SailPoint Custom Connectors

IdentityIQ Version: 5.5, 6.0, 6.1, 6.2, 6.3

This document describes how to create a custom connector using SailPoint's openconnector framework.

Document Revision History

Revision Date	Written/Edited By	Comments
Aug 2012	Michael Hovis	Initial Creation
Sep 2012	Michael Hovis	Minor corrective edits
Oct 2012	Michael Hovis	Removed OLE source code
Dec 2012	Jennifer Mitchell	Edits to clarify some details (required methods, etc.)
Sep 2013	Jennifer Mitchell	Updated for 6.1 (no changes to doc contents)
Feb 2014	Jennifer Mitchell	Updated for 6.2 (no content changes)
Aug 2014	Jennifer Mitchell	Add details for clarity; add references to example memory connector now available on Compass

Table of Contents

Introduction

IdentityIQ includes many connectors through which it can connect to external systems, both to read their data and to process provisioning requests. Occasionally, however, installations may need to create a custom connector to connect to an application for which IdentityIQ does not have a built-in connector.

Unless direct provisioning is required, custom connectors are seldom necessary because user data is generally available through extracts, reports, database connections, etc. IdentityIQ's built in connectors can consume data from delimited file extracts, database connections (JDBC), and SOAP, just to name a few. Even in cases where the data extract is not available in a readily digestible format, rules provided with these available connectors can parse or otherwise pre-process the data to prepare it for recording in IdentityIQ; the developer writing the rules must, of course, be familiar with file parsing in Java. However, when provisioning is required or if these built-in features do not meet the business need, writing a custom connector using the openconnector framework is a good option.

Requirements

The writer of the connector should be proficient in Java, comfortable working within the constraints of a software framework, and versed in the use of an IDE such as Eclipse or Netbeans. The writer should also have at least completed the Fundamentals of IdentityIQ Implementation training course.

Design

Connector design involves identifying the features the connector will support, the object types it will read or provision, and the object schemas.

1. Identify the desired features for the connector. Available features are:
    - AUTHENTICATE: will the connector support authenticating a user and password combination against the resource
    - CREATE: will the connector be able to create a resource object (e.g. account or group)
    - DELETE: will the connector be able to delete a given resource object (by ID)
    - DISCOVER_SCHEMA: will the connector report schemas (account or group) to IdentityIQ when queried, including both attribute names and types (valid types are BOOLEAN, DATE, INT, LONG, PERMISSION, SECRET, STRING)
    - ENABLE: will the connector support enabling or disabling of a resource object
    - GET: will the connector be able to read and return a map of a specific resource object (by ID)
    - ITERATE: will the connector return an iterator of resource objects (possibly a filtered set)
    - PERMISSIONS: will the connector return permission attributes
    - SET_PASSWORD: will the connector set the password of a resource object
    - UNLOCK: will the connector be able to unlock a resource object
    - UPDATE: will the connector support updates of resource objects

NOTE: These features are listed as an enumeration in the openconnector.connector class, but this enumeration is not currently used in determining connector functionality; it is provided only as a connector design guide. The Application object also has a Feature enumeration, specified as the FeaturesString attribute of the application definition, which *is* used to control the functionality available through the connector (as discussed in the *Integrating with IdentityIQ* section of this document). The Connector Feature enumeration overlaps with but is not identical to the Application Feature enumeration. They correlate as shown here:

Connector Features Enumeration	Application Features Enumeration
AUTHENTICATE	AUTHENTICATE
CREATE	PROVISIONING
DELETE	PROVISIONING
DISCOVER_SCHEMA	DISCOVER_SCHEMA
ENABLE	ENABLE and PROVISIONING together
GET	opposite of NO_RANDOM_ACCESS (this is an assumed function for applications unless explicitly turned off)
ITERATE	no specific feature: assumed functionality for all applications
PERMISSIONS	DIRECT_PERMISSIONS
SET_PASSWORD	PASSWORD and PROVISIONING together
UNLOCK	UNLOCK and PROVISIONING together
UPDATE	PROVISIONING

2. Determine the object types the connector must support. Generally these are **account** and **group**, though other object types can be defined if required.
3. Identify the object schemas and their attributes' datatypes. The connector's schema can support these data types: Boolean, Date, Int, Long, Permission (SailPoint class containing rights and targets), Secret (an attribute that should not be logged or displayed to any end users), and String.

Implementing the openconnector

The architecture uses the openconnector framework provided by SailPoint in the openconnector package. All custom connectors must inherit from openconnector.AbstractConnector, which in turn implements the openconnector.Connector interface. The connectors are stateful and require configuration, use, and finally closure. The connector SDK, which includes the openConnector classes and the javadocs, is included in the IdentityIQ zip file for any GA version, in the /integration/connector/openConnector.zip file.

Using the openconnector framework, developers must implement separate methods for different types operations. These methods are automatically called by the AbstractConnector class' provision() method after it parses the provisioning plan passed to it. This simplifies development since it relieves the developer of the responsibility for parsing and interpreting the provisioning plan in the custom connector's code. The provision() method can, of course, be overridden with custom logic in the connector, but it is usually preferable to leverage the built-in functionality.

To create a custom connector based on this framework, complete these steps:

1. Start a new Java project, adding all the IdentityIQ libraries to the build path (these are located in IIQ's WEB-INF/lib directory), or leverage the services standard build infrastructure (documented and available from Compass) for the build process.
1. Create a new package called openconnector.
2. Create a new class in the openconnector package with the desired class name for the new connector. This image shows how to do this in Eclipse, but any IDE can be used.

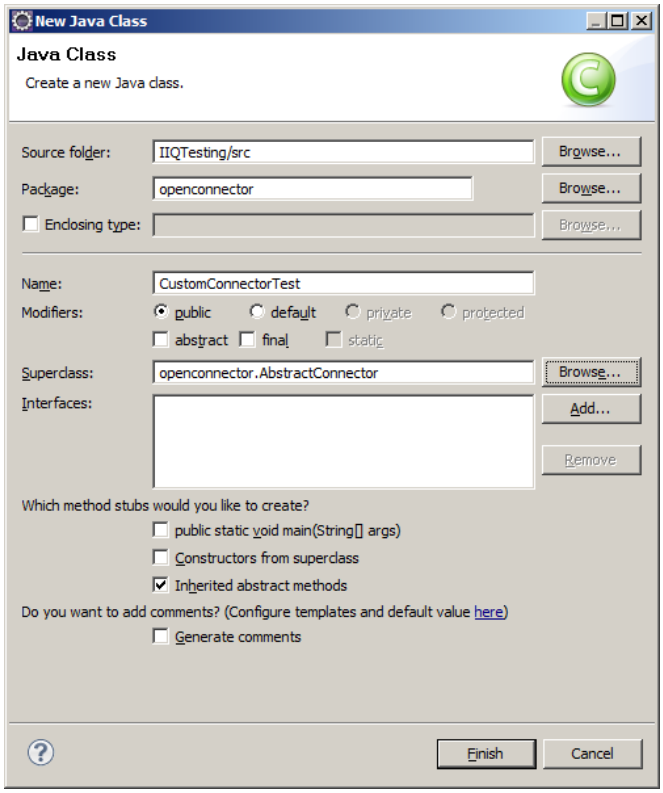

Figure 1: Create Java Class

3. Implement the methods required to support the selected features. Methods marked with an asterisk (*) are required for any custom connector, regardless of the feature set chosen, though the custom connector can simply defer to the AbstractConnector's default behavior instead of implementing its own version of the method when that method has been defined in the AbstractConnector class. AbstractConnector's provided functionality for each of these methods is noted below. In some cases, it provides some default functionality that can be used or overwritten by the custom class; in other cases, it contains no method or contains a method that throws an unsupported operation exception, both of which require the method to be written in the custom connector class for the operation to be supported.

Method	Description	Default Behavior of AbstractConnector's Method
getSupportedObjectTypes() *	Should return a list of the connector's supported object types	Returns Account

getSupportedFeatures()	Should return the list of supported features for the connector **NOTE**: This method is not currently used by IdentityIQ; instead, the application object's FeaturesString is used to determine the connector's supported features	Returns GET and ITERATE
configure()*	Should set up any configuration information which might be helpful in supporting other connector functionality; this is called only once during connector construction **NOTE**: The default behavior is helpful since it makes the connector's attributes available to other methods, so if this method is implemented in the custom connector class, it should call its super method as well.	Saves the connector config and log objects passed to it in local variables for later use by other methods
setObjectType()*	Should set the object type, which can influence the read, iterate, and provision behaviors of the connector when the methods supporting those functions contain separate functionality for each object type; is called by the OpenConnectorAdapter to set the correct target object type before executing operations	Sets the object type to the type named in the method call (defaulting to "account" if none named) and retrieves the schema for that type from the connector config
close()	Should perform any required steps to close the connection to the application	Does nothing
testConnection()*	Should determine if the resource is accessible; e.g. connect to the application with login credentials pulled from the connector configuration	Throws unsupported
discoverSchema()	Should return the schema (attributes and data types) of the currently selected object type	Throws unsupported
authenticate()	Should connect to the application with the credentials provided in the method call and attempt to authenticate to the application with that user id and password	Throws unsupported
read()	Should refresh the account or group in IdentityIQ corresponding to the provided native identity to match its current values in the application (i.e. targeted aggregation functionality) This can also be used in preparation for an	Not implemented

	update operation to ensure IdentityIQ has the latest data from the application before performing any updates.	
iterate()	Should return an iterator over the accounts/groups in the application (can be a subset of accounts/groups based on a specified filter)	Not implemented
provision()	Should parse the provisioningPlan and drive provisioning activities **NOTE**: Most custom connectors will not implement this method but will instead of rely on the AbstractConnector's provision method to do the plan parsing.	Splits the plan into individual actions for each request type (delete, create, update, enable, etc.) and calls other methods to provide the requested action
delete() ^	Should delete the object from the application based on the native identifier	Throws unsupported
create() ^	Should create an account (or group) in the application given a native identifier and a list of attribute values	Throws unsupported
update() ^	Should update the values for the specified account or group, matched by native identifier	Throws unsupported
enable() ^	Should enable the object on the application based a native identifier	throws unsupported
disable() ^	Should disable the object on the application, matched by native identifier	Throws unsupported
unlock() ^	Should unlock the account on the application based on native identifier	Throws unsupported
setPassword()	Should set/reset the account password on the application, given a native identifier, new password, optional current password, and expiration date	Throws unsupported

* Required method

^ These methods each accept a hashmap of options which can influence the operation. Any arguments passed in the request (within the provisioning plan) are passed to the target method by the provision() method to be used as needed by the connector.

NOTE: Methods can throw any of the following exceptions: ConnectorException, AuthenticationFailedException, UnsupportedOperationException, ObjectAlreadyExistsException, ObjectNotFoundException, or ExpiredPasswordException.

4. Compile the connector and deploy it in the IdentityIQ installations WEB-INF/lib directory. The example ant build script below could manage that compilation and deployment. (Replace directory names and project name with names appropriate to the specific installation.)

```
<project name="ProjectName" default="jar" basedir=".">
```

```
<property name="tomcat.home" value="../tomcat"/>
<property name="lib.dir"   value="${tomcat.home}/webapps/identityiq/WEB-INF/lib"/>
<property name="src.dir"   value="src"/>
<property name="dest.dir"  value="build/classes"/>
<property name="jar.dir"   value="build/jar"/>

    <path id="classpath">
        <fileset dir="${lib.dir}" includes="**/*.jar"/>
    </path>

    <target name="clean">
       <delete dir="build"/>
    </target>

    <target name="compile">
        <mkdir dir="${dest.dir}"/>
        <javac srcdir="${src.dir}" destdir="${dest.dir}" classpathref="classpath"
         includeantruntime="false"/>
    </target>

    <target name="jar" depends="compile">
       <mkdir dir="${jar.dir}"/>
        <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="build/classes"/>
    </target>
    <target name="deploy" depends="jar">
       <copy file="${jar.dir}/${ant.project.name}.jar" todir="${lib.dir}"/>
       <exec executable="/bin/bash">
              <arg value="${tomcat.home}/bin/shutdown.sh"/>
       </exec>
       <exec executable="/bin/bash">
              <arg value="${tomcat.home}/bin/startup.sh"/>
       </exec>
    </target>
</project>
```

Integrating with IdentityIQ

The final step in the implementation process is integrating the connector class with IdentityIQ. The class must be connected to an application type in IdentityIQ, and an application must be defined with that application type so the class will be used for interacting with the resource. This involves these steps:

1. Define the connector type in the ConnectorRegistry.
2. Define the .xhtml page which specifies required and optional connection parameters.
3. Create an application which uses the connector.

Define the ConnectorRegistry Entry

Create a ConnectorRegistry entry to define the application type. This is commonly done by creating and importing an XML file which modifies the ConnectorRegistry; alternatively, the ConnectorRegistry could be manually edited through the debug page or a whole copy of the ConnectorRegistry could be edited in XML and reimported with the new connector type defined in it.

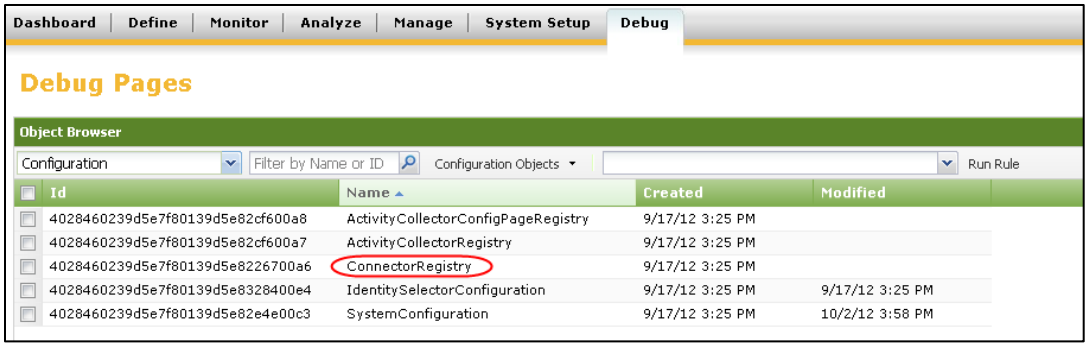

Figure 2: Accessing the Connector Registry through Debug Pages

The application definition for each connector in the ConnectorRegistry generically looks like this:

```
<Application connector="sailpoint.connector.OpenConnectorAdapter"
featuresString="[LIST OF SUPPORTED FEATURES IN ALL CAPS IN A CSV LIST]"
name="[CONNECTOR_NAME]" type="[CONNECTOR_TYPE_NAME]">
   <Attributes>
      <Map>
         <entry key="connectorClass" value="[FULLY QUALIFIED CUSTOM CONNECTOR NAME]"/>
         …
      </Map>
   </Attributes>
   <Schemas> (optional; one per object type, defining all the attributes for each)
     <Schema displayAttribute="UserID" groupAttribute="Groups"
     identityAttribute="UserID" nativeObjectType="account" objectType="account">
      <AttributeDefinition name="UserID" remediationModificationType="None"
       required="true" type="string"/>
        …
     </Schema>
     …
   </Schemas>
   <Templates> (optional; one per provisioning policy needed, naming all applicable fields)
        <Template name="account" usage="Create">
         <Field displayName="User ID" name="UserID" required="true" section=""
           type="string"/>
         …
        </Template>
        …
    </Templates>
</Application>
```

Specifically, these key attributes must be recorded in the application definition in the ConnectorRegistry:

- The **connector** attribute must be specified as ‘sailpoint.connector.OpenConnectorAdapter’. This tells IdentityIQ to use that class as a bridge to the internal SailPoint connector architecture from the custom openconnector implementation.
- The **connectorClass** attribute in the attributes map specifies the fully qualified class name of the custom connector class.
- The **name** and **type** values for the connector registry entry are usually the same value and reflect the name or type of application with which the connector interacts.
- **T**he **FeaturesString** list of features in the application definition is based on the Feature enumeration in the Application object; it tells IdentityIQ what operations the connector is configured to support. The features which can be enabled for custom connectors are:

- AUTHENTICATE: supports pass-through authentication
- DIRECT_PERMISSIONS: supports returning Permission objects
- DISCOVER_SCHEMA: supports discovering schemas for users and groups
- ENABLE: supports reading whether account is enabled or disabled, and if PROVISIONING is also specified, supports account enable/disable operations
- NO_RANDOM_ACCESS: does *not* support random access (e.g. getObject() methods) – random, or targeted, access is assumed possible by default and must be turned off with this option if it is not desired or supported
- UNLOCK: supports reading whether account is locked or unlocked, and if PROVISIONING is also specified, supports account unlock operation
- PROVISIONING: supports write functionality (e.g. account creation or deletion, entitlement update, unlock/enable/disable if those features are enabled too)
- PASSWORD: supports password updates (only applicable with PROVISIONING feature)

- The **Schemas** element (optional) can predefine account or group schemas for the application type.

Provisioning policies can be pre-specified in the **Templates** element (optional) to support provisioning, if enabled.The XML example below adds the CustomConnectorTest connector definition to the registry. It also illustrates predefining an account schema and a group schema as well as an account creation provisioning policy.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE sailpoint PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<sailpoint>
 <ImportAction name="merge">
  <Configuration name="ConnectorRegistry">
   <Attributes>
    <Map>
     <entry key="applicationTemplates">
       <value>
        <List>
         <Application connector="sailpoint.connector.OpenConnectorAdapter"
           featuresString="GROUP_PROVISIONING, PROVISIONING, SYNC_PROVISIONING,
           AUTHENTICATE, PASSWORD, ENABLE, SEARCH, UNSTRUCTURED_TARGETS,
           DISCOVER_SCHEMA, ACCOUNT_ONLY_REQUEST" icon="internetIcon"
           name="CustomConnectorTest" profileClass="" type="CustomConnectorTest">
             <Attributes>
              <Map>
               <entry key="authSearchAttributes">
                 <value>
                   <List>
                     <String>UserID</String>
                   </List>
                 </value>
               </entry>
               <entry key="compositeDefinition"/>
               <entry key="connectorClass" value="openconnector.CustomConnectorTest"/>
               <entry key="division" value="IT"/>
               <entry key="formPath" value="CustomConnectorTest.xhtml"/>
               <entry key="nativeChangeDetectionAttributeScope" value="entitlements"/>
               <entry key="nativeChangeDetectionAttributes"/>
               <entry key="nativeChangeDetectionEnabled">
                 <value>
                   <Boolean></Boolean>
                 </value>
               </entry>
               <entry key="nativeChangeDetectionOperations"/>
```

```
    </Map>
  </Attributes>
  <Schemas>
     <Schema displayAttribute="username" groupAttribute="groups"
      identityAttribute="username" instanceAttribute=""
      nativeObjectType="account" objectType="account">
       <AttributeDefinition name="username"
         remediationModificationType="None" type="string"/>
       <AttributeDefinition name="firstname"
         remediationModificationType="None" type="string"/>
       <AttributeDefinition name="lastname"
         remediationModificationType="None" type="string"/>
       <AttributeDefinition name="email" remediationModificationType="None"
         type="string"/>
       <AttributeDefinition multi="true" name="groups"
         remediationModificationType="None" type="string"/>
       <AttributeDefinition name="disabled"
         remediationModificationType="None" type="boolean"/>
       <AttributeDefinition name="locked"
         remediationModificationType="None" type="boolean"/>
       <AttributeDefinition name="password"
         remediationModificationType="None" type="string"/>
     </Schema>
     <Schema displayAttribute="name" identityAttribute="name"
      instanceAttribute="" nativeObjectType="group" objectType="group">
       <AttributeDefinition name="name" remediationModificationType="None"
        type="string"/>
       <AttributeDefinition name="description"
        remediationModificationType="None" type="string"/>
     </Schema>
   </Schemas>
   <Templates>
     <Template name="account" usage="Create">
       <Field displayName="User Name" name="username" type="string">
         <Script>
           <Source>return identity.getName(); </Source>
         </Script>
       </Field>
       <Field displayName="First Name" name="firstname" type="string">
         <Script>
           <Source>return identity.getFirstname(); </Source>
         </Script>
       </Field>
       <Field displayName="Last Name" name="lastname" type="string">
         <Script>
           <Source>return identity.getLastname(); </Source>
         </Script>
       </Field>
       <Field displayName="Email Address" name="email" type="string">
         <Script>
           <Source>return identity.getEmail(); </Source>
         </Script>
       </Field>
       <Field name="disabled" type="boolean" value="false"/>
     </Template>
     <Template name="Update Group" usage="UpdateGroup"/>
     <Template name="Group Creation" usage="CreateGroup">
       <Field displayName="Group Name" name="name" type="string"/>
       <Field displayName="Description" name="description" type="string"/>
     </Template>
   </Templates>
 </Application>
</List>
```

```
          </value>
        </entry>
      </Map>
    </Attributes>
  </Configuration>
</ImportAction>
</sailpoint>
```

Define the Connection Parameters XHTML Page

Usually, some parameters are required to define the connection to the target resource (e.g. host, port, username, password, etc.). An .xhtml page must be written to define how the Application Configuration user interface will request and record those parameters. This file must be placed in the [IdentityIQ Installation Directory]/define/application directory and must be referenced in the application definition's XML as the "formPath" entry (see the example above). The page's contents are automatically rendered on the Application Configuration's Attributes tab through the FacesServlet.

NOTE: If the application connection details will never be configured through IdentityIQ but instead the connection parameters will be specified in the Application XML object only and imported into the system, this .xhtml page is not strictly required. However, without it, the connection parameters for the application cannot be modified through the IdentityIQ user interface.

The example below illustrates definition of two required fields: input1 and input2. The CustomConnectorTest (described in Appendix A and available for download from Compass) is a memory connector which actually requires no connection parameters, so these two fields are defined just for illustration.

```
<!DOCTYPE html PUBLIC
      "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- (c) Copyright 2008 SailPoint Technologies, Inc., All Rights Reserved. -->

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:a4j="http://richfaces.org/a4j"
      xmlns:sp="http://sailpoint.com/ui">
<body>

<ui:composition>
    <script type="text/javascript">
      <!--//--><![CDATA[//><!--
        helpKey = 'APPLICATION';
        function getFieldValue(field) {
            var val = null;
            var fileName = $(field);
            if ( fileName ) {
               val = fileName.value;
            }
            return val;
        }
        Ext.onReady(function () {
           Ext.QuickTips.init();
```

```
        // This is our validation hook
        Page.on('beforeSave', function() {
            var input1 = getFieldValue('editForm:input1');
            Validator.validateNonBlankString(input1, "Input1 cannot be null!");
            var input2 = getFieldValue('editForm:input2');
            Validator.validateNonBlankString(email, "Input2 cannot be null!");

            var errors = Validator.getErrors();
            if (errors && errors.length > 0) {
                var errorDivTop = $('formErrorsTop');
                var errorDivBottom = $('formErrorsBottom');
                Validator.displayErrors(errorDivBottom);
                Validator.displayErrors(errorDivTop);
                return false;
            }
           //return false will kill cancel the save
            return true;
        });
    });
   //--><!]]>
</script>
<f:view>
  <h:outputText styleClass="sectionHeader"
     value="CustomConnector Test Settings"/><br/><br/>
  <div id='formErrorsTop' class="formError" style="display:none"/>


  <div id="accountSettings" class='spContent'>
    <table class="spTable" style="border:0px" cellspacing="0">
      <tr>
        <td class='titleColumn certScheduleTitleCol' valign="center">
          <h:outputText style='padding:5px' value="input1"/>
          <h:outputText styleClass="requiredText" value="*"/>
        </td>
        <td class="certScheduleHelpCol" valign="middle">
          <h:graphicImage id="imgHlpHostName" styleClass="helpIcon"
              url="/images/icons/dashboard_help_16.png"
              alt="input1"/>
        </td>
        <td valign="center">
          <h:inputText id="input1"
             value="#{applicationObject.attributes['input1']}"
             size="20"
             disabled="#{!sp:hasRight(facesContext, 'ManageApplication')}"
             readonly="#{!sp:hasRight(facesContext, 'ManageApplication')}"/>
        </td>
      </tr>
      <tr>
        <td class='titleColumn certScheduleTitleCol'>
          <h:outputText style='padding:5px' value="input2"/>
          <h:outputText styleClass="requiredText" value="*"/>
        </td>
        <td class="certScheduleHelpCol">
          <h:graphicImage id="imgHlpPort" styleClass="helpIcon"
             url="/images/icons/dashboard_help_16.png"
             alt="input2"/>
        </td>
        <td>
          <h:inputText id="input2"
            value="#{applicationObject.attributes['input2']}"
            size="20"
            disabled="#{!sp:hasRight(facesContext, 'ManageApplication')}"
            readonly="#{!sp:hasRight(facesContext, 'ManageApplication')}"/>
```

```
                </td>
              </tr>
            </table>
          </div>
          <div id='formErrorsBottom' class="formError" style="display:none"/>
        </f:view>
    </ui:composition>
    </body>
    </html>
```

Create the Application

Finally, an Application object must be created for the application, using the custom connector type. Use the IdentityIQ user interface (**Define** -> **Applications** menu option) to create the application, selecting the appropriate **Application Type** to tie it to the connector registry configuration and specifying any connection parameters through the Attributes tab UI.

Alternatively, a copy of the same application definition XML that was added to the connector registry could be independently imported as an application object with any required connection parameters hard coded into its attributes map to avoid having to define the .xhtml page at all. The **name** attribute specified in the application definition is the name displayed in IdentityIQ to represent the specific application instance; hence, unlike in the ConnectorRegistry, it value is often different from the **type** value in the individual application definition.

Testing

The integration console (run **iiq integration** from the [IdentityIQ Install Directory]/WEB-INF/bin directory) can be used to test the various features of the custom connector including aggregation and provisioning. From within that console, use the **list** command to view all configured integrations and connectors – the new application should be listed. Enter **use <app name>** to start testing the new adapter/connector. Type **?** at the command prompt to list all available commands in the integration console.

The IIQ console also offers commands which can be useful in connector testing. Use the **connectorDebug** command to test read operations, and use **provision** to test a provisioning plan directly against the connector.

Appendix A: Example Memory Connector

An example custom connector, implemented with the openconnector framework, is available for download from Compass to use as a model for developing other custom connectors. It is designed to read and write data from an application which is simply a hashmap stored in memory. It pre-creates a few accounts which allow it to be aggregated and automatically correlated with the data in the SailPoint Provisioning Training VM or the Fundamentals of IdentityIQ Implementation Training VM. The data in the application, other than the 3 hard-coded accounts, will be flushed any time the application server is restarted, so the application itself has limited usefulness in a real installation, but the artifacts provide solid examples of the piece required to implement a custom connector.

To deploy the memory connector into one of these two training VMs, complete the following steps:

1. Download the CustomConnectorTest.zip package from Compass into the VM and unzip it.
2. Copy the CustomConnectorTest.xhtml file to the /home/spadmin/tomcat/webapps/IdentityIQ/define/applications directory.
3. Import the CustomConnectorTest.xml file into the IdentityIQ instance; this updates the connector registry to add the new connector type.
4. Copy the CustomConnectorTest.java file to the /home/spadmin/ProvisioningTraining (or ImplementerTraining)/src directory. From the /home/spadmin/ProvisioningTraining (or ImplementerTraining) directory, run the **ant clean** and **ant deploy** commands to compile the connector and automatically place it in the tomcat runtime. The ant deploy command will also bounce the application server.
5. Log in to IdentityIQ as an administrator (e.g. spadmin) and configure a new application (**Define** -> **Applications**), choosing the CustomConnectorTest application type as its connector. Specify the required attributes on the Attributes tab and save the application definition.
6. Create and execute an account aggregation task and a group aggregation task for the application. Three new accounts and two groups will be aggregated. Two of the accounts will correlate to existing users (Catherine.Simmons and Aaron.Nichols) and one will not; this third account can be used to test delete operations. (**NOTE**: Correlation is done based on the nativeIdentity value matching an Identity name – this is IdentityIQ's default correlation algorithm when no other correlation config/rule is specified.)
7. Test various operations against the application through LifecycleManager; new accounts can be created, entitlements can be updated, accounts can be disabled/enabled, etc.

To deploy the memory connector into an IdentityIQ instance unrelated to the training instances, complete the following steps:

1. Download the CustomConnectorTest.zip package from Compass and unzip it.
2. Copy the CustomConnectorTest.xhtml file to the [IdentityIQ installation directory]/define/applications directory.
3. Import the CustomConnectorTest.xml file into the IdentityIQ instance; this updates the connector registry to add the new connector type.

4. Compile the CustomConnectorTest.java file against the IdentityIQ jar files and deploy it into the IdentityIQ installation, as generically described in *Implementing the openconnector* above. Stop and restart the app server.
5. Log in to IdentityIQ as an administrator (e.g. spadmin) and configure a new application (**Define** -> **Applications**), choosing the CustomConnectorTest application type as its connector. Specify the required attributes on the Attributes tab and save the application definition.
6. Create and execute an account aggregation task and a group aggregation task for the application. Three new accounts and two groups will be created; the accounts will most likely not correlate to existing users. Optionally, the CustomConnectorTest.java file could be modified before compilation to specify account names which would correlate to existing users in the target installation.
7. Test various operations against the application through LifecycleManager; new accounts can be created, entitlements can be updated, accounts can be disabled/enabled, etc.

To use this as a model for implementing other custom connectors, examine the .java file for examples of how to implement each of the custom connector methods. These, of course, will contain different logic in other custom connectors, allowing them to work with the API for the target application or whatever other infrastructure is available to connect to and communicate with the application.