolable Connector Interface

Certain connectors support the ability to be pooled. For a pooled connector, OpenICF maintains a pool of connector instances and reuses these instances for multiple provisioning and reconciliation operations. When an operation must be executed, an existing connector instance is taken from the connector pool. If no connector instance exists, a new instance is initialized. When the operation has been executed, the connector instance is released back into the connector pool, ready to be used for a subsequent operation.

For an unpooled connector, a new connector instance is initialized for every operation. When the operation has been executed, OpenICF disposes of the connector instance. Because the initialization of a connector is an expensive operation, reducing the number of connector initializations can substantially improve performance.

The following connection pooling configuration parameters can be set:

maxObjects
: The maximum number of connector instances in the pool (both idle and active). The default value is 10 instances.

maxIdle
: The maximum number of idle connector instances in the pool. The default value is 10 idle instances.

maxWait
: The maximum period to wait for a free connector instance to become available before failing. The default period is 150000 milliseconds, or 15 seconds.

minEvictableIdleTimeMillis
: The minimum period to wait before evicting an idle connector instance from the pool. The default period is 120000 milliseconds, or 12 seconds.

minIdle
: The minimum number of idle connector instances in the pool. The default value is 1 instance.

A `PoolableConnector` extends the connector interface with the `checkAlive()` method. You should use a `PoolableConnector` when the `init(Configuration)` method is so expensive that it is worth keeping the connector instance in a pool and reusing it between operations. When an existing connector instance is pooled, the framework calls the `checkAlive()` method. If this method throws an error, the framework discards it from the pool and obtains another instance, or creates a new connector instance and calls the `init()` method. The `checkAlive()` method is used to make sure that the instance in the pool is still operational.

4.4 Implementing the Operation Interfaces

The SPI provides several operations. The subset of operations that you implement will depend on the target resource to which you are connecting. Each operation interface defines an action that the connector can perform on the target resource.

The following sections describe the operation interfaces that are provided by the SPI, and provide examples of how they can be implemented in your connector. The sections include the API- and SPI-level rules for each operation.

4.4.1 Authenticate Operation

The authenticate operation authenticates an object on the target system, based on two parameters, usually a unique identifier (username) and a password. If possible, your connector should try to authenticate these credentials natively.

If authentication fails, the connector should throw a runtime exception. The exception must be an `IllegalArgumentException` or, if a native exception is available and is of type `RuntimeException`, that native runtime exception. If the native exception is not a `RuntimeException`, it should be wrapped in a `RuntimeException`, and then thrown.

The exception should provide as much detail as possible for logging problems and failed authentication attempts. Several exceptions are provided in the `exceptions` package, for this purpose. For example, one of the most common authentication exceptions is the `InvalidPasswordException`.

For more information about the common exceptions provided in the OpenICF framework, see Section 4.5, "Common Exceptions".

4.4.1.1 Using the OpenICF Authenticate Operation

This section shows how your application can use the framework's `authentication` operation, and how to write a unit test for this operation, when you are developing your connector.

The `authentication` operation throws a `RuntimeException` if the credentials do not pass authentication, otherwise returns the `UID`.

Example 4.11. Sample Unit Test for the Authentication Operation (Java)

```
@Test
public void authenticateTest() {
    logger.info("Running Authentication Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    Uid uid =
        facade.authenticate(ObjectClass.ACCOUNT, "username", new GuardedString("Passw0rd"
        .toCharArray()), builder.build());
    Assert.assertEquals(uid.getUidValue(), "username");
}
```

4.4.1.2 Implementing the Authenticate Operation in Your Connector

To implement the `authenticate` operation in your connector, add the `AuthenticateOp` interface to your connector class, for example:

```
@ConnectorClass(
    displayNameKey = "Sample.connector.display",
    configurationClass = SampleConfiguration.class)
public class SampleConnector implements Connector, AuthenticateOp...
```

For more information, see the AuthenticateOp JavaDoc.

The SPI provides the following detailed exceptions:

- UnknownUidException - the UID does not exist on the resource

  `(org.identityconnectors.framework.common.exceptions.UnknownUidException)`

- ConnectorSecurityException - base exception for all security-related exceptions

  `(org.identityconnectors.framework.common.exceptions.ConnectorSecurityException)`

- InvalidCredentialException - generic invalid credential exception that should be used if the specific error cannot be obtained

  `(org.identityconnectors.framework.common.exceptions.UnknownUidException)`

- InvalidPasswordException - the password provided is incorrect

  `(org.identityconnectors.framework.common.exceptions.InvalidPasswordException)`

- PasswordExpiredException - the password is correct, but has expired

  `(org.identityconnectors.framework.common.exceptions.PasswordExpiredException)`

- PermissionDeniedException - the user can be identified but does not have permission to authenticate

  `(org.identityconnectors.framework.common.exceptions.PermissionDeniedException)`

Example 4.12. Implementation of the Authentication Operation, at the SPI Level

```
public Uid authenticate(final ObjectClass objectClass, final String userName,
        final GuardedString password, final OperationOptions options) {
    if (ObjectClass.ACCOUNT.equals(objectClass)) {
        return new Uid(userName);
    } else {
        logger.warn("Authenticate of type {0} is not supported", configuration
                .getConnectorMessages().format(objectClass.getDisplayNameKey(),
                        objectClass.getObjectClassValue()));
        throw new UnsupportedOperationException("Authenticate of type"
                + objectClass.getObjectClassValue() + " is not supported");
    }
}
```

4.4.2 Create Operation

The create operation interface enables the connector to create objects on the target system. The operation includes one method (`create()`). The method takes an `ObjectClass`, and any provided attributes, and creates the object and its UID. The connector must return the UID so that the caller can refer to the created object.

The connector should make a best effort to create the object, and should throw an informative `RuntimeException`, indicating to the caller why the operation could not be completed. Defaults can be used for any required attributes, as long as the defaults are documented.

The UID is never passed in with the attribute set for this method. If the resource supports a mutable UID, you can create a resource-specific attribute for the ID, such as `unix_uid`.

If the `create` operation is only partially successful, the connector should attempt to roll back the partial change. If the target system does not allow this, the connector should report the partial success of the create operation and throw a `RetryableException`. For example:

```
public static RetryableException wrap(final String message, final Uid uid) {
    return new RetryableException(message, new AlreadyExistsException().initUid(Assertions
    .nullChecked(uid, "Uid")));
}
```

4.4.2.1 Using the OpenICF Create Operation

The following exceptions are thrown by the Create API operation:

- `IllegalArgumentException` - if `ObjectClass}` is missing, or if elements of the set produce duplicate values of `Attribute#getName()`

- `NullPointerException` - if the `createAttributes` parameter is `null`

- `RuntimeException` - if the `Connector` SPI throws a native exception

Example 4.13. Consumption of the Create Operation, at the API Level

```
@Test
public void createTest() {
    logger.info("Running Create Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    Set<Attribute> createAttributes = new HashSet<Attribute>();
    createAttributes.add(new Name("Foo"));
    createAttributes.add(AttributeBuilder.buildPassword("Password".toCharArray()));
    createAttributes.add(AttributeBuilder.buildEnabled(true));
    Uid uid = facade.create(ObjectClass.ACCOUNT, createAttributes, builder.build());
    Assert.assertEquals(uid.getUidValue(), "foo");
}
```

4.4.2.2 Implementing the Create Operation in Your Connector

The SPI provides the following detailed exceptions:

- `UnsupportedOperationException` - the create operation is not supported for the specified object class

- `InvalidAttributeValueException` - a required attribute is missing, an attribute is present that cannot be created, or a provided attribute has an invalid value

- `AlreadyExistsException` - an object with the specified `Name` already exits on the target system

- `PermissionDeniedException` - the target resource will not allow the connector to perform the specified operation

- `ConnectorIOException, ConnectionBrokenException, ConnectionFailedException` - a problem as occurred with the connection

- `RuntimeException` - thrown if anything else goes wrong. You should try to throw a native exception in this case.

Example 4.14. Implementation of the Create Operation, at the SPI Level

```
public Uid create(final ObjectClass objectClass, final Set<Attribute> createAttributes,
        final OperationOptions options) {
    if (ObjectClass.ACCOUNT.equals(objectClass) || ObjectClass.GROUP.equals(objectClass)) {
        Name name = AttributeUtil.getNameFromAttributes(createAttributes);
        if (name != null) {
            // do real create here
            return new Uid(AttributeUtil.getStringValue(name).toLowerCase());
        } else {
            throw new InvalidAttributeValueException("Name attribute is required");
        }
    } else {
        logger.warn("Delete of type {0} is not supported", configuration.getConnectorMessages()
                .format(objectClass.getDisplayNameKey(), objectClass.getObjectClassValue()));
        throw new UnsupportedOperationException("Delete of type"
                + objectClass.getObjectClassValue() + " is not supported");
    }
}
```

4.4.3 Delete Operation

The delete operation interface enables the connector to delete an object on the target system. The operation includes one method (`delete()`). The method takes an `ObjectClass`, a `Uid`, and any operation options.

The connector should call the native delete methods to remove the object, specified by its unique ID.

4.4.3.1 Using the OpenICF Delete Operation

The following exceptions are thrown by the Delete API operation:

- `UnknownUidException` - the UID does not exist on the resource

Example 4.15. Consumption of the Delete Operation, at the API Level

```
@Test
public void deleteTest() {
    logger.info("Running Delete Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    facade.delete(ObjectClass.ACCOUNT, new Uid("username"), builder.build());
}
```

4.4.3.2 Implementing the Delete Operation in Your Connector

Example 4.16. Implementation of the Delete Operation, at the SPI Level

```
public void delete(final ObjectClass objectClass, final Uid uid, final OperationOptions options) {
    if (ObjectClass.ACCOUNT.equals(objectClass) || ObjectClass.GROUP.equals(objectClass)) {
        // do real delete here
    } else {
        logger.warn("Delete of type {0} is not supported", configuration.getConnectorMessages()
                .format(objectClass.getDisplayNameKey(), objectClass.getObjectClassValue()));
        throw new UnsupportedOperationException("Delete of type"
                + objectClass.getObjectClassValue() + " is not supported");
    }
}
```

4.4.4 Resolve Username Operation

The resolve username operation enables the connector to resolve an object to its UID, based on its username. This operation is similar to the simple authentication operation. However, the resolve username operation does not include a password parameter, and does not attempt to authenticate the credentials. Instead, it returns the UID that corresponds to the supplied username.

The implementation must, however, validate the username (that is, the connector must throw an exception if the username does not correspond to an existing object). If the username validation fails, the the connector should throw a runtime exception, either an `IllegalArgumentException` or, if a native exception is available and is of type `RuntimeException`, simply throw that exception. If the native exception is not a `RuntimeException`, it should be wrapped in a `RuntimeException`, and then thrown.

The exception should provide as much detail as possible for logging problems and failed attempts. Several exceptions are provided in the `exceptions` package, for this purpose. For example, one of the most common exceptions is the `UnknownUidException`.

4.4.4.1 Using the OpenICF Resolve Username Operation

The operation throws a `RuntimeException` if the username validation fails, otherwise returns the `UID`.

Example 4.17. Consumption of the ResolveUsername operation, at the API Level

```
@Test
public void resolveUsernameTest() {
    logger.info("Running ResolveUsername Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    Uid uid = facade.resolveUsername(ObjectClass.ACCOUNT, "username", builder.build());
    Assert.assertEquals(uid.getUidValue(), "username");
}
```

4.4.4.2 Implementing the Resolve Username Operation in Your Connector

The SPI provides the following detailed exceptions:

- UnknownUidException - the UID does not exist on the resource

Example 4.18. Implementation of the ResolveUsername Operation, at the SPI Level

```
public Uid resolveUsername(final ObjectClass objectClass, final String userName,
        final OperationOptions options) {
    if (ObjectClass.ACCOUNT.equals(objectClass)) {
        return new Uid(userName);
    } else {
        logger.warn("ResolveUsername of type {0} is not supported", configuration
                .getConnectorMessages().format(objectClass.getDisplayNameKey(),
                        objectClass.getObjectClassValue()));
        throw new UnsupportedOperationException("ResolveUsername of type"
                + objectClass.getObjectClassValue() + " is not supported");
    }
}
```

4.4.5 Schema Operation

The Schema Operation interface enables the connector to describe the types of objects that it can handle on the target system, and the operations and options that the connector supports for each object type.

The operation has one method, `schema()`, which returns the types of objects on the target system that the connector supports. The method should return the object class name, its description, and a set of attribute definitions.

The implementation of this operation includes a mapping between the native object class and the corresponding connector object. The special `Uid` attribute should not be returned, because it is not a true attribute of the object, but a reference to it. For more information about special attributes in OpenICF, see *OpenICF Special Attributes*.

If your resource object class has a writable unique ID attribute that is different to its `Name`, your schema should contain a resource-specific attribute that represents this unique ID. For example, a Unix account object might contain a `unix_uid`.

4.4.5.1 Using the OpenICF Schema Operation

Example 4.19. Consumption of the Schema Operation, at the API Level

```
@Test
public void schemaTest() {
    logger.info("Running Schema Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    Schema schema = facade.schema();
    Assert.assertNotNull(schema.findObjectClassInfo(ObjectClass.ACCOUNT_NAME));
}
```

4.4.5.2 Implementing the Schema Operation in Your Connector

Example 4.20. Implementation of the SchemaOp operation, at the SPI Level

```
public Schema schema() {
    if (null == schema) {
        final SchemaBuilder builder = new SchemaBuilder(BasicConnector.class);
        // Account
        ObjectClassInfoBuilder accountInfoBuilder = new ObjectClassInfoBuilder();
        accountInfoBuilder.addAttributeInfo(Name.INFO);
        accountInfoBuilder.addAttributeInfo(OperationalAttributeInfos.PASSWORD);
        accountInfoBuilder.addAttributeInfo(PredefinedAttributeInfos.GROUPS);
        accountInfoBuilder.addAttributeInfo(AttributeInfoBuilder.build("firstName"));
        accountInfoBuilder.addAttributeInfo(AttributeInfoBuilder.define("lastName")
                .setRequired(true).build());
        builder.defineObjectClass(accountInfoBuilder.build());

        // Group
        ObjectClassInfoBuilder groupInfoBuilder = new ObjectClassInfoBuilder();
        groupInfoBuilder.setType(ObjectClass.GROUP_NAME);
        groupInfoBuilder.addAttributeInfo(Name.INFO);
        groupInfoBuilder.addAttributeInfo(PredefinedAttributeInfos.DESCRIPTION);
        groupInfoBuilder.addAttributeInfo(AttributeInfoBuilder.define("members").setCreatable(
                false).setUpdateable(false).setMultiValued(true).build());

        // Only the CRUD operations
        builder.defineObjectClass(groupInfoBuilder.build(), CreateOp.class, SearchOp.class,
                UpdateOp.class, DeleteOp.class);

        // Operation Options
        builder.defineOperationOption(OperationOptionInfoBuilder.buildAttributesToGet(),
                SearchOp.class);

        // Support paged Search
        builder.defineOperationOption(OperationOptionInfoBuilder.buildPageSize(),
                SearchOp.class);
        builder.defineOperationOption(OperationOptionInfoBuilder.buildPagedResultsCookie(),
                SearchOp.class);
        // Support to execute operation with provided credentials
        builder.defineOperationOption(OperationOptionInfoBuilder.buildRunWithUser());
        builder.defineOperationOption(OperationOptionInfoBuilder.buildRunWithPassword());

        schema = builder.build();
    }
    return schema;
}
```

4.4.6 Script On Connector Operation

The script on connector operation runs a script in the environment of the connector. This is different to the script on resource operation, which runs a script on the target resource that the connector manages.

The corresponding API operation (`scriptOnConnectorApiOp`) provides a minimum contract to which the connector must adhere. (See the javadoc for more information). If you do not implement the `scriptOnConnector` interface in your connector, the framework provides a default implementation. If you intend your connector to provide more to the script than what is required by this minimum contract, you must implement the `scriptOnConnectorOp` interface.

4.4.6.1 Using the OpenICF Script on Connector Operation

The API operation allows an application to run a script in the context of any connector.

This operation runs the script in the same JVM or .Net Runtime as the connector. That is, if you are using a local framework, the script runs in your JVM. If you are connected to a remote framework, the script runs in the remote JVM or .Net Runtime.

Example 4.21. Consumption of the ScriptOnConnector operation, at the API Level

```
@Test
public void runScriptOnConnectorTest() {
    logger.info("Running RunScriptOnConnector Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    builder.setRunAsUser("admin");
    builder.setRunWithPassword(new GuardedString("Passw0rd".toCharArray()));

    final ScriptContextBuilder scriptBuilder =
            new ScriptContextBuilder("Groovy", "return argument");
    scriptBuilder.addScriptArgument("argument", "value");

    Object result = facade.runScriptOnConnector(scriptBuilder.build(), builder.build());
    Assert.assertEquals(result, "value");
}
```

4.4.6.2 Implementing the Script on Connector Operation in Your Connector

The `scriptOnConnector` SPI operation takes the following parameters:

- `request` - the script and the arguments to be run
- `options` - additional options that control how the script is run

The operation returns the result of the script. The return type must be a type that the framework supports for serialization. See the ObjectSerializerFactory javadoc for a list of supported return types.

Example 4.22. Implementation of the ScriptOnConnector operation, at the SPI Level

```
public Object runScriptOnConnector(ScriptContext request, OperationOptions options) {
    final ScriptExecutorFactory factory =
            ScriptExecutorFactory.newInstance(request.getScriptLanguage());
    final ScriptExecutor executor =
            factory.newScriptExecutor(getClass().getClassLoader(), request.getScriptText(),
                    true);

    if (StringUtil.isNotBlank(options.getRunAsUser())) {
        String password = SecurityUtil.decrypt(options.getRunWithPassword());
        // Use these to execute the script with these credentials
    }
    try {
        return executor.execute(request.getScriptArguments());
    } catch (Throwable e) {
        logger.warn(e, "Failed to execute Script");
        throw ConnectorException.wrap(e);
    }
}
```

4.4.7 Script On Resource Operation

The script on resource operation runs a script directly on the target resource (unlike the Section 4.4.6, "Script On Connector Operation", which runs a script in the context of a specific connector.)

Implement this interface if your connector intends to support the `ScriptOnResourceApiOp` API operation. If your connector implements this interface, you must document the script languages that the connector supports, as well as any supported `OperationOptions`.

4.4.7.1 Using the OpenICF Script on Resource Operation

The contract at the API level is intentionally very loose. Each connector decides what script languages it supports, what running a script on a target resource actually means, and what script options (if any) the connector supports.

Example 4.23. Consumption of the ScriptOnResource operation, at the API Level

```
@Test
public void runScriptOnResourceTest() {
    logger.info("Running RunScriptOnResource Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    builder.setRunAsUser("admin");
    builder.setRunWithPassword(new GuardedString("Passw0rd".toCharArray()));

    final ScriptContextBuilder scriptBuilder = new ScriptContextBuilder("bash", "whoami");

    Object result = facade.runScriptOnResource(scriptBuilder.build(), builder.build());
    Assert.assertEquals(result, "admin");
}
```

4.4.7.2 Implementing the Script on Resource Operation in Your Connector

The `scriptOnResource` SPI operation takes the following parameters:

- `request` - the script and the arguments to be run
- `options` - additional options that control how the script is run

The operation returns the result of the script. The return type must be a type that the framework supports for serialization. See the ObjectSerializerFactory javadoc for a list of supported return types.

Example 4.24. Implementation of the ScriptOnResource operation, at the SPI Level

```
public Object runScriptOnResource(ScriptContext request, OperationOptions options) {
    try {
        // Execute the script on remote resource
        if (StringUtil.isNotBlank(options.getRunAsUser())) {
            String password = SecurityUtil.decrypt(options.getRunWithPassword());
            // Use these to execute the script with these credentials
            return options.getRunAsUser();
        }
        throw new UnknownHostException("Failed to connect to remote SSH");
    } catch (Throwable e) {
        logger.warn(e, "Failed to execute Script");
        throw ConnectorException.wrap(e);
    }
}
```

4.4.8 Search Operation

The search operation enables the connector to search for objects on the target system.

The OpenICF framework handles searches as follows:

1. The application sends a query, with a search filter, to the OpenICF framework
2. The framework submits the query, with the filter, to the connector
3. The connector implements the `createFilterTranslator()` method to obtain a `FilterTranslator` object
4. The framework then uses this `FilterTranslator` object to transform the filter to a format that the `executeQuery()` method expects

You can implement the `FilterTranslator` object in two ways:

- The `FilterTranslator` translates the original filter into one or more native queries.

  The framework then calls the `executeQuery()` method for each native query.

- The `FilterTranslator` does not modify the original filter.

  The framework then calls the `executeQuery()` method with the original OpenICF filter.

  Using this second approach enables your connector to distinguish between a search and a get operation and to benefit from the visitor design pattern.

Based on the `resultsHandlerConfiguration`, the OpenICF framework can perform additional filtering on the returning results. For more information on the `resultsHandlerConfiguration`, see *Results Handler Configuration*.

The connector facade calls the `executeQuery` method once for each native query that the filter translator produces. If the filter translator produces more than one native query, the connector facade merges the results from each query and eliminates any duplicates.

Note that this implies an in-memory data structure that holds a set of UID values. Memory usage, in the event of multiple queries, will be O(N) where N is the number of results. It is therefore important that the filter translator for the connector implement `OR` operators, if possible.

Whether the application calls a `get` API operation, or a `search` API operation, the OpenICF framework translates that request to a `search` request on the connector.

4.4.8.1 Using the OpenICF Get Operation

The `GetApiOp` returns `null` when the UID does not exist on the resource.

Example 4.25. Consumption of the Get operation, at the API Level

```
@Test
public void getObjectTest() {
    logger.info("Running GetObject Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    builder.setAttributesToGet(Name.NAME);
    ConnectorObject co =
            facade.getObject(ObjectClass.ACCOUNT, new Uid(
                    "3f50eca0-f5e9-11e3-a3ac-0800200c9a66"), builder.build());
    Assert.assertEquals(co.getName().getNameValue(), "Foo");
}
```

4.4.8.2 Using the OpenICF Search Operation

Example 4.26. Consumption of the Search operation, at the API Level

```
@Test
public void searchTest() {
    logger.info("Running Search Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    builder.setPageSize(10);
    final ResultsHandler handler = new ToListResultsHandler();

    SearchResult result =
            facade.search(ObjectClass.ACCOUNT, FilterBuilder.equalTo(new Name("Foo")), handler,
                    builder.build());
    Assert.assertEquals(result.getPagedResultsCookie(), "0");
    Assert.assertEquals(((ToListResultsHandler) handler).getObjects().size(), 1);
}
```

4.4.8.3 Implementing the Search Operation in Your Connector

Example 4.27. Implementation of the Search operation, at the SPI Level

```
public FilterTranslator<String> createFilterTranslator(ObjectClass objectClass,
        OperationOptions options) {
    return new BasicFilterTranslator();
}

public void executeQuery(ObjectClass objectClass, String query, ResultsHandler handler,
        OperationOptions options) {
    final ConnectorObjectBuilder builder = new ConnectorObjectBuilder();
    builder.setUid("3f50eca0-f5e9-11e3-a3ac-0800200c9a66");
    builder.setName("Foo");
    builder.addAttribute(AttributeBuilder.buildEnabled(true));

    for (ConnectorObject connectorObject : CollectionUtil.newSet(builder.build())) {
        if (!handler.handle(connectorObject)) {
            // Stop iterating because the handler stopped processing
            break;
        }
    }
    if (options.getPageSize() != null && 0 < options.getPageSize()) {
        logger.info("Paged Search was requested");
        ((SearchResultsHandler) handler).handleResult(new SearchResult("0", 0));
    }
}
```

4.4.9 Sync Operation

The sync operation polls the target system for synchronization events, that is, native changes to target objects.

The operation has two methods:

- `sync()` - request synchronization events from the target system

  This method calls the specified handler, once, to pass back each matching synchronization event. When the method returns, it will no longer invoke the specified handler.

- `getLatestSyncToken()` - returns the token corresponding to the most recent synchronization event

4.4.9.1 Using the OpenICF Sync Operation

Example 4.28. Consumption of the Sync Operation (`getLatestSyncToken()` Method)at the API Level

```
@Test
public void getLatestSyncTokenTest() {
    logger.info("Running GetLatestSyncToken Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    SyncToken token = facade.getLatestSyncToken(ObjectClass.ACCOUNT);
    Assert.assertEquals(token.getValue(), 10);
}
```

The `getLatestSyncToken` method throws an `IllegalArgumentException` if the `objectClass` is null or invalid.

Example 4.29. Consumption of the Sync Operation (`sync()` Method), at the API Level

```
@Test
public void syncTest() {
    logger.info("Running Sync Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    builder.setPageSize(10);
    final SyncResultsHandler handler = new SyncResultsHandler() {
        public boolean handle(SyncDelta delta) {
            return false;
        }
    };

    SyncToken token =
            facade.sync(ObjectClass.ACCOUNT, new SyncToken(10), handler, builder.build());
    Assert.assertEquals(token.getValue(), 10);
}
```

The `sync` method throws an `IllegalArgumentException` if the `objectClass` or `handler` is null, or if any argument is invalid.

4.4.9.2 Implementing the Sync Operation in Your Connector

Example 4.30. Implementation of the Sync Operation at the SPI Level

```
public void sync(ObjectClass objectClass, SyncToken token, SyncResultsHandler handler,
        final OperationOptions options) {
    if (ObjectClass.ALL.equals(objectClass)) {
        //
    } else if (ObjectClass.ACCOUNT.equals(objectClass)) {
        final ConnectorObjectBuilder builder = new ConnectorObjectBuilder();
        builder.setUid("3f50eca0-f5e9-11e3-a3ac-0800200c9a66");
        builder.setName("Foo");
        builder.addAttribute(AttributeBuilder.buildEnabled(true));

        final SyncDeltaBuilder deltaBuilder = new SyncDeltaBuilder();
        deltaBuilder.setObject(builder.build());
        deltaBuilder.setDeltaType(SyncDeltaType.CREATE);
        deltaBuilder.setToken(new SyncToken(10));

        for (SyncDelta connectorObject : CollectionUtil.newSet(deltaBuilder.build())) {
            if (!handler.handle(connectorObject)) {
                // Stop iterating because the handler stopped processing
                break;
            }
        }
    } else {
        logger.warn("Sync of type {0} is not supported", configuration.getConnectorMessages()
                .format(objectClass.getDisplayNameKey(), objectClass.getObjectClassValue()));
        throw new UnsupportedOperationException("Sync of type"
                + objectClass.getObjectClassValue() + " is not supported");
    }
    ((SyncTokenResultsHandler) handler).handleResult(new SyncToken(10));
}

public SyncToken getLatestSyncToken(ObjectClass objectClass) {
    if (ObjectClass.ACCOUNT.equals(objectClass)) {
        return new SyncToken(10);
    } else {
        logger.warn("Sync of type {0} is not supported", configuration.getConnectorMessages()
                .format(objectClass.getDisplayNameKey(), objectClass.getObjectClassValue()));
        throw new UnsupportedOperationException("Sync of type"
                + objectClass.getObjectClassValue() + " is not supported");
    }
}
```

4.4.10 Test Operation

The test operation tests the connector configuration. Unlike validation, testing a configuration verifies that every part of the environment that is referred to by the configuration is available. The operation therefore validates that the connection details that are provided in the configuration are accurate, and that the backend is accessible when using them.

For example, the connector might make a physical connection to the host that is specified in the configuration, to check that it exists and that the credentials supplied in the configuration are valid.

The test operation can be invoked before the configuration has been validated, or can validate the configuration before testing it.

4.4.10.1 Using the OpenICF Test Operation

At the API level, the test operation throws a `RuntimeException` if the configuration is not valid, or if the test fails. Your connector implementation should throw the most specific exception available. When no specific exception is available, your connector implementation should throw a `ConnectorException`.

Example 4.31. Consumption of the Test Operation at the API Level

```
@Test
public void testTest() {
    logger.info("Running Test Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    facade.test();
}
```

4.4.10.2 Implementing the Test Operation in Your Connector

Example 4.32. Implementation of the Test Operation at the SPI Level

```
public void test() {
    logger.ok("Test works well");
}
```

4.4.11 Update Operation

If your connector will allow an authorized caller to update (modify or replace) objects on the target system, you must implement either the update operation, or the Section 4.4.12, "Update Attribute Values Operation". At the API level update operation calls either the `UpdateOp` or the `UpdateAttributeValuesOp`, depending on what you have implemented.

The update operation is somewhat simpler to implement than the Section 4.4.12, "Update Attribute Values Operation", because the update attribute values operation must handle any type of update that the caller might specify. However a true implementation of the update attribute values operation offers better performance and atomicity semantics.

4.4.11.1 Using the OpenICF Update Operation

At the API level, the update operation returns an `UnknownUidException` if the UID does not exist on the target system resource and if the connector does not implement the Section 4.4.12, "Update Attribute Values Operation" interface.

Example 4.33. Consumption of the Update Operation at the API Level

```
@Test
public void updateTest() {
    logger.info("Running Update Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    Set<Attribute> updateAttributes = new HashSet<Attribute>();
    updateAttributes.add(new Name("Foo"));

    Uid uid = facade.update(ObjectClass.ACCOUNT, new Uid("Foo"), updateAttributes, builder
                  .build());
    Assert.assertEquals(uid.getUidValue(), "foo");
}
```

4.4.11.2 Implementing the Update Operation in Your Connector

At the SPI level, the update operation returns an `UnknownUidException` if the UID does not exist on the target system.

Example 4.34. Implementation of the Update Operation at the SPI Level

```
public Uid update(ObjectClass objectClass, Uid uid, Set<Attribute> replaceAttributes,
        OperationOptions options) {
    AttributesAccessor attributesAccessor = new AttributesAccessor(replaceAttributes);
    Name newName = attributesAccessor.getName();
    Uid uidAfterUpdate = uid;
    if (newName != null) {
        logger.info("Rename the object {0}:{1} to {2}", objectClass.getObjectClassValue(), uid
                .getUidValue(), newName.getNameValue());
        uidAfterUpdate = new Uid(newName.getNameValue().toLowerCase());
    }

    if (ObjectClass.ACCOUNT.equals(objectClass)) {

    } else if (ObjectClass.GROUP.is(objectClass.getObjectClassValue())) {
        if (attributesAccessor.hasAttribute("members")) {
            throw new InvalidAttributeValueException(
                    "Requested to update a read only attribute");
        }
    } else {
        logger.warn("Update of type {0} is not supported", configuration.getConnectorMessages()
                .format(objectClass.getDisplayNameKey(), objectClass.getObjectClassValue()));
        throw new UnsupportedOperationException("Update of type"
                + objectClass.getObjectClassValue() + " is not supported");
    }
    return uidAfterUpdate;
}
```

4.4.11.2.1 Suggested Approach for Deleting Attributes and Removing Attribute Values

If the target resource to which you are connecting supports the removal of attributes, you can implement the removal in several ways. All the samples in this document assume the following syntax rules for deleting attributes or removing their values.

Update	Syntax rule	Query filter
Set an empty attribute value	`[""]` (application sends an attribute value that is a list containing one empty string)	`equal=""`
Set an attribute value to null	`[]` (application sends an attribute value that is an empty list)	`ispresent` search returns 1
Removing an attribute	`null` (application sends an attribute value that is `null`	`ispresent` search returns 1

4.4.12 Update Attribute Values Operation

The update attribute values operation is an advanced implementation of the update operation. You should implement this operation if you want your connector to offer better performance and atomicity for the following methods:

- `UpdateApiOp.addAttributeValues(ObjectClass, Uid, Set, OperationOptions)`
- `UpdateApiOp.removeAttributeValues(ObjectClass, Uid, Set, OperationOptions)`

Example 4.35. Consumption of the Add and Remove Attribute Values Methods at the API Level

```
@Test
public void addAttributeValuesTest() {
    logger.info("Running AddAttributeValues Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    Set<Attribute> updateAttributes = new HashSet<Attribute>();
    // add 'group2' to existing groups
    updateAttributes.add(AttributeBuilder.build(PredefinedAttributes.GROUPS_NAME, "group2"));

    Uid uid =
            facade.addAttributeValues(ObjectClass.ACCOUNT, new Uid("Foo"), updateAttributes,
                    builder.build());
    Assert.assertEquals(uid.getUidValue(), "foo");
}

@Test
public void removeAttributeValuesTest() {
    logger.info("Running RemoveAttributeValues Test");
    final ConnectorFacade facade = createConnectorFacade(BasicConnector.class, null);
    final OperationOptionsBuilder builder = new OperationOptionsBuilder();
    Set<Attribute> updateAttributes = new HashSet<Attribute>();
    // remove 'group2' from existing groups
    updateAttributes.add(AttributeBuilder.build(PredefinedAttributes.GROUPS_NAME, "group2"));

    Uid uid =
            facade.removeAttributeValues(ObjectClass.ACCOUNT, new Uid("Foo"), updateAttributes,
                    builder.build());
    Assert.assertEquals(uid.getUidValue(), "foo");
}
```

4.4.12.1 Implementing the Update Attribute Values Operation in Your Connector

At the SPI level, the update attribute values operation returns an `UnknownUidException` when the UID does not exist on the resource.

Example 4.36. Implementation of the update attribute values operation, at the SPI Level

```
public Uid addAttributeValues(ObjectClass objectClass, Uid uid, Set<Attribute> valuesToAdd,
        OperationOptions options) {
    return uid;
}

public Uid removeAttributeValues(ObjectClass objectClass, Uid uid,
        Set<Attribute> valuesToRemove, OperationOptions options) {
    return uid;
}
```

4.5 Common Exceptions

The following sections describe the commonly used exceptions that can be thrown, depending on the operation.

4.5.1 AlreadyExistsException

The `AlreadyExistsException` is thrown if a create operation attempts to create an object that exists prior to the method execution, or if an update operation attempts to rename an object to that exists prior to the method execution.

4.5.2 ConfigurationException

A `ConfigurationException` is thrown if a configuration problem is encountered when the connector bundles are loaded. A `ConfigurationException` can also be thrown during validation operations in the SPI.

4.5.3 ConnectionBrokenException

A `ConnectionBrokenException` is thrown when a connection to a target resource instance fails during an operation. An instance of the `ConnectionBrokenException` generally wraps the native exception (or describes the native error) that is returned by the target resource.

4.5.4 ConnectionFailedException

A `ConnectionFailedException` is thrown when a connector cannot reach the target resource. An instance of the `ConnectionFailedException` generally wraps the native exception (or describes the native error) that is returned by the target resource.

4.5.5 ConnectorException

This is the base exception for the connector framework. The framework only throws exceptions that extend `ConnectorException`.

4.5.6 ConnectorIOException

This is the base exception for all Input-Output (I/O-related) exceptions, including instance connection failure, socket error and so forth.

4.5.7 ConnectorSecurityException

This is the base exception for all security-related exceptions.

4.5.8 InvalidAttributeValueException

An `InvalidAttributeValueException` is thrown when an attempt is made to add to an attribute a value that conflicts with the attribute's schema definition. This might happen, for example, in the following situations:

- The connector attempts to add an attribute with no value when the attribute is required to have at least one value
- The connector attempts to add more than one value to a single valued-attribute
- The connector attempts to add a value that conflicts with the attribute type
- The connector attempts to add a value that conflicts with the attribute syntax

4.5.9 InvalidCredentialException

An `InvalidCredentialException` indicates that user authentication has failed. This exception is thrown by the connector when authentication fails, and when the specific reason for the failure is not known. For example, the connector might throw this exception if a user has entered an incorrect password, or username.

4.5.10 InvalidPasswordException

An `InvalidPasswordException` is thrown when a password credential is invalid.

4.5.11 OperationTimeoutException

An `OperationTimeoutException` is thrown when an operation times out. The framework cancels an operation when the corresponding method has been executing for longer than the limit specified in `APIConfiguration`.

4.5.12 PasswordExpiredException

A `PasswordExpiredException` indicates that a user password has expired. This exception is thrown by the connector when it can determine that a password has expired. For example, after successfully authenticating a user, the connector might determine that the user's password has expired. The connector throws this exception to notify the application, which can then take the appropriate steps to notify the user.

4.5.13 PermissionDeniedException

A `PermissionDeniedException` is thrown when the target resource will not allow a connector to perform a particular operation. An instance of the `PermissionDeniedException` generally describes a native error (or wraps a native exception) that is returned by the target resource.

4.5.14 PreconditionFailedException

A `PreconditionFailedException` is thrown to indicate that a resource's current version does not match the version provided. This exception is equivalent to the HTTP status: `412 Precondition Failed`.

4.5.15 PreconditionRequiredException

A `PreconditionRequiredException` is thrown to indicate that a resource requires a version, but that no version was supplied in the request. This exception is equivalent to the HTTP status: `428 Precondition Required`.

4.5.16 RetryableException

A `RetryableException` indicates that the failure might be temporary, and that retrying the same request might succeed in the future.

4.5.17 UnknownUidException

An `UnknownUidException` is thrown when a UID that is specified as input to a connector operation identifies no object on the target resource. When you implement the `AuthenticateOp`, your connector can throw this exception if it is unable to locate the account necessary to perform authentication.

4.5.18 NullPointerException (c# NullReferenceException)

Generic native exception

4.5.19 UnsupportedOperationException (c# NotSupportedException)

Generic native exception

4.5.20 IllegalStateException (c# InvalidOperationException)

Generic native exception

4.5.21 IllegalArgumentException (c# ArgumentException)

Generic native exception

4.5.22 Mapping OpenICF Exceptions to ForgeRock CREST Exceptions

The following table maps the errors that are thrown by the OpenICF framework to the errors that are returned by the ForgeRock Commons REST (CREST) implementation.

OpenICF Exception	CREST Exception	HTTP Error Code
AlreadyExistsException	ConflictException	
ConfigurationException	InternalServerErrorException	
ConnectionBrokenException	InternalServerErrorException	
ConnectionFailedException	ConnectionFailedException	
ConnectorException	InternalServerErrorException	
ConnectorIOException	InternalServerErrorException	
ConnectorSecurityException	ForbiddenException	
InvalidAttributeValueException	BadRequestException	
InvalidCredentialException	ForbiddenException	
InvalidPasswordException	ForbiddenException	
OperationTimeoutException		
PasswordExpiredException	ForbiddenException	
PermissionDeniedException	ForbiddenException	
PreconditionFailedException	PreconditionFailedException	
PreconditionRequiredException	PreconditionRequiredException	
RetryableException	RetryableException (ServiceUnavailableException)	
UnknownUidException	NotFoundException	
UnsupportedOperationException	NotSupportedException	

OpenICF Exception	CREST Exception	HTTP Error Code
IllegalArgumentException	InternalServerErrorException	
NullPointerException	InternalServerErrorException	

4.6 Generic Exception Rules

The generic exception rules are common to all API or SPI level operations and are described in the following sections.

4.6.1 Framework (API Level) Exception Rules

IllegalArgumentException or NullPointerException
Thrown when the ObjectClass is null or the name is blank.

OperationTimeoutException
Thrown when the operation timed out.

ConnectionFailedException
Thrown if any problem occurs with the connector server connection.

UnsupportedOperationException
Thrown if the connector does not implement the required interface.

ConnectorIOException
Thrown if the connector failed to initialize a remote connection due to a SocketException.

ConnectorException
Thrown in the following situations:

- The connector failed to initiate the remote connection due to a `SocketException`
- An unexpected request was sent to the remote connector server
- An unexpected response was received from the remote connector server

InvalidCredentialException
Thrown if the remote framework key is invalid

The following exceptions are thrown specifically in the context of a poolable connector.

ConnectorException
Thrown if the pool has no available connectors after the `maxWait` time has elapsed.

IllegalStateException
Thrown if the object pool has already shut down.

4.6.2 Connector (SPI Level) Exception Rules

InvalidAttributeValueException
Thrown when single-valued attribute has multiple values.

IllegalArgumentException
Thrown when the value of the `__PASSWORD__` or the `__CURRENT_PASSWORD__` attribute is not a `GuardedString`.

IllegalStateException
Thrown when the `Attribute` name is blank.

PermissionDeniedException
Thrown when the target resource will not allow a specific operation to be performed. An instance of the `PermissionDeniedException` generally describes a native error that is returned by (or wraps a native exception that is thrown by) the target resource.

ConnectorIOException, ConnectionBrokenException, ConnectionFailedException
Thrown when any problem occurs with the connection to the target resource.

PreconditionFailedException
Thrown when the current version of the resource object does not match the version provided by the connector.

PreconditionRequiredException
Thrown when a resource object requires a version, but no version was supplied in the `getRevision` operation.

Chapter 5

Writing Java Connectors

If none of the existing OpenICF connectors are suitable for your deployment, you can write your own connector bundle. This chapter describes the steps to develop an OpenICF-compatible Java connector.

Similar chapters exist to help you with writing scripted Groovy, and PowerShell connectors.

5.1 Before You Begin

Before you start developing your own connector, familiarize yourself with the structure of the SPI, by reading Chapter 4, "Implementing the OpenICF SPI" and the corresponding Javadoc for the OpenICF framework and its supported operations. You can also download and study the source code for the existing connectors from the ForgeRock Projects page.

5.2 Using the Connector Archetype

OpenICF provides a Maven connector archetype that enables you to get started with connector development.

Note

A separate archetype is available for building connectors with the Groovy Connector Toolkit. For more information, see *Writing Connectors With the Groovy Connector Toolkit*.

To get started with the Connector Archetype, execute the following command, customizing these options to describe your new connector:

- `-DartifactId=sample-connector`
- `-Dversion=0.0-SNAPSHOT`
- `-Dpackage=org.forgerock.openicf.connectors.sample`
- `-DconnectorName=Sample`

This command imports the archetype and generates a new connector project.

```
$ mvn archetype:generate \
     -DarchetypeGroupId=org.forgerock.openicf \
     -DarchetypeArtifactId=connector-archetype \
     -DarchetypeVersion=1.0.0-SNAPSHOT \
     -DremoteRepositories=http://maven.forgerock.org/repo/snapshots \
     -DarchetypeRepository=http://maven.forgerock.org/repo/snapshots \
     -DgroupId=org.forgerock.openicf.connectors \
     -DartifactId=sample-connector \
     -Dversion=0.0-SNAPSHOT \
     -Dpackage=org.forgerock.openicf.connectors.sample \
     -DconnectorName=Sample
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.3:generate (default-cli) @ standalone-pom >>>
...
ALL_OPERATIONS: n
OP_AUTHENTICATE: n
OP_CREATE: y
OP_DELETE: y
OP_RESOLVEUSERNAME: n
OP_SCHEMA: n
OP_SCRIPTONCONNECTOR: n
OP_SCRIPTONRESOURCE: n
OP_SEARCH: y
OP_SYNC: n
OP_TEST: y
OP_UPDATE: y
OP_UPDATEATTRIBUTEVALUES: n
attributeNormalizer: n
compatibility_version: 1.1
```

```
connectorName: Sample
framework_version: 1.0
jira_componentId: 10191
jira_fixVersionIds: 0
poolableConnector: n
 Y: :
```

At this point, you can enter Y (YES) to accept the default project, or N (NO) to customize the project for your connector. As you can see from the preceding output, the default connector supports only the `create`, `delete`, `search`, `test`, and `update` operations, and is not a poolable connector. To add support for additional operations, or to change any of the connector parameters, enter N (NO). The archetype then prompts you to set values for each parameter.

After you have imported the archetype once, you can use the local version of the archetype, as follows:

```
$ mvn archetype:generate -DarchetypeCatalog=local
```

5.3 Implementing OpenICF Operations

When you have generated the archetype, implement the OpenICF operations that your connector will support.

For information about implementing operations, and examples for a Java connector, see Chapter 4, "Implementing the OpenICF SPI".

Then build the connector, as follows:

5.4 Bundling the Java Connector

```

$ cd sample-connector
$ mvn install
```

5.5 Next Steps

If you are thinking of contributing your connector to the OpenICF project, please review ForgeRock's Development Guidelines.

You are welcome to join the community and ask questions on the mailing list, to report bugs, and submit patches when you get involved.

Chapter 6

Writing Scripted Connectors With the Groovy Connector Toolkit

The Groovy Connector Toolkit enables you to run Groovy scripts to interact with any external resource.

The Groovy Connector Toolkit is not a complete connector, in the traditional sense. Rather, it is a framework within which you must write your own Groovy scripts to address the requirements of your deployment. OpenICF and the OpenIDM project provide specific utilities to help you get started with the Groovy Connector Toolkit.

The Groovy Connector Toolkit is bundled with OpenIDM 4.0.0, and can also be downloaded from the Forgerock Maven repository.

A number of deployment-specific scripts are provided in the samples that are delivered with OpenIDM. These scripts demonstrate how the Groovy Connector Toolkit can be used. The scripts cannot be used "as is" in your deployment, but can be used as a starting point on which to base your customization. OpenIDM also provides a custom connector bundler, that does much of the development work for you. For more information, see *Building Connectors With the Custom Scripted Connector Bundler*.

The Groovy Connector Toolkit can be used with any OpenICF-enabled project (that is, any project in which the OpenICF Connector Framework is installed).

6.1 About the Groovy Scripting Language

Groovy is a powerful, convenient scripting language for the Java platform. Groovy enables you to take advantage of existing Java resources, and generally makes development quicker. Syntactically, Groovy is similar to JavaScript. Extensive information about Groovy is available on the Groovy documentation site.

6.2 Selecting a Scripted Connector Implementation

The Groovy Connector Toolkit provides five default connector implementations. These default implementations should address the requirements of most target resources. If you use one of the default implementations, you need only write the accompanying scripts, and point your connector to their location. If your target resource is not covered by the default implementations, you can use the Maven archetype to create a new connector project, and write a custom Groovy-based connector from scratch.

The following list describes the default scripted connector implementations provided with the Groovy Connector Toolkit:

- `GROOVY` - a basic non-pooled Groovy connector, provided in the `org.forgerock.openicf.connectors.groovy.ScriptedConnector` class.

  `POOLABLEGROOVY` - a poolable Groovy connector, provided in the `org.forgerock.openicf.connectors.groovy.ScriptedPoolableConnector` class.

  `CREST` - a connector based on the ForgeRock Commons REST (CREST) API, and provided in the `org.forgerock.openicf.connectors.groovy.ScriptedCRESTConnector` class. The scripted CREST connector takes a schema configuration file to define the attribute mapping from the OpenICF connector object to the CREST resource.

  `REST` - a scripted REST connector, provided in the `org.forgerock.openicf.connectors.groovy.ScriptedRESTConnector` class. The scripted REST connector enables you to connect to any resource, over HTTP/REST. The connector creates the HTTP/REST context (specifying the content type, authentication mode, encoding, and so on), and manages the connection. The connector relies on the Groovy scripting language and its RESTClient package.

  `SQL` - a scripted SQL connector, provided in the `org.forgerock.openicf.connectors.groovy.ScriptedSQLConnector` class. The scripted SQL connector uses Groovy scripts to interact with a JDBC database.

When you have selected a scripted connector implementation, write the required scripts that correspond to that connector type. Section 6.3, "Implementing OpenICF Operations With Groovy Scripts" provides information and examples on how to write scripts for the basic scripted connector implementation, and information on the extensions available for the other implementations.

6.3 Implementing OpenICF Operations With Groovy Scripts

The Groovy Connector Toolkit enables you to run a Groovy script for any OpenICF operation, such as search, update, create, and so forth, on any external resource.

You must write a Groovy script that corresponds to each operation that your connector will support. For information about all the operations that are supported by the OpenICF framework, see Chapter 4, "Implementing the OpenICF SPI".

Your scripted connector can implement the following OpenICF interfaces:

Authenticate
: Provides simple authentication with two parameters, presumed to be a user name and password.

Create
: Creates an object and its `uid`.

Delete
: Deletes an object, referenced by its `uid`.

Resolve Username
: Deletes an object, referenced by its `uid`.

Schema
: Describes the object types, operations, and options that the connector supports.

Script on Connector
: Enables an application to run a script in the context of the connector. Any script that runs on the connector has the following characteristics:

  - The script runs in the same execution environment as the connector and has access to all the classes to which the connector has access.
  - The script has access to a connector variable that is equivalent to an initialized instance of the connector. At a minimum, the script can access the connector configuration.
  - The script has access to any script-arguments passed in by the application.

Script on Resource
: Runs a script on the target resource that is managed by the connector.

Search
: Searches the target resource for all objects that match the specified object class and filter.

Sync
: Polls the target resource for synchronization events, that is, native changes to objects on the target resource.

Test
: Tests the connector configuration. Testing a configuration checks that all elements of the environment that are referred to by the configuration are available. For example, the connector might make a physical connection to a host that is specified in the configuration to verify that it exists and that the credentials that are specified in the configuration are valid.

  This operation might need to connect to a resource, and, as such, might take some time. Do not invoke this operation too often, such as before every provisioning operation. The test operation is not intended to check that the connector is alive (that is, that its physical connection to the resource has not timed out).

  You can invoke the test operation before a connector configuration has been validated.

Update
: Updates (modifies or replaces) objects on a target resource.

The following sections provide more information and sample scripts for all the operations that are implemented in the Groovy Connector toolkit. Sample scripts are provided for the basic scripted connector implementation, with additional information provided for the remaining implementations.

6.3.1 Variables Available to All Groovy Scripts

The following variables are available to all scripts used by the Groovy Connector. Additional variables are available to specific scripts, as described in the sections that follow:

`configuration`
: A handle to the connector's configuration object is injected into all scripts.

`operation`
: The connector injects the name of the action or operation into the script, to indicate which action is being called.

The samples for the Groovy connector define one script file per action. You can use a single file, or amalgamate multiple actions into one file. For example, the CREATE and UPDATE operations often share the same code.

The operation type can be one of the following:

- `ADD_ATTRIBUTE_VALUES`
- `AUTHENTICATE`
- `CREATE`
- `DELETE`
- `GET_LATEST_SYNC_TOKEN`
- `REMOVE_ATTRIBUTE_VALUES`
- `RESOLVE_USERNAME`
- `RUNSCRIPTONCONNECTOR`
- `RUNSCRIPTONRESOURCE`
- `SCHEMA`
- `SEARCH`
- `SYNC`
- `TEST`
- `UPDATE`

`options`

The OpenICF framework passes an `OperationOptions` object to most of the operations. The Groovy connector injects this object, as is, into the scripts. For example, the search, query, and sync operations pass the attributes to get as an operation option.

The most common options are as follows:

- `AttributesToGet` (String[]) for search and sync operations
- `RunAsUser` (String) for any operation
- `RunWithPassword` (GuardedString) for any operation
- `PagedResultsCookie` (String) for search operations

- `PagedResultsOffset` (Int) for search operations
- `PageSize` (Int) for search operations
- `SortKeys` (Sortkey[]) for search operations

objectClass
: The category or type of object that is managed by the connector, such as ACCOUNT and GROUP.

log
: A handle to the default OpenICF logging facility.

connection
: Available to the ScriptedREST, ScriptedCREST, and ScriptedSQL implementations, this variable initiates the HTTP or SQL connection to the resource.

6.3.2 Writing a Groovy Authentication Script

An authentication script is *required* if you want to use pass-through authentication to the backend resource. If your connector does not need to authenticate to the resource, the authentication script should allow the `authId` to pass through by default.

In addition to the common variables available to all scripts (see Section 6.3.1, "Variables Available to All Groovy Scripts"), the following variables are available to the authentication script:

username
: A string that provides the username to authenticate.

password
: A guarded string that provides the password with which to authenticate.

The authenticate script must return the user unique ID (`__UID__`). In the case of failure, the script must throw an exception.

Sample AuthenticateScript.groovy.

6.3.3 Writing a Groovy Test Script

A test script tests the connection to the external resource to ensure that the other operations that are provided by the connector can succeed.

The common variables are available to the test script (see Section 6.3.1, "Variables Available to All Groovy Scripts").

The script returns nothing if it is successful and can throw any exception if it fails.

Sample TestScript.groovy.

6.3.4 Writing a Groovy Create Script

A create script creates a new object on the external resource. If your connector does not support creating an object, this script should throw an `UnsupportedOperationException`. On success, the script should return a `Uid` object that represents the ID of the newly created object.

In addition to the common variables available to all scripts (see Section 6.3.1, "Variables Available to All Groovy Scripts"), the following variables are available to the create script:

attributes
: The set of attributes that describe the object to be created.

uid
: The UID of the object to be created, if specified. If the UID is `null`, the UID should be generated by the server. The UID corresponds to the OpenICF `__NAME__` attribute if it is provided as part of the attribute set.

The script must return the user unique ID (OpenICF `__UID__`).

Sample CreateScript.groovy.

6.3.5 Writing a Groovy Search Script

A search script searches for one or more objects on the external resource. Connectors that do not support searches should throw an `UnsupportedOperationException`. A search script should use the `handler` object to process the result set

In addition to the common variables available to all scripts (see Section 6.3.1, "Variables Available to All Groovy Scripts"), the following variables are available to the search script:

filter
: The query filter for this operation.

result
: A handle to process the search results.

Sample SearchScript.groovy.

6.3.6 Writing a Groovy Update Script

An update script updates an object in the external source. Connectors that do not support update operations should throw an `UnsupportedOperationException`. An update script must return the `uid` of the updated object.

In addition to the common variables available to all scripts (see Section 6.3.1, "Variables Available to All Groovy Scripts"), the following variables are available to the update script:

attributes
: The set of attributes that describe the object to be updated.

uid
: The UID of the object to be updated. The UID corresponds to the OpenICF `__NAME__` attribute if it is provided as part of the attribute set.

The script must return the user unique ID (OpenICF `__UID__`).

Sample UpdateScript.groovy.

6.3.7 Writing a Groovy Delete Script

A delete script deletes an object in the external resource. Connectors that do not support delete operations should throw an `UnsupportedOperationException`.

In addition to the common variables available to all scripts (see Section 6.3.1, "Variables Available to All Groovy Scripts"), the following variables are available to the delete script:

uid
: The UID of the object to be deleted. The UID corresponds to the OpenICF `UID` attribute.

objectClass
: The objectClass of the object to be deleted, for example, `ACCOUNT` or `GROUP`.

This script has no return value but should throw an exception if the delete is unsuccessful.

Sample DeleteScript.groovy.

6.3.8 Writing a Groovy Synchronization Script

A synchronization script synchronizes objects between two resources. The script should retrieve all objects in the external resource that have been updated since some defined token. The script should use the `handler` object to process the result set.

In addition to the common variables available to all scripts (see Section 6.3.1, "Variables Available to All Groovy Scripts"), the following variables are available to the sync script:

token
: The value of the sync token.

If the operation type is `GET_LATEST_SYNC_TOKEN`, the script must return an object that represents the last known `SyncToken` for the corresponding `ObjectClass`. If the operation type is `SYNC`, the script must return a new `SyncToken` for the corresponding `ObjectClass`.

Sample SyncScript.groovy.

6.3.9 Writing a Groovy Schema Script

A schema script builds the schema for the connector, either from a static, predefined schema, or by reading the schema from the external resource. The script should use the `builder` object to create the schema.

In addition to the common variables available to all scripts (see Section 6.3.1, "Variables Available to All Groovy Scripts"), the following variables are available to the schema script:

builder
: The schema builder object, an instance of `ICFObjectBuilder`.

Sample SchemaScript.groovy.

6.4 Advanced - Customizing the Configuration Initialization

Connectors created with the Groovy Connector Toolkit are, by default, stateful connectors. This means that the connector configuration instance is created only once.

The Groovy Connector Toolkit is precompiled code, and connector configurations are initialized in a specific way. If you have specific initialization requirements, you can customize the way in which the connector configuration instance is initialized, before the first script is evaluated. The `CustomizerScript.groovy` file enables you to define custom closures to interact with the default implementation.

The `CustomizerScript.groovy` file, provided with each compiled connector implementation, defines closures, such as `init {}`, `decorate {}`, and `destroy {}`. These closures are called during the lifecycle of the configuration.

When you unpack the Groovy Connector Toolkit JAR file, the `CustomizerScript.groovy` file is located at `org/forgerock/openicf/connectors/`*`connector-implementation`*.

Chapter 7

Building Connectors With the Custom Scripted Connector Bundler

OpenIDM 4.0.0 provides a utility that facilitates the creation, packaging and distribution of custom connectors based on the Groovy Connector Toolkit.

The Custom Scripted Connector Bundler enables you to create connectors using one of the default implementations described in *Selecting a Scripted Connector Implementation*. The utility takes a JSON configuration file as an argument, and produces all the required source code to build an OSGi-compatible `jar` file that can be used in any OpenICF-compliant project, such as OpenIDM.

The Connector Bundler generates a connector template that includes placeholders for meta-data, a structure for scripts, and a Maven project (`pom.xml`) file to instruct Maven how to build the connector JAR. To be able to use the connector in a deployment, you must populate the generated Groovy templates with code that is appropriate to the resource that your connector is accessing. The generated templates include notes and, where appropriate, "starter code" to help you develop your scripts.

OpenIDM 4.0.0 also provides a sample that shows how the connector bundler is used to create a scripted SQL connector. For more information, see *Sample 3 - Using the Groovy Connector Toolkit to Connect to MySQL With ScriptedSQL* in the *OpenIDM Installation Guide*.

7.1 Using the Connector Bundler

The Connector Bundler is provided as a JAR file in the OpenIDM delivery. To use the utility, download and extract OpenIDM, as described in *Installing and Running OpenIDM*, then follow these steps:

1. Navigate to the `tools` directory in your OpenIDM installation.

   ```
   $ cd /path/to/openidm/tools
   ```

2. The Connector Bundler takes a JSON configuration file as an argument. The configuration file includes the connector name and version, a description of the connector, the author, and any properties and object types that must be supported by the connector.

   To display a sample JSON configuration file, run the following command:

   ```
   $ java -jar custom-scripted-connector-bundler-4.0.0.jar -h
   Custom Scripted Connector Bundler for OpenIDM 4.0.0
   usage: bundle [OPTIONS] -c <FILE>
    -c,--config <file>   bundle configuration file for this connector
    -h,--help            print this help
    -v,--verbose         print the configuration file post-interpretation
   Configuration format:
   {
     "packageName" : "MyConnector",
     "displayName" : "My Connector",
     "description" : "This is my super awesome connector",
     "version" : "1.0",
     "author" : "Coder McLightningfingers",
     "providedProperties" : [ {
       "name" : "provided1",
       "value" : "default",
       "type" : "string"
     }, {
       "name" : "provided2",
       "value" : 2,
       "type" : "integer"
     } ],
     "properties" : [ {
       "order" : 0,
       "type" : "string",
       "name" : "FirstProperty",
       "value" : "firstValue",
       "required" : true,
       "confidential" : false,
       "displayMessage" : "This is my first property",
       "helpMessage" : "This should be a String value",
       "group" : "default"
     }, {
       "order" : 1,
       "type" : "double",
       "name" : "SecondProperty",
       "value" : 1.234,
       "required" : false,
       "confidential" : false,
       "displayMessage" : "This is my second property",
   ```

```
    "helpMessage" : "This should be a Double value",
    "group" : "default"
  } ],
  "objectTypes" : [ ],
  "baseConnectorType" : "GROOVY"
}
```

3. Use the sample configuration file as a starting point for writing your own configuration file to describe your connector bundle.

   This example assumes that you are writing a connector for an SQL database named ExampleDB.

   Set the following properties in the configuration file:

   "packageName"

   The name of the connector package.

   For this example, the `packageName` is `ExampleDB`.

   "displayName"

   The name of the connector.

   For this example, the `displayName` is `ExampleDB Connector`.

   "description"

   Any string that describes your custom connector.

   For this example, the `description` is `An SQL connector that connects to the ExampleDB database`.

   "version"

   The version number of the connector. Set this to a string that you update incrementally, as you develop/publish your connector.

   For this example, the `version` is `1.0`.

   "author"

   The author of the connector.

   For this example, the `author` is `Babs Jensen`.

   "baseConnectorType"

   The type of scripted connector that you are creating. The connector type can be one of the following:

   `GROOVY` - a non-pooled Groovy connector
   `POOLABLEGROOVY` - a poolable Groovy connector
   `CREST` - a connector based on the ForgeRock Commons REST (CREST) API. The connector takes a schema configuration file to define the

attribute mapping from the OpenICF connector object to the CREST resource.
`REST` - a scripted REST connector that enables you to connect to any resource over HTTP/REST. The connector creates the HTTP/REST context (specifying the content type, authentication mode, encoding, and so on), and manages the connection. The connector relies on the Groovy scripting language and its RESTClient package.
`SQL` - a scripted SQL connector that uses Groovy scripts to interact with a JDBC database.

For this example, the `baseConnectorType` is `SQL`.

"providedProperties"

These properties are provided by the base configuration class for your connector. For example, if you choose `SQL` as the `baseConnectorType`, the base configuration class for the connector is `ScriptedSQLConfiguration`. This base class provides several properties, such as `"username"`, and `"password"`, by default. You do not need to define these properties, but you must provide a default value for them if you want them to appear in the provisioner configuration file. To specify a default value, you add them as `"providedProperties"` in this configuration file.

For a list of the properties that are provided by the base configuration class, see the Groovy Connector Reference documentation.

"properties"

These properties include any custom properties that your connector must support (properties that are not provided by the base configuration class). Define the following attributes for each custom property that you add here:

- `"order"` (integer) - the order in which this property is displayed. The property with the value `0` is displayed first, then the property with the value `1`, and so on.

- `"type"` (string) - the data type of the property, for example, `string`, `int` or `boolean`.

- `"name"` (string) - the name of the property.

- `"value"` - the default value of the property, if any. The format depends on the `"type"` specified.

- `"required"` (boolean) - `true` if this property is required, `false` if it is optional.

- `"confidential"` (boolean) - Specifies whether the property is confidential, and whether its value should be encrypted by the application when persisted.
- `"displayMessage"` (string) - Optional text message that is written to the `Messages.properties` resource bundle. These messages can be used by a UI.
- `"helpMessage"` (boolean) - Optional help message that is written to the `Messages.properties` resource bundle. These messages can be used by a UI.
- `"group"` (string) - Optional string that enables you to categorize properties into groups.

"objectTypes"

The object types that the connector will manage. If your application includes a mechanism that can populate this list (such as a schema script in OpenIDM), you do not need to define the object types at this stage. The list is converted directly to the `"objectTypes"` list in the provisioner configuration file.

Each object type definition must have the following properties:

- `"name"` (string) - the name of the object type.
- `"id"` (string) - the unique ID of the object type, for example, `__GROUP__`.
- `"type"` - the JSON schema type of the object, as defined in the JSON Schema Internet Draft.
- `"nativeType"` - the data type of the object that is expected by the OpenICF API. The `ConnectorUtil` class includes a utility to convert the `"type"` to the `"nativeType"`.
- `"objectClass"` - can be one of the following:

  ObjectClass.ACCOUNT
  ObjectClass.GROUP
  ObjectClass.ALL
  Any arbitrary string. Although the bundler will accept an arbitrary string, you might need to edit the resulting provisioner file and Groovy scripts to work with the arbitrary string.

This example defines an `account` and a `group` object class.

The complete configuration file for the ExampleDB connector follows:

```
{
  "packageName": "ExampleDB",
  "displayName": "ExampleDB Connector",
  "description": "Connector for the ExampleDB database",
  "version": "1.0",
  "author": "Babs Jensen",
  "baseConnectorType": "SQL",
  "providedProperties": [
    {
      "name": "username",
      "value": "changeme",
      "type": "string"
    },
    {
      "name": "password",
      "value": "changeme",
      "type": "string"
    },
    {
      "name": "fullName",
      "value": "myfullname",
      "type": "string"
    },
    {
      "name": "lastName",
      "value": "mylastname",
      "type": "string"
    },
    {
      "name": "organization",
      "value": "myorg",
      "type": "string"
    }
  ],
  "properties": [
    {
      "order": 0,
      "type": "String",
      "name": "Name",
      "value": "firstValue",
      "required": true,
      "confidential": false,
      "displayMessage": "This is my first property",
      "helpMessage": "This should be a String value",
      "group": "default"
    },
    {
      "order": 1,
      "type": "Double",
      "name": "SecondProperty",
      "value": 1.234,
      "required": false,
      "confidential": false,
      "displayMessage": "This is my second property",
      "helpMessage": "This should be a Double value",
      "group": "default"
    }
  ],
  "objectTypes": [
    {
      "name": "group",
      "id": "__GROUP__",
      "type": "object",
      "nativeType": "__GROUP__",
      "objectClass": "ObjectClass.GROUP",
      "properties": [
        {
          "name": "name",
          "type": "string",
          "required": true,
          "nativeName": "__NAME__",
          "nativeType": "string"
        },
        {
          "name": "gid",
          "type": "string",
          "required": true,
          "nativeName": "gid",
          "nativeType": "string"
        },
        {
          "name": "description",
          "type": "string",
          "required": false,
          "nativeName": "description",
          "nativeType": "string"
        },
        {
          "name": "users",
          "type": "array",
          "nativeName": "users",
          "nativeType": "object",
          "items":  {
            "type": "object",
            "properties": [
              {
                "name": "uid",
                "type": "string"
              }
            ]
          }
        }
      ]
    },
    {
      "name": "account",
      "id": "__ACCOUNT__",
      "type": "object",
      "nativeType": "__ACCOUNT__",
      "objectClass": "ObjectClass.ACCOUNT",
      "properties": [
        {
          "name": "firstName",
          "type": "string",
          "nativeName": "firstname",
          "nativeType": "string",
          "required": true
        },
        {
          "name": "email",
          "type": "string",
          "nativeName": "email",
          "nativeType": "string"
        },
        {
          "name": "password",
          "type": "string",
          "nativeName": "password",
          "nativeType": "string",
          "flags": [
            "NOT_READABLE",
            "NOT_RETURNED_BY_DEFAULT"
          ]
        },
        {
          "name": "uid",
          "type": "string",
          "nativeName": "__NAME__",
          "required": true,
          "nativeType": "string"
        },
        {
          "name": "fullName",
          "type": "string",
          "nativeName": "fullname",
          "nativeType": "string"
        },
        {
          "name": "lastName",
          "type": "string",
          "required": true,
          "nativeName": "lastname",
          "nativeType": "string"
        }
      ]
    }
  ]
}
```

4. Save your customised connector configuration file as a JSON file, in a clean directory, for example:

```
$ ls /path/to/openidm/ExampleDBconnector
ExampleDB.json
```

5. Run the Connector Bundler, providing the path to your connector configuration file:

```
$ java -jar custom-scripted-connector-bundler-4.0.0.jar \
 --config ../ExampleDBConnector/ExampleDB.json
```

   This step generates a directory tree of source files that can be used to build your custom connector.

6. Edit the Groovy script templates that are located in the `src/main/resources/script/<connectorname>` directory.

These scripts must be enhanced with the code that will perform the operations on the resource. For more information, see *Implementing OpenICF Operations With Groovy Scripts*.

7. When you have completed the scripts for all the operations that you want your connector to perform, build the custom connector as follows:

   ```
   $ mvn install
   ```

   Building the connector produces an OSGi-compatible jar file in the `./target` directory. Copy this jar file to your OpenICF-compatible project or distribute it for others to use.

8. A provisioner configuration file named `provisioner.openicf-`*`connector-name`*`.json` is included in the jar file. You must extract the provisioner file to the filesystem because it cannot be detected and used directly from within the connector jar file.

   Extract the provisioner configuration file as follows:

   ```
   $ jar -xvf connector-name.jar conf/provisioner.openicf-connector-name.json
   ```

9. The connector jar includes all the Groovy scripts (which are executed directly from the jar). Alternatively, you can extract the scripts to the filesystem and run them from there. To do so, change the path to the scripts in the extracted provisioner configuration file as follows:

   Change:

   ```
   "scriptRoots" : [
       "jar:file:&{launcher.install.location}/connectors/awesome-connector-1.0.jar!/script/awesome/"
   ],
   "classpath" : [
       "jar:file:&{launcher.install.location}/connectors/awesome-connector-1.0.jar!/script/awesome/"
   ]
   ```

   to

   ```
   "scriptRoots" : [
       "file:&{launcher.project.location}/<your_extracted_script_location>/"
   ],
   "classpath" : [
       "file:&{launcher.project.location}/<your_extracted_script_location>/"
   ],
   ```

Chapter 8

Writing Scripted Connectors With the PowerShell Connector Toolkit

You can use the PowerShell Connector Toolkit to create connectors that can provision any Microsoft system, including, but not limited to, Active Directory, MS SQL, MS Exchange, Sharepoint, Office365, and Azure. Essentially, any task that can be performed with PowerShell can be executed through connectors based on this toolkit.

The PowerShell Connector Toolkit is not a complete connector, in the traditional sense. Rather, it is a framework within which you must write your own PowerShell scripts to address the requirements of your Microsoft Windows ecosystem.

Connectors created with the PowerShell Connector Toolkit run on the .NET platform and require the installation of a .NET connector server on the Windows system. To install the .NET connector server, follow the instructions in Installing a .NET Connector Server. These connectors also require PowerShell V2.

The PowerShell Connector Toolkit is available, with a subscription, from ForgeRock Backstage. To install the connector, download the archive (`mspowershell-connector-1.4.1.0.zip`) and extract the `MsPowerShell.Connector.dll` to the same folder in which the Connector Server (`connectorserver.exe`) is located. OpenIDM Enterprise includes sample connectors for Active Directory, and scripts that will enable you to get started with this toolkit.

8.1 About the PowerShell Scripting Language

PowerShell combines a command-line shell and scripting language, built on the .NET Framework. For a comprehensive introduction to PowerShell, and examples of how to use it, see Microsoft's TechNet article, Scripting with Windows PowerShell.

Chapter 9

Troubleshooting Connectors

Sometimes it is difficult to assess whether the root of a problem is at the OpenICF or connector level, or at the application level.

If you are using OpenICF connectors with OpenIDM, you can adjust the log levels for specific parts of the system in the `path/to/openidm/conf/logging.properties` file.

The OpenICF API sets the `LoggingProxy` at a very high level. You can consider the Logging Proxy as the *border* between the application (OpenIDM) and the OpenICF framework.

To start a troubleshooting process, you should therefore enable the Logging Proxy and set it at a level high enough to provide the kind of information you need:

```
org.identityconnectors.framework.impl.api.LoggingProxy.level=FINE
```

```
org.identityconnectors.framework.impl.api.LoggingProxy.level=DEBUG
```

```

#Enable the LoggingProxy
org.identityconnectors.framework.impl.api.LoggingProxy.level=FINE

#Select the operation you want to trace, to trace all add:
org.identityconnectors.framework.api.operations.level=FINE

#To trace only some:
org.identityconnectors.framework.api.operations.CreateApiOp.level=FINE
org.identityconnectors.framework.api.operations.UpdateApiOp.level=FINE
org.identityconnectors.framework.api.operations.DeleteApiOp.level=FINE
```

The complete list of operations that you can trace is as follows:

```
AuthenticationApiOp
CreateApiOp
DeleteApiOp
GetApiOp
ResolveUsernameApiOp
SchemaApiOp
ScriptOnConnectorApiOp
ScriptOnResourceApiOp
SearchApiOp
SyncApiOp
TestApiOp
UpdateApiOp
ValidateApiOp
```

To enable logging in the remote Java Connector Server, edit the xml configuration file `/lib/framework/logback.xml` to uncomment the following line:

```
<logger name="org.identityconnectors.framework.impl.api.LoggingProxy" level="DEBUG" additivity="false">
    <appender-ref ref="TRACE-FILE"/>
</logger>
```

To enable logging in the remote .NET Connector Server, edit the configuration file `ConnectorServer.exe.config`, setting the following value to `true`

```
<add key="logging.proxy" value="false"/>
```

Appendix A. Release Levels & Interface Stability

This appendix includes ForgeRock definitions for product release levels and interface stability.

A.1 ForgeRock Product Release Levels

ForgeRock defines Major, Minor, and Maintenance product release levels. The release level is reflected in the version number. The release level tells you what sort of compatibility changes to expect.

Table A.1. 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

Release Label	Version Numbers	Characteristics
		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 - Can include backwards-compatible changes to Stable interfaces in the same Major

Release Label	Version Numbers	Characteristics
		release, and incompatible changes to Evolving interfaces • Can remove previously Deprecated functionality • Include changes present in previous Minor and Maintenance releases
Maintenance	Version: x.y.z	• Bring bug fixes • Are intended to be fully compatible with previous versions from the same Minor release

A.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.

Table A.2. 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.
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.

Index